Основы многопоточного, параллельного и распределенного программирования

Foundations of Multithreaded, Parallel, and Distributed Programming

GREGORY R. ANDREWS
University of Arizona

ADDISON-WESLEY
An imprint of Addison Wesley Longman, Inc.
Reading, Massachusetts • Menlo Park, California
Ney York • Harlow, England • Don Mills, Ontario
Sydney • Mexico City • Madrid • Amsterdam

Основы многопоточного, параллельного и распределенного программирования

ГРЕГОРИ Р. ЭНДРЮС
Университет штата Аризона

Москва • Санкт-Петербург • Киев
2003

ББК 32.973.26-018.2.75
Э64
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *А.В. Слепцов*

Перевод с английского *А.С. Подосельника*, *Г.И. Сингаевской*,
канд.физ.-мат.наук *А.Б. Ставровского*

Под редакцией канд.физ.-мат.наук *А.Б. Ставровского*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Эндрюс Г.Р.

Э64 Основы многопоточного, параллельного и распределенного программирования. : Пер. с англ. — М. : Издательский дом "Вильямс", 2003. — 512 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0388-2 (рус.)

В книге рассматриваются важнейшие концепции многопоточного, параллельного и распределенного программирования, которые должен знать каждый программист, создающий программное обеспечение подобного типа. Все обсуждаемые концепции и методы тщательно проиллюстрированы многочисленными примерами, написанными на основных языках программирования с использованием наиболее распространенных библиотек. Обсуждение каждого учебного примера включает описание соответствующих элементов используемого языка или библиотеки и содержит полный текст прикладной программы. В книге освещаются общие механизмы параллельного программирования с использованием разделяемых переменных, основные концепции распределенного программирования и механизмы взаимодействия и синхронизации процессов с помощью обмена сообщениями. Заключительная часть книги посвящена обсуждению применения методов параллельного программирования при проведении сложных научных вычислений.

Книга может быть полезна как студентам, изучающим соответствующие курсы, так и специалистам-практикам в области разработки программного обеспечения.

ББК 32.973.26-018.2.75

ISBN 5-8459-0388-2 (рус.)
ISBN 0-2013-5752-6 (англ.)

Оглавление

Содержание

Предисловие

Параллельное программирование возникло в 1962 г. с изобретением каналов — независимых аппаратных контроллеров, позволявших центральному процессору выполнять новую прикладную программу одновременно с операциями ввода-вывода других (приостановленных) программ. Параллельное программирование (слово *параллельное* в данном случае означает "происходящее одновременно"[1]) первоначально было уделом разработчиков операционных систем. В конце 60-х годов были созданы многопроцессорные машины. В результате не только были поставлены новые задачи разработчикам операционных систем, но и появились новые возможности у прикладных программистов.

Первой важной задачей параллельного программирования стало решение проблемы так называемой *критической секции*. Эта и сопутствующие ей задачи ("обедающих философов", "читателей и писателей" и т.д.) привели к появлению в 60-е годы огромного числа научных работ. Для решения данной проблемы и упрощения работы программиста были разработаны такие элементы синхронизации, как семафоры и мониторы. К середине 70-х годов стало ясно, что для преодоления сложности, присущей параллельным программам, необходимо использовать формальные методы.

На рубеже 70-х и 80-х годов появились компьютерные сети. Для глобальных сетей стандартом стал Arpanet, а для локальных — Ethernet. Сети привели к росту распределенного программирования, которое стало основной темой в 80-х и, особенно, в 90-х годах. Суть распределенного программирования состоит во взаимодействии процессов путем передачи сообщений, а не записи и чтения разделяемых переменных.

Сейчас, на заре нового века, стала заметной необходимость обработки с массовым параллелизмом, при которой для решения одной задачи используются десятки, сотни и даже тысячи процессоров. Также видна потребность в технологии клиент-сервер, сети Internet и World Wide Web. Наконец, стали появляться многопроцессорные рабочие станции и ПК. Параллельное аппаратное обеспечение используется больше, чем когда-либо, а параллельное программирование становится необходимым.

Это моя третья книга, в которой предпринята попытка охватить часть истории параллельного программирования. Первая книга — *Concurrent Programming: Principles and Practice*, опубликованная в 1991 г., — дает достаточно полное описание периода между 1960 и 1990 годами. Основное внимание в ней уделяется новым задачам, механизмам программирования и формальным методам.

Вторая книга — *The SR Programming Language: Concurrency in Practice*, опубликованная в 1993 году, — подводит итог моей работы с Роном Олсоном (Ron Olsson) в конце 80-х и начале 90-х годов над специальным языком программирования, который может использоваться при написании параллельных программ для систем как с разделяемой, так и с распределенной памятью. Книга о языке SR является скорее практическим руководством, чем формальным описанием языка, поскольку демонстрирует пути решения многих проблем с использованием одного языка программирования.

Данная книга выросла из предыдущих и является отражением моих мыслей о том, что важно сейчас и в будущем. Многое почерпнуто из книги *Concurrent Programming*, но полностью переработан каждый раздел, взятый из нее, и переписаны все примеры программ на псевдоС вместо языка SR. Все разделы дополнены новым материалом; особенно это каса-

---

[1] Слово "параллельный" в большинстве случаев является переводом английского "concurrent". Применительно к программам, вычислениям и программированию его смысл, возможно, лучше передавался бы словом "многопроцессный", но этого термина в русскоязычной литературе нет. Перевод английского "parallel" связан с синхронным параллелизмом, и его смысл определяется в разделах 1.3 и 3.5 и уточняется в самом начале части 3. Отметим, что смысл слова "concurrent" шире, чем "parallel". — *Прим. ред.*

ется параллельного научного программирования. Также добавлены учебные примеры по наиболее важным языкам программирования и библиотекам программ, содержащие завершенные учебные программы. И, наконец, я по-новому вижу использование этой книги — в аудиториях и личных библиотеках.

Новое видение и новая роль

Идеи параллельных и распределенных вычислений сегодня распространены повсеместно. Как обычно в вычислительной технике, прогресс исходит от разработчиков аппаратного обеспечения, которые создают все более быстрые, большие и мощные компьютеры и коммуникационные сети. Большей частью, они достигают успеха, и лучшее доказательство тому — фондовая биржа!

Новые компьютеры и сети создают новые проблемы и возможности, а разработчики программного обеспечения по-прежнему не отстают. Вспомните Web-броузеры, Internet-коммерцию, потоки Pthread, язык Java, MPI, и вновь доказательством служит фондовая биржа! Эти программные продукты разрабатываются специально для того, чтобы использовать все преимущества параллельности в аппаратной и программной части. Короче говоря, большая часть мира вычислительной техники сейчас параллельна!

Аспекты параллельных вычислений — многопоточных, параллельных или распределенных — теперь рассматриваются почти в каждом углубленном курсе по вычислительной технике. Отражая историю, курсы по операционным системам охватывают такие темы, как многопоточность, протоколы взаимодействия и распределенные файловые системы. Курсы по архитектуре вычислительной техники — многопроцессорность и сети, по компиляторам — вопросы компиляции для параллельных машин, а теоретические курсы — модели параллельной обработки данных. В курсах по алгоритмам изучаются параллельные алгоритмы, а по базам данных — блокировка и распределенные базы данных. В курсах по графике используется параллелизм при визуализации и трассировке лучей. Этот список можно продолжить. Кроме того, параллельные вычисления стали фундаментальным инструментом в широкой области научных и инженерных дисциплин.

Главная цель книги, как видно из ее названия, — заложить основу для программирования многопоточных, параллельных и распределенных вычислений. Частная цель — создать текст, который можно использовать в углубленном курсе для студентов и магистров. Когда некоторая тема становится распространенной, как это произошло с параллелизмом, вводятся учебные курсы по ее основам. Аналогично, когда тема становится хорошо изученной, как сейчас параллелизм, она переносится в нормативный курс.

Я пытался охватить те вопросы параллельной и распределенной обработки данных, которые, по моему мнению, должен знать любой студент, изучающий вычислительную технику. Сюда включены основные принципы, методы программирования, наиболее важные приложения, реализации и вопросы производительности. Я добавил также материал по важнейшим языкам программирования и библиотекам программ — потоки Pthread (три главы), язык Java (три главы), CSP, Linda, MPI (две главы), языки Ада, SR и OpenMP. В каждом примере сначала описаны соответствующие части языка программирования или библиотеки, а затем представлена полная программа. Исходные тексты программ доступны на Web-сайте этой книги. Кроме того, в главе 12 приведен обзор нескольких дополнительных языков, моделей и инструментов для научных расчетов.

С другой стороны, ни в одной книге нельзя охватить все, поэтому, возможно, студенты и преподаватели захотят дополнить данный текст. Исторические справки и списки литературы в конце каждой главы описывают дополнительные материалы и содержат указания для дальнейшего изучения. Информация для дальнейшего изучения и ссылки на соответствующие материалы представлены на Web-сайте этой книги.

Обзор содержания

Эта книга состоит из 12 глав. В главе 1 дается обзор основных идей параллелизма, аппаратной части и приложений. Затем рассматриваются пять типичных приложений: умножение матриц, адаптивная квадратура, каналы ОС Unix, файловые системы и распределенное умножение матриц. Каждое приложение просто, но, тем не менее, полезно: вместе они иллюстрируют некоторый диапазон задач и пять стилей программирования многократных вычислений. В последнем разделе главы 1 резюмируется программная нотация, использованная в книге.

Оставшиеся главы разделены на три части. Часть 1 описывает механизмы параллельного программирования, которые используют разделяемые переменные и поэтому непосредственно применяются к машинам с разделяемой памятью. В главе 2 представлены базовые понятия процессов и синхронизации; основные моменты иллюстрируются рядом примеров. Заканчивается глава обсуждением формальной семантики параллелизма. Понимание семантических концепций поможет вам разобраться в некоторых разделах последующих глав. В главе 3 показано, как реализовать и использовать блокировки и барьеры, описаны алгоритмы, параллельные по данным, и метод параллельного программирования, называемый "портфель задач". В главе 4 представлены семафоры и многочисленные примеры их использования. Семафор был первым механизмом параллельного программирования высокого уровня и остается одним из важнейших. В главе 5 подробно описаны мониторы. Они появились в 1974 г.; в 80-е и в начале 90-х годов внимание к ним несколько угасло, но появление языка Java возобновило интерес к ним. Наконец, в главе 6 представлена реализация процессов, семафоров и мониторов на одно- и многопроцессорных машинах.

Часть 2 посвящена распределенному программированию, в котором процессы взаимодействуют и синхронизируются, используя сообщения. В главе 7 описана передача сообщений с помощью элементарных операций `send` и `receive`. Демонстрируется, как использовать эти операции для программирования фильтров (программ с односторонней связью), клиентов и серверов (с двусторонней связью), а также взаимодействующих равных (с передачей в обоих направлениях). В главе 8 рассматриваются два дополнительных примитива взаимодействия: удаленный вызов процедуры (RPC) и рандеву. Процесс-клиент инициирует связь, посылая сообщение `call` (неявно оно является последовательностью сообщений `send` и `receive`). Взаимодействие обслуживается или новым процессом (RPC), или с помощью рандеву с существующим процессом. В главе 9 описано несколько моделей взаимодействия процессов в распределенных программах. Три из них обычно используются в параллельных вычислениях — "управляющий-рабочие", алгоритм пульсации и конвейер. Еще четыре возникли в распределенных системах — зонд-эхо, оповещение (рассылка), передача маркера и дублируемые серверы. Наконец, в главе 10 описана реализация передачи сообщений, RPC и рандеву. Показано, как реализовать так называемую распределенную разделяемую память, которая поддерживает модель программирования с разделяемой памятью в распределенной среде.

Часть 3 посвящена синхронному параллельному программированию, особенно его применению к высокопроизводительным научным вычислениям. (Многие другие типы синхронных параллельных вычислений рассмотрены в предыдущих главах и в упражнениях к нескольким из них.) Цель параллельной программы — ускорение, т.е. более быстрое решение задачи с помощью большого числа процессоров. Синхронные параллельные программы пишутся с использованием разделяемых переменных или передачи сообщений, следовательно, в них применяется методика, описанная в частях 1 и 2. В главе 11 рассматриваются три основных класса приложений для научных вычислений: сеточные, точечные и матричные. Они возникают при моделировании физических и биологических систем; матричные вычисления используются и для таких задач, как экономическое прогнозирование. В главе 12 дан обзор наиболее важных инструментов для написания научных параллельных вычислительных программ: библиотеки (Pthread, MPI, OpenMP), распараллеливающие компиляторы, языки и модели, а также такие инструменты более высокого уровня, как метавычисления.

В конце каждой главы размещены историческая справка, ссылки на литературу и упражнения. В исторической справке резюмируются происхождение, развитие и связи каждой темы с другими темами, а также описываются статьи и книги из списка литературы. В упражнениях представлены вопросы, поднятые в главе, и дополнительные приложения.

Использование в учебном процессе

Я использую эту книгу каждой весной в университете штата Аризона, читая курс примерно для 60 студентов. Около половины из них — магистры; остальные — студенты старших курсов. В основном это студенты специальности "computer science". Данный курс для них не обязателен, но большинство студентов его изучают. Также каждый год курс проходят несколько аспирантов других отделений, интересующихся вычислительной техникой и параллельными вычислениями. Большинство студентов одновременно проходят и наш курс по операционным системам.

В наших курсах по ОС, как и везде, рассматриваются *потребности* ОС в синхронизации и изучается реализация синхронизации, процессов и других элементов ОС, в основном, на однопроцессорных машинах. Мой курс демонстрирует, как *использовать* параллельность в качестве общего инструмента программирования для широкого диапазона приложений. Я рассматриваю методы программирования, концепции высокого уровня, параллельную и распределенную обработку данных. По существу, мой курс относится к курсу по ОС примерно так же, как сравнительный курс языков программирования относится к курсу по компиляторам.

В первой половине моего курса я подробно излагаю главу 2, а затем быстро прохожу по остальным главам первой части, акцентируя внимание на темах, не рассматриваемых в курсе по ОС, — протокол "проверить-проверить-установить" (Test-and-Test-and-Set), алгоритм билета, барьеры, передача эстафеты для семафоров, некоторые способы программирования мониторов и многопроцессорное ядро. В дополнение к этому студенты самостоятельно изучают библиотеку Pthread, потоки языка Java и синхронизированные методы. После лекций по барьерам они выполняют проект по синхронным параллельным вычислениям (на основе материала главы 11).

Во второй половине курса я использую многое из материала части 2 книги, касающегося распределенного программирования. Мы рассматриваем передачу сообщений и ее использование в программировании как распределенных систем, так и распределенных параллельных вычислений. Затем мы изучаем RPC и рандеву, их воплощение в языках Java и Ada и приложения в распределенных системах. Наконец, мы рассматриваем каждую парадигму из главы 9, вновь используя для иллюстрации и мотивировки приложения из области синхронных параллельных и распределенных систем. Самостоятельно студенты изучают библиотеку MPI и снова используют язык Java.

В течение семестра я даю четыре домашних задания, два аудиторных экзамена, проект по параллельным вычислениям и заключительный проект. (Примеры представлены на Web-сайте книги.) Каждое домашнее задание состоит из письменных упражнений и одной или двух задач по программированию. Магистры должны выполнить все упражнения, другие студенты — две трети задач (по своему выбору). Экзамены проводятся аналогично: магистры решают все задачи, а остальные выбирают вопросы, на которые хотят отвечать. В Аризонском университете начальные задачи по программированию мы решаем с помощью языка SR, который студенты могут использовать и в дальнейшем, но поощряется использование таких языков и библиотек, как Pthreads, Java и MPI.

Проект по синхронному параллельному программированию связан с задачами из главы 11 (обычно это сеточные вычисления). Студенты пишут программы и экспериментируют на небольшом мультипроцессоре с разделяемой памятью. Магистры реализуют более сложные алгоритмы и обязаны написать подробный отчет о своих опытах. Заключительный проект — это статья или программный проект по какому-либо вопросу распределенного программирования. Студенты выбирают тему, которую я утверждаю. Большинство студентов выполняют

проекты по реализации, многие из них работают парами. Студенты создают разнообразные интересные системы, в основном, с графическим интерфейсом.

Несмотря на то что студенты, изучающие мой курс, имеют различную подготовку, почти все оценивают его отлично. Студенты часто отмечают, насколько хорошо курс согласуется с их курсом по ОС; им нравится взгляд на параллельность с другой точки зрения, изучение многопроцессорных систем, им интересно рассматривать широкий спектр приложений и инструментов. Магистры отмечают, что курс связывает воедино разные вещи, о которых они уже что-то слышали, и вводит их в область параллельного программирования. Однако многим из них было бы интересно изучить параллельные вычисления более подробно. Со временем мы надеемся сделать отдельные курсы для магистров и студентов. Я не буду значительно изменять курс для студентов, но в нем будет меньше углубленных тем. В курсе для магистров я буду тратить меньше времени на материал, с которым они уже знакомы (часть 1), и больше времени посвящать синхронным параллельным приложениям и инструментарию синхронных вычислений (часть 3).

Эта книга идеально подходит для курса, который охватывает область механизмов параллельного программирования, средств и приложений. Почти все студенты смогут использовать *что-то* из рассмотренного здесь, но не все будут заниматься только параллельной обработкой данных, только распределенными системами или программировать только на языке Java. Тем не менее книгу можно использовать и как пособие в более специализированных курсах, например, в курсе синхронных параллельных вычислений, вместе с книгами по параллельным алгоритмам и приложениям.

Информация в Internet

"Домашняя страничка" этой книги находится по адресу:

```
http://www.cs.arizona.edu/people/greg/mpdbook
```

На этом сайте есть исходные тексты программ из книги, ссылки на программное обеспечение и информацию о языках программирования и библиотеках, описанных в примерах; большое число других полезных ссылок. Также сайт содержит обширную информацию о моем курсе в Аризонском университете, включая программу курса, конспекты лекций, копии домашних заданий, проектов и экзаменационных вопросов. Информация об этой книге также доступна по адресу:

```
http://www.awlonline.com/cs
```

Несмотря на мои усилия, книга, без сомнения, содержит ошибки, так что по этому адресу появится (уверен, что скоро) их список. Безусловно, в будущем добавятся и другие материалы.

Благодарности

Я получил множество полезных замечаний от читателей чернового варианта этой книги. Марти Тиенари (Martti Tienary) и его студенты из университета Хельсинки обнаружили несколько неочевидных ошибок. Мои последние аспиранты, Винс Фрих (Vince Freeh) и Дейв Ловентал (Dave Lowenthal), прокомментировали новые главы и за последние несколько лет помогли отладить если не мои программы, то мои мысли. Студенты, изучавшие мой курс весной 1999 г., служили "подопытными кроликами" и нашли несколько досадных ошибок. Энди Бернат (Andy Bernat) предоставил отзыв о своем курсе в университете в Эль-Пасо, Техас, который он провел весной 1999 г. Благодарю следующих рецензентов за их бесценные отзывы: Джаспал Субхлок (Jaspal Subhlok) из университета Carnegy Mellon, Болеслав Жимански (Boleslaw Szymansky) из политехнического института Rensselaer, Дж. С. Стайлс (G. S. Stiles) из Utah State University, Нарсингх Део (Narsingh Deo) из Central Florida University, Джанет Хартман (Janet Hartman) из Illinois State University, Нэн Шаллер (Nan C. Schaller) из Технологического института Рочестера и Марк Файнап (Mark Fineup) из университета Северной Айовы.

В течение многих лет организация National Science Foundation поддерживает мои исследования, в основном по грантам CCR-9415303 и ACR-9720738. Грант от NSF (CDA-9500991) помог обеспечить оборудование для подготовки книги и проверки программ.

И главное — я хочу поблагодарить мою жену Мэри, еще раз смирившуюся с долгими часами, которые я провел, работая над этой книгой (несмотря на клятвы вроде "больше никогда" после завершения книги о языке SR).

Грег Эндрюс
Таксон, Аризона

Глава 1

Обзор области параллельных вычислений

Представьте себе такую картину: несколько автомобилей едут из пункта А в пункт В. Машины могут бороться за дорожное пространство и либо следуют в колонне, либо обгоняют друг друга (попадая при этом в аварии!). Они могут также ехать по параллельным полосам дороги и прибыть почти одновременно, не "переезжая" дорогу друг другу. Возможен вариант, когда все машины поедут разными маршрутами и по разным дорогам.

Эта картина демонстрирует суть параллельных вычислений: есть несколько задач, которые должны быть выполнены (едущие машины). Можно выполнять их по одной на одном процессоре (дороге), параллельно на нескольких процессорах (дорожных полосах) или на распределенных процессорах (отдельных дорогах). Однако задачам нужно синхронизироваться, чтобы избежать столкновений или задержки на знаках остановки и светофорах.

Данная книга — это "атлас" параллельных вычислений. В ней рассматриваются типы автомашин (процессов), возможные пути их следования (приложения), схемы дорог (аппаратного обеспечения) и правила дорожного движения (взаимодействие и синхронизация). Так что заправьте полный бак и приготовьтесь к старту.

В данной главе рассказывается о надписях на карте параллельного программирования. В разделе 1.1 представлены основные понятия. В разделах 1.2 и 1.3 описаны виды аппаратной части и приложения, которые делают параллельное программирование интересным и перспективным. В разделах с 1.4 по 1.8 описываются и иллюстрируются пять стилей программирования циклических вычислений: итеративный параллелизм, рекурсивный параллелизм, "производители и потребители", "клиенты и серверы" и, наконец, взаимодействующие каналы. В последнем разделе определена нотация программ, используемая в дальнейшем.

В следующих главах подробно рассмотрены приложения и методы программирования. Книга состоит из трех частей, в которых описано программирование с разделяемыми переменными, распределенное (основанное на сообщениях) и синхронное параллельное. Введение в каждую часть и главу служит картой маршрута, подводя итоги пройденного и предстоящего пути.

1.1. Суть параллельного программирования

Параллельная программа содержит несколько процессов, работающих совместно над выполнением некоторой задачи. Каждый процесс — это последовательная программа, а точнее — последовательность операторов, выполняемых один за другим. Последовательная программа имеет *один поток управления*, а параллельная — *несколько*.

Совместная работа процессов параллельной программы осуществляется с помощью их *взаимодействия*. Взаимодействие программируется с применением разделяемых переменных или пересылки сообщений. Если используются разделяемые переменные, то один процесс осуществляет запись в переменную, считываемую другим процессом. При пересылке сообщений один процесс отправляет сообщение, которое получает другой.

При любом виде взаимодействия процессам необходима взаимная *синхронизация*. Существуют два основных вида синхронизации — взаимное исключение и условная синхрониза-

ция. *Взаимное исключение* обеспечивает, чтобы критические секции операторов не выполнялись одновременно. *Условная синхронизация* задерживает процесс до тех пор, пока не выполнится определенное условие. Например, взаимодействие процессов производителя и потребителя часто обеспечивается с помощью буфера в разделяемой памяти. Производитель записывает в буфер, потребитель читает из него. Чтобы предотвратить одновременное использование буфера и производителем, и потребителем (тогда может быть считано не полностью записанное сообщение), используется взаимное исключение. Условная синхронизация используется для проверки, было ли считано потребителем последнее записанное в буфер сообщение.

Как и другие прикладные области компьютерных наук, параллельное программирование прошло несколько стадий. Оно возникло благодаря новым возможностям, предоставленным развитием аппаратного обеспечения, и развилось в соответствии с технологическими изменениями. Через некоторое время *специализированные* методы были объединены в набор основных принципов и общих методов программирования.

Параллельное программирование возникло в 1960-е годы в сфере операционных систем. Причиной стало изобретение аппаратных модулей, названных *каналами*, или *контроллерами устройств*. Они работают независимо от управляющего процессора и позволяют выполнять операции ввода-вывода параллельно с инструкциями центрального процессора. Канал взаимодействует с процессором с помощью *прерывания* — аппаратного сигнала, который говорит: "Останови свою работу и начни выполнять другую последовательность инструкций".

Результатом появления каналов стала проблема программирования (настоящая интеллектуальная проблема) — теперь части программы могли быть выполнены в непредсказуемом порядке. Следовательно, пока одна часть программы обновляет значение некоторой переменной, может возникнуть прерывание, приводящее к выполнению другой части программы, которая тоже попытается изменить значение этой переменной. Эта специфическая проблема (*задача критической секции*) подробно рассматривается в главе 3.

Вскоре после изобретения каналов началась разработка многопроцессорных машин, хотя в течение двух десятилетий они были слишком дороги для широкого использования. Однако сейчас все крупные машины являются многопроцессорными, а самые большие имеют сотни процессоров и часто называются *машинами с массовым параллелизмом* (massively parallel processors). Скоро даже персональные компьютеры будут иметь несколько процессоров.

Многопроцессорные машины позволяют разным прикладным программам выполняться одновременно на разных процессорах. Они также ускоряют выполнение приложения, если оно написано (или переписано) для многопроцессорной машины. Но как синхронизировать работу параллельных процессов? Как использовать многопроцессорные системы для ускорения выполнения программ?

Итак, при использовании каналов и многопроцессорных систем возникают и возможности, и трудности. При написании параллельной программы необходимо решать, сколько процессов и какого типа нужно использовать, и как они должны взаимодействовать. Эти решения зависят как от конкретного приложения, так и от аппаратного обеспечения, на котором будет выполняться программа. В любом случае ключом к созданию корректной программы является правильная синхронизация взаимодействия процессов.

Эта книга охватывает все области параллельного программирования, но основное внимание уделяет *императивным программам* с явными параллельностью, взаимодействием и синхронизацией. Программист должен специфицировать действия всех процессов, а также их взаимодействие и синхронизацию. Это контрастирует с *декларативными программами*, например, функциональными или логическими, где параллелизм скрыт и отсутствуют чтение и запись состояния программы. В декларативных программах независимые части программы могут выполняться параллельно; их взаимодействие и синхронизация происходят неявно, когда одна часть программы зависит от результата выполнения другой. Декларативный подход тоже интересен и важен (см. главу 12), но императивный распространен гораздо шире. Кроме того, для реализации декларативной программы на стандартной машине необходимо писать императивную программу.

Изучаются также параллельные программы, в которых процессы выполняются *асинхронно*, т.е. каждый со своей скоростью. Такие программы могут выполняться с помощью чередования процессов на одном процессоре или их параллельного выполнения на мультипроцессоре со многими командами и многими данными (MIMD-процессоре). К этому классу машин относятся также мультипроцессоры с разделяемой памятью, многомашинные системы с распределенной памятью и сети рабочих станций (см. следующий раздел). Несмотря на внимание к асинхронным параллельным вычислениям, в главе 3 мы описываем *синхронную* мультиобработку (SIMD-машины), а в главах 3 и 12 — связанный с ней стиль программирования, параллельного по данным.

1.2. Структуры аппаратного обеспечения

В данной главе дается обзор основных атрибутов архитектуры современных компьютеров. В следующем разделе описаны приложения параллельного программирования и использование архитектуры в них. Описание начинается с однопроцессорных систем и кэш-памяти. Затем рассматриваются мультипроцессоры с разделяемой памятью. В конце описываются машины с распределенной памятью, к которым относятся многомашинные системы с распределенной памятью и сети машин.

1.2.1. Процессоры и кэш-память

Современная однопроцессорная машина состоит из нескольких компонентов: центрального процессорного устройства (ЦПУ), первичной памяти, одного или нескольких уровней кэш-памяти (кэш), вторичной (дисковой) памяти и набора периферийных устройств (дисплей, клавиатура, мышь, модем, CD, принтер и т.д.). Основными компонентами для выполнения программ являются ЦПУ, кэш и память. Отношения между ними изображены на рис. 1.1.

Процессор выбирает инструкции из памяти, декодирует их и выполняет. Он содержит управляющее устройство (УУ), арифметико-логическое устройство (АЛУ) и регистры. УУ вырабатывает сигналы, управляющие действиями АЛУ, системой памяти и внешними устройствами. АЛУ выполняет арифметические и логические инструкции, определяемые набором инструкций процессора. В регистрах хранятся инструкции, данные и состояние машины (включая счетчик команд).

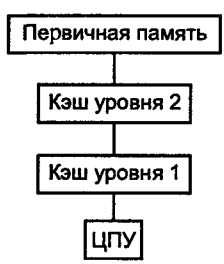


Рис. 1.1. Процессор, кэш и память в современной вычислительной машине

Кэш — это небольшой по объему, но скоростной модуль памяти, используемый для ускорения выполнения программы. В нем хранится содержимое областей памяти, часто используемых процессором. Причина использования кэш-памяти состоит в том, что в большинстве программ наблюдается *временная локальность*, означающая, что если произошло обращение к некоторой области памяти, то, скорее всего, в ближайшем будущем обращения к этой области повторятся. Например, инструкции внутри циклов выбираются и выполняются многократно.

Когда программа обращается к адресу в памяти, процессор сначала ищет его в кэше. Если данные находятся там (происходит *попадание* в кэш), то они считываются из кэша. Если данных в кэше нет (*промах*), то данные считываются из первичной памяти и в процессор, и в кэш-память. Аналогично, когда программа записывает данные, они помещаются в первичную память и, возможно, в локальный кэш. В *сквозном кэше* данные помещаются в память немедленно, в *кэше с обратной записью* — позже. Ключевой момент состоит в том, что после записи содержимое первичной памяти временно может не соответствовать содержимому кэша.

Чтобы ускорить передачу (увеличить пропускную способность) между кэшем и первичной памятью, в элемент кэш-памяти обычно включают непрерывную последовательность слов из памяти. Эти элементы называются *блоками* или *строками* кэша. При промахе из памяти в кэш

передается полная строка. Это эффективно, поскольку в большинстве программ наблюдается *пространственная локальность*, т.е. за обращением к одному слову памяти вскоре последуют обращения к другим близлежащим словам.

В современных процессорах обычно есть два типа кэша. Кэш уровня 1 находится ближе к процессору, а кэш уровня 2 — между кэшем уровня 1 и первичной памятью. Кэш уровня 1 меньше и быстрее, чем кэш уровня 2, и зачастую иначе организован. Например, кэш-память уровня 1 обычно *отображается непосредственно*, а кэш уровня 2 является *множественно-ассоциативным*.[2] Кроме того, кэш уровня 1 часто содержит отдельные области кэш-памяти для инструкций и данных, в то время как кэш уровня 2 обычно *унифицирован*, т.е. в нем хранятся и данные, и инструкции.

Проиллюстрируем различия в скорости работы уровней иерархии памяти. Доступ к регистрам происходит за один такт работы процессора, поскольку они невелики и находятся внутри ЦПУ. Данные кэш-памяти уровня 1 также доступны за один-два такта. Однако для доступа к кэш-памяти уровня 2 необходимо порядка 10 тактов, а к первичной памяти — от 50 до 100 тактов. Аналогичны и различия в размере типов памяти: ЦПУ содержит несколько десятков регистров, кэш уровня 1 — несколько десятков килобайт, кэш уровня 2 — порядка мегабайта, а первичная память — десятки и сотни мегабайт.

1.2.2. Мультипроцессоры с разделяемой памятью

В *мультипроцессоре с разделяемой памятью* процессоры и модули памяти связаны с помощью *соединительной сети* (рис. 1.2). Процессоры совместно используют первичную память, но каждый из них имеет собственный кэш.

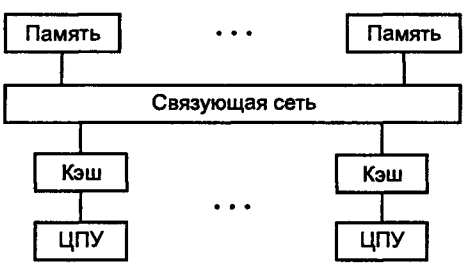


Рис. 1.2. Структура мультипроцессоров с разделяемой памятью

В небольшом мультипроцессоре, имеющем от двух до (порядка) 30 процессоров, соединительная сеть реализована в виде шины памяти или, возможно, матричного коммутатора. Такой мультипроцессор называется *однородным* (UMA machine — от "uniform memory access"), поскольку время доступа каждого из процессоров к любому участку памяти одинаково. Однородные машины также называются *симметричными мультипроцессорами*.

В больших мультипроцессорах с разделяемой памятью, включающих десятки или сотни процессоров, память организована иерархически. Соединительная сеть имеет вид древообразного набора переключателей и областей памяти. Следовательно, одна часть памяти ближе к определенному процессору, другая — дальше от него. Такая организация памяти

[2] В непосредственно отображаемом кэше каждый адрес памяти отображается в один элемент кэша, а в множественно-ассоциативном — во множество элементов (обычно два или четыре). Таким образом, если два адреса памяти отображаются в одну и ту же ячейку, в непосредственно отображаемом кэше может находиться только ячейка, к которой производилось самое последнее обращение, а в ассоциативном — обе ячейки. С другой стороны, непосредственно отображаемый кэш быстрее, поскольку проще выяснить, есть ли данное слово в кэше.

позволяет избежать перегрузки, возникающей при использовании одной шины или коммутатора, и приводит к неравным временам доступа, поэтому такие мультипроцессоры называются *неоднородными* (NUMA machines).

В машинах обоих типов у каждого процессора есть собственный кэш. Если два процессора ссылаются на разные области памяти, их содержимое можно безопасно поместить в кэш каждого из них. Проблема возникает, когда два процессора обращаются к одной области памяти примерно одновременно. Если оба процессора только считывают данные, в кэш каждого из них можно поместить копию данных. Но если один из процессоров записывает в память, возникает *проблема согласованности кэша*: в кэш-памяти другого процессора теперь содержатся неверные данные. Значит, необходимо либо обновить кэш другого процессора, либо признать содержимое кэша недействительным. В каждом мультипроцессоре протокол согласования кэш-памяти должен быть реализован аппаратно. Один из способов состоит в том, чтобы каждый кэш "следил" за адресной шиной, отлавливая ссылки на области памяти, находящиеся в нем.

Запись в память также приводит к проблеме *согласованности памяти*: когда в действительности обновляется первичная память? Например, если один процессор выполняет запись в область памяти, а другой позже считывает данные из этой области, будет ли считано обновленное значение? Существует несколько различных моделей согласованности памяти. *Последовательная согласованность* — это наиболее сильная модель. Она гарантирует, что обновления памяти будут происходить в некоторой последовательности, причем каждому процессору будет "видна" одна и та же последовательность. *Согласованность процессоров* — более слабая модель. Она обеспечивает, что записи в память, выполняемые каждым процессом, совершаются в том порядке, в котором их производит процессор, но записи, инициированные различными процессорами, для других процессоров могут выглядеть по-разному. Еще более слабая модель — *согласование освобождения*, при которой первичная память обновляется в указанных программистом точках синхронизации.

Проблема согласования памяти представляет противоречия между простотой программирования и расходами на реализацию. Программист интуитивно ожидает последовательного согласования, поскольку программа считывает и записывает переменные независимо от того, в какой части машины они хранятся в действительности. Когда процесс присваивает переменной значение, программист ожидает, что результаты этого присваивания станут немедленно известными всем процессам программы. С другой стороны, последовательное согласование очень дорого в реализации и замедляет работу машины. Дело в том, что при каждой записи аппаратная часть должна проверить все кэши (и, возможно, обновить их или сделать недействительными) и модифицировать первичную память. Вдобавок, эти действия должны быть *неделимыми*. Вот почему в мультипроцессорах обычно реализуется более слабая модель согласования памяти, а программистам необходимо вставлять инструкции синхронизации памяти. Это часто обеспечивают компиляторы и библиотеки, так что прикладной программист этим может не заниматься.

Как было отмечено, строки кэш-памяти часто содержат последовательности слов, которые блоками передаются из памяти и обратно. Предположим, что переменные x и y занимают по одному слову и хранятся в соседних ячейках памяти, отображенных в одну и ту же строку кэш-памяти. Пусть некоторый процесс выполняется на процессоре 1 мультипроцессора и производит записи и чтения переменной x. Наконец, допустим, что еще один процесс, выполняемый на процессоре 2, считывает и записывает переменную y. Тогда при каждом обращении процессора 1 к переменной x строка кэш-памяти этого процессора будет содержать и копию переменной y. Аналогичная картина будет и в кэше процессора 2.

Ситуация, описанная выше, называется *ложным разделением*: процессы в действительности не разделяют переменные x и y, но аппаратная часть кэш-памяти интерпретирует обе переменные как один блок. Следовательно, когда процессор 1 обновляет переменную x, должна быть признана недействительной и обновиться строка кэша в процессоре 2, содержащая и x, и y. Таким же образом, когда процессор 2 обновляет значение y, строка кэш-памяти, содержащая значения x и y, в процессоре 1 тоже должна быть обновлена или признана недействительной. Эти операции замедляют работу системы памяти, поэтому программа будет выполняться намного

медленнее, чем тогда, когда две переменные попадают в разные строки кэша. Главное здесь — чтобы программист ясно понимал, что процессы не разделяют переменные, когда фактически система памяти вынуждена обеспечивать их разделение и тратить время на управление им.

Чтобы избежать ложного разделения, программист должен гарантировать, что переменные, записываемые различными процессами, не будут находиться в смежных областях памяти. Одно из решений этой проблемы заключается в *выравнивании*, т.е. резервировании фиктивных переменных, которые просто занимают место и отделяют реальные переменные друг от друга. Это пример противоречия между временем и пространством: для сокращения времени выполнения приходится тратить пространство.

Итак, мультипроцессоры используют кэш-память для повышения производительности. Однако иерархичность памяти порождает проблемы согласованности кэша и памяти, а также возможность ложного разделения. Следовательно, для получения максимальной производительности на данной машине программист должен знать характеристики системы памяти и писать программы, учитывая их. К этим вопросам мы еще вернемся.

1.2.3. Мультикомпьютеры с распределенной памятью и сети

В *мультипроцессоре с распределенной памятью* тоже есть соединительная сеть, но каждый процессор имеет собственную память. Как показано на рис. 1.3, соединительная сеть и модули памяти меняются местами по сравнению с их положением в мультипроцессоре с разделяемой памятью. Соединительная сеть поддерживает *передачу сообщений*, а не чтение и запись в память. Следовательно, процессоры взаимодействуют, передавая и принимая сообщения. У каждого процессора есть свой кэш, но из-за отсутствия разделения памяти нет проблем согласованности кэша и памяти.

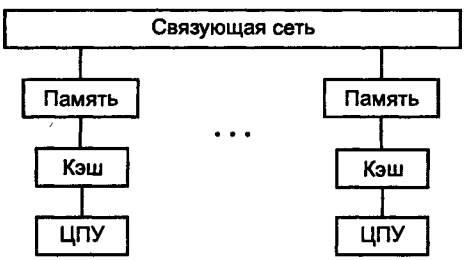


Рис. 1.3. Структура машин с распределенной памятью

Мультикомпьютер (*многомашинная система*) — это мультипроцессор с распределенной памятью, в котором процессоры и сеть расположены физически близко (в одном помещении). По этой причине такую многомашинную систему часто называют *тесно связанной* (*спаренной*) *машиной*. Она одновременно используется одним или небольшим количеством приложений; каждое приложение задействует выделенный набор процессоров. Соединительная сеть с большой пропускной способностью предоставляет высокоскоростной путь связи между процессорами. Обычно она организована в гиперкуб или матричную структуру. (Машины со структурой типа гиперкуб были одними из первых многомашинных систем.)

Сетевая система — это многомашинная система с распределенной памятью, узлы которой связаны с помощью локальной сети типа Ethernet или такой глобальной сети, как Internet. Сетевые системы называются *слабо связанными мультикомпьютерами*. Здесь процессоры взаимодействуют также с помощью передачи сообщений, но время их доставки больше, чем в многомашинных системах, и в сети больше конфликтов. С другой стороны, сетевая система строится на основе обычных рабочих станций и сетей, тогда как в многомашинной системе часто есть специализированные компоненты, особенно у связующей сети.

Сетевая система, состоящая из набора рабочих станций, часто называется *сетью рабочих станций* (network of workstations — NOW) или *кластером рабочих станций* (cluster of workstations — COW). Все рабочие станции выполняют одно или несколько приложений, возможно, связанных между собой. Популярный сейчас способ построения недорогого мультипроцессора с распределенной памятью — собрать так называемую *машину Беовулфа* (Beowulf). Она состоит из базового аппаратного обеспечения и таких бесплатных программ, как чипы к процессорам Pentium, сетевые карты, диски и операционная система Linux. (Имя Беовулф взято из старинной английской поэмы, первого шедевра английской литературы.)

Существуют также гибридные сочетания мультипроцессоров с распределенной и разделяемой памятью. Узлами системы с распределенной памятью могут быть мультипроцессоры с разделяемой памятью, а не обычные процессоры. Возможен вариант, когда связующая сеть поддерживает и механизмы передачи сообщений, и механизмы прямого доступа к удаленной памяти (на сегодня это наиболее мощные машины). Наиболее общая комбинация — машина с поддержкой *распределенной разделяемой памяти*, т.е. распределенной реализации абстракции разделяемой памяти. Она облегчает программирование многих приложений, но ставит проблемы согласованности кэша и памяти. (В главе 10 описана распределенная разделяемая память и ее реализация в программном обеспечении.)

1.3. Приложения и стили программирования

Параллельное программирование обеспечивает способ организации программного обеспечения, состоящего из относительно независимых частей. Оно также позволяет использовать множественные процессоры. Существует три обширных перекрывающихся класса приложений — многопоточные системы, распределенные системы и синхронные параллельные вычисления — и три соответствующих им типа параллельных программ.

Напомним, что процесс — это последовательная программа, которая при выполнении имеет собственный поток управления. Каждая параллельная программа содержит несколько процессов, поэтому имеет несколько потоков. Однако термин *многопоточный* обычно означает, что программа содержит больше процессов (потоков), чем существует процессоров для их выполнения. Следовательно, процессы на процессорах выполняются по очереди. Многопоточная программная система управляет множеством независимых процессов, например таких:

- оконные системы на персональных компьютерах или рабочих станциях;
- многопроцессорные операционные системы и системы с разделением времени;
- системы реального времени, управляющие электростанциями, космическими аппаратами и т.д.

Эти системы разработаны как многопоточные программы, поскольку организовать код и структуры данных в виде набора процессов намного проще, чем реализовать огромную последовательную программу. Кроме того, каждый процесс может планироваться и выполняться независимо. Например, когда пользователь нажимает кнопку мыши персонального компьютера, посылается сигнал процессу, управляющему окном, в котором в данный момент находится курсор мыши. Этот процесс (поток) может выполняться и отвечать на щелчок мыши. Приложения в других окнах могут продолжать при этом свое выполнение в фоновом режиме.

Второй широкий класс приложений образуют *распределенные вычисления*, в которых компоненты выполняются на машинах, связанных локальной или глобальной сетью. По этой причине процессы взаимодействуют, обмениваясь сообщениями. Вот примеры:

- файловые серверы в сети;
- системы баз данных для банков, заказа авиабилетов и т.д.;
- Web-серверы сети Internet;

- предпринимательские системы, объединяющие компоненты производства;
- отказоустойчивые системы, которые продолжают работать независимо от сбоев в компонентах.

Такие системы пишутся для распределения обработки (как в файловых серверах), обеспечения доступа к удаленным данным (как в базах данных и в Web), интеграции и управления данными, распределенными по своей сути (как в промышленных системах), или повышения надежности (как в отказоустойчивых системах). Многие распределенные системы организованы как системы типа клиент-сервер. Например, файловый сервер предоставляет файлы данных для процессов, выполняемых на клиентских машинах. Компоненты распределенных систем часто сами являются многопоточными программами.

Синхронные параллельные вычисления — третий широкий класс приложений. Их цель — быстро решить данную задачу или за то же время решить большую задачу. Примеры синхронных вычислений:

- научные вычисления, которые моделируют и имитируют такие явления, как глобальный климат, эволюция солнечной системы или результат действия нового лекарства;
- графика и обработка изображений, включая создание спецэффектов в кино;
- крупные комбинаторные или оптимизационные задачи, например, планирование авиаперелетов или экономическое моделирование.

Программы решения таких задач требуют больших вычислительных мощностей. Для достижения высокой производительности они выполняются на параллельных процессорах, причем обычно количество процессов (потоков) равно числу процессоров. Параллельные вычисления описываются в виде *программ, параллельных по данным*, в которых все процессы выполняют одни и те же действия, но с собственной частью данных, или в виде *программ, параллельных по задачам*, в которых различные процессы решают различные задачи.

В данной книге рассмотрены все три вида приложений и, что более важно, показано, как их программировать. В многопоточных программах процессы (потоки) взаимодействуют, используя разделяемые переменные. В распределенных системах взаимодействие процессов обеспечивается с помощью обмена сообщениями или удаленного вызова операций. При выполнении синхронных параллельных вычислений процессы взаимодействуют, используя или разделяемые переменные, или передачу сообщений, в зависимости от аппаратного обеспечения, на котором выполняется программа. В части 1 этой книги показано, как написать программу, использующую для взаимодействия и синхронизации разделяемые переменные. Часть 2 описывает передачу сообщений и удаленные операции. В части 3 подробно рассмотрено синхронное параллельное программирование, ориентированное на научные вычисления.

Существует немало параллельных программных приложений, однако в них используется лишь небольшое число моделей решений, или *парадигм*. В частности, существует пять основных парадигм: 1) итеративный параллелизм, 2) рекурсивный параллелизм, 3) "производители и потребители" (конвейеры), 4) "клиенты и серверы", 5) взаимодействующие равные. С использованием одной или нескольких из этих парадигм и программируются приложения.

Итеративный параллелизм используется, когда в программе есть несколько процессов (часто идентичных), каждый из которых содержит один или несколько циклов. Таким образом, каждый процесс является итеративной программой. Процессы программы работают совместно над решением одной задачи; они взаимодействуют и синхронизируются с помощью разделяемых переменных или передачи сообщений. Итеративный параллелизм чаще всего встречается в научных вычислениях, выполняемых на нескольких процессорах.

Рекурсивный параллелизм может использоваться, когда в программе есть одна или несколько рекурсивных процедур, и их вызовы независимы, т.е. каждый из них работает над своей частью общих данных. Рекурсия часто применяется в императивных языках программирования, особенно при реализации алгоритмов типа "разделяй и властвуй" или "перебор с воз-

вращением" (backtracking). Рекурсия является одной из фундаментальных парадигм и в символических, логических, функциональных языках программирования. Рекурсивный параллелизм используется для решения таких комбинаторных проблем, как сортировка, планирование (задача коммивояжера) и игры (шахматы и другие).

Производители и потребители — это взаимодействующие процессы. Они часто организуются в *конвейер*, через который проходит информация. Каждый процесс конвейера является *фильтром*, который потребляет выход своего предшественника и производит входные данные для своего последователя. Фильтры встречаются на уровне приложений (оболочки) в операционных системах типа ОС Unix, внутри самих операционных систем, внутри прикладных программ, если один процесс производит выходные данные, которые потребляет (читает) другой процесс.

Клиенты и серверы — наиболее распространенная модель взаимодействия в распределенных системах, от локальных сетей до World Wide Web. Клиентский процесс запрашивает сервис и ждет ответа. Сервер ожидает запросов от клиентов, а затем действует в соответствии с этими запросами. Сервер может быть реализован как одиночный процесс, который не может обрабатывать одновременно несколько клиентских запросов, или (при необходимости параллельного обслуживания запросов) как многопоточная программа. Клиенты и серверы представляют собой параллельное программное обобщение процедур и их вызовов: сервер выполняет роль процедуры, а клиенты ее вызывают. Но если код клиента и код сервера расположены на разных машинах, обычный вызов процедуры использовать нельзя. Вместо этого необходимо использовать *удаленный вызов процедуры* или *рандеву*, как это описано в главе 8.

Взаимодействующие равные — последняя парадигма взаимодействия. Она встречается в распределенных программах, в которых несколько процессов для решения задачи выполняют один и тот же код и обмениваются сообщениями. Взаимодействующие равные используются для реализации распределенных параллельных программ, особенно при итеративном параллелизме и децентрализованном принятии решений в распределенных системах. Некоторые приложения и схемы взаимодействия описаны в главе 9.

В следующих пяти разделах приводятся примеры, иллюстрирующие применение каждой модели. В примерах представлена программная нотация, используемая во всей книге. (Обзор нотации дается в разделе 1.9.) Еще больше примеров приведено в тексте и упражнениях последующих глав.

1.4. Итеративный параллелизм: умножение матриц

Итеративная последовательная программа использует для обработки данных и вычисления результатов циклы типа `for` и `while`. Итеративная параллельная программа содержит несколько итеративных процессов. Каждый процесс вычисляет результаты для подмножества данных, а затем эти результаты собираются вместе.

В качестве простого примера рассмотрим задачу из области научных вычислений. Предположим, даны матрицы `a` и `b`, у каждой по `n` строк и столбцов, и обе инициализированы. Цель — вычислить произведение матриц, поместив результат в матрицу `c` размером `n×n`. Для этого нужно вычислить `n2` промежуточных произведений, по одному для каждой пары строк и столбцов.

Матрицы являются разделяемыми переменными, объявленными следующим образом.

```
double a[n,n], b[n,n], c[n,n];
```

При условии, что `n` уже объявлено и инициализировано, это объявление резервирует память для трех массивов действительных чисел двойной точности. По умолчанию индексы строк и столбцов изменяются от `0` до `n-1`.

После инициализации массивов `a` и `b` можно вычислить произведение матриц по такой последовательной программе.

```
for [i = 0 to n-1] {
  for [j = 0 to n-1] {
```

```
    # вычислить произведение a[i,*] и b[*,j]
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Внешние циклы (с индексами i и j) повторяются для каждой строки и столбца. Во внутреннем цикле (с индексом k) вычисляется промежуточное произведение строки i матрицы a и столбца j матрицы b; результат сохраняется в ячейке c[i,j]. Строка с символом # в начале является комментарием.

Умножение матриц — это пример *приложения с массовым параллелизмом*, поскольку программа содержит большое число операций, которые могут выполняться параллельно. Две операции могут выполняться параллельно, если они *независимы*. Предположим, что *множество чтения* операции содержит переменные, которые она читает, но не изменяет, а *множество записи* — переменные, которые она изменяет (и, возможно, читает). Две операции являются независимыми, если их множества записи не пересекаются. Говоря неформально, процессы всегда могут безопасно читать переменные, которые не изменяются. Однако двум процессам в общем случае небезопасно выполнять запись в одну и ту же переменную или одному процессу читать переменную, которая записывается другим. (Эта тема рассматривается подробно в главе 2.)

При умножении матриц вычисления промежуточных произведений являются независимыми операциями. В частности, строки с 4 по 6 приведенной выше программы выполняют инициализацию и последующее вычисление элемента матрицы c. Внутренний цикл программы читает строку матрицы a и столбец матрицы b, а затем читает и записывает один элемент матрицы c. Следовательно, множество чтения для внутреннего произведения — это строка матрицы a и столбец матрицы b, а множество записи — элемент матрицы c.

Поскольку множества записи внутренних произведений не пересекаются, их можно выполнять параллельно. Возможны варианты, когда параллельно вычисляются результирующие строки, результирующие столбцы или группы строк и столбцов. Ниже будет показано, как запрограммировать такие параллельные вычисления.

Сначала рассмотрим параллельное вычисление строк матрицы c. Его можно запрограммировать с помощью оператора co (от "concurrent" — "параллельный"):

```
co [i = 0 to n-1] { # параллельное вычисление строк
  for [j = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Между этой программой и ее последовательным вариантом есть лишь *одно* синтаксическое различие — во внешнем цикле оператор for заменен оператором co. Но семантическая разница велика: оператор co определяет, что его тело для каждого значения индекса i будет выполняться *параллельно* (если не в действительности, то, по крайней мере, теоретически, что зависит от числа процессоров).

Другой способ выполнения параллельного умножения матриц состоит в параллельном вычислении столбцов матрицы c. Его можно запрограммировать следующим образом.

```
co [j = 0 to n-1] { #параллельное вычисление столбцов
  for [i = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Здесь два внешних цикла (по i и по j) *поменялись местами*. Если тела двух циклов независимы и приводят к вычислению одинаковых результатов, их можно безопасно менять местами, как это было сделано здесь. (Это и другие аналогичные преобразования программ рассматриваются в главе 12.)

Программу для параллельного вычисления всех промежуточных произведений можно составить несколькими способами. Используем один оператор co для двух индексов.

```
co [i = 0 to n-1, j = 0 to n-1] { # все строки и
  c[i,j] = 0.0;                   # все столбцы
  for [k = 0 to n-1]
    c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Тело оператора co выполняется параллельно для каждой комбинации значений индексов i и j, поэтому программа задает n2 процессов. (Будут ли они выполняться параллельно, зависит от конкретной реализации.) Другой способ параллельного вычисления промежуточных произведений состоит в использовании вложенных операторов co.

```
co [i = 0 to n-1] {   # строки параллельно, затем
  co [j = 0 to n-1] { # столбцы параллельно
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Здесь для каждой строки (внешний оператор co) и затем для каждого столбца (внутренний оператор co) задается по одному процессу. Третий способ написать эту программу — поменять местами две строки последней программы. Результат всех трех программ будет одинаковым: выполнение внутреннего цикла для всех n2 комбинаций значений i и j. Разница между ними — в задании процессов, а значит, и во времени их создания.

Заметим, что все параллельные программы, приведенные выше, были получены заменой оператора for на co. Но мы сделали это только для индексов i и j. А как быть с внутренним циклом по индексу k? Нельзя ли и этот оператор заменить оператором co? Ответ — "нет", поскольку тело внутреннего цикла как читает, так и записывает переменную c[i,j]. Промежуточное произведение — цикл for с переменной k — можно вычислить, используя двоичный параллелизм, но для большинства машин это непрактично (см. упражнения в конце главы).

Другой способ определить параллельные вычисления, показанные выше, — использовать декларацию (объявление) process вместо оператора co. В сущности, process — это оператор co, выполняемый как "фоновый". Например, первая параллельная программа из показанных выше — та, что параллельно вычисляет строки результата, — может быть записана следующим образом.

```
process row[i = 0 to n-1] { # строки параллельно
  for [j = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Здесь определен массив процессов — row[1], row[2] и т.д. — по одному для каждого значения индекса i. Эти n процессов создаются и начинают выполняться, когда встречается данная строка описания. Если за декларацией процесса следуют операторы, то они выполняются параллельно с процессом, тогда как операторы, записанные после оператора co, не выполняются до его завершения. Декларации процесса, в отличие от операторов co, не могут быть вложены в другие декларации или операторы. Декларации процессов и операторы co подробно описаны в разделе 1.9.

В программах, приведенных выше, для каждого элемента, строки или столбца результирующей матрицы использовано по одному процессу. Предположим, что число процессоров в системе меньше n (так обычно и бывает, особенно когда n велико). Остается еще очевидный способ полного использования всех процессоров: разделить матрицу на полосы (строк или столбцов) и для каждой полосы создать *рабочий* процесс. В частности, каждый рабочий процесс вычисляет результаты для элементов своей полосы. Предположим, что есть P процессоров и n кратно P (т.е. n делится на P без остатка). Тогда при использовании полос строк рабочие процессы можно запрограммировать следующим образом.

```
process worker[w = 1 to P] {   # полосы параллельно
  int first = (w-1) * n/P;     # первая строка полосы
  int last = first + n/P - 1; # последняя строка полосы
  for [i = first to last] {
    for [j = 0 to n-1] {
      c[i,j] = 0.0;
      for [k = 0 to n-1]
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
  }
}
```

Отличие этой программы от предыдущей состоит в том, что n строк делятся на P полос, по n/P строк каждая. Для этого в программу добавлены операторы, необходимые для определения первой и последней строки каждой полосы. Затем строки полосы указываются в цикле (по индексу i), чтобы вычислить промежуточные произведения для этих строк.

Итак, существенным условием распараллеливания программы является наличие независимых вычислений, т.е. вычислений с непересекающимися множествами записи. Для произведения матриц независимыми вычислениями являются промежуточные произведения, поскольку каждое из них записывает (и читает) свой элемент c[i,j] результирующей матрицы. Поэтому можно параллельно вычислять все промежуточные произведения, строки, столбцы или полосы строк. И, наконец, параллельные программы можно записывать, используя операторы co или обьявления process.

1.5. Рекурсивный параллелизм: адаптивная квадратура

Программа считается рекурсивной, если она содержит процедуры, которые вызывают сами себя — прямо или косвенно. Рекурсия дуальна итерации в том смысле, что рекурсивные программы можно преобразовать в итеративные и наоборот. Однако каждый стиль программирования имеет свое применение, поскольку одни задачи по своей природе рекурсивны, а другие — итеративны.

В теле многих рекурсивных процедур обращение к самой себе встречается больше одного раза. Например, алгоритм *quicksort* часто используется для сортировки. Он разбивает массив на две части, а затем дважды вызывает себя: первый раз для сортировки левой части, а второй — для правой. Многие алгоритмы для работы с деревьями и графами имеют подобную структуру.

Рекурсивную программу можно реализовать с помощью параллелизма, если она содержит несколько независимых рекурсивных вызовов. Два вызова процедуры (или функции) являются независимыми, если их множества записи не пересекаются. Это условие выполняется, если: 1) процедура не обращается к глобальным переменным или только читает их; 2) аргументы и результирующие переменные (если есть) являются различными переменными. Например, если процедура не обращается к глобальным переменным и имеет только параметры-значения, то любой ее вызов будет независимым. (Хорошо, если процедура читает и записывает только локальные переменные, тогда каждый экземпляр процедур имеет локальную копию переменных.) В соответствии с этими требованиями можно запрограммировать и алгоритм быстрой сортировки. Рассмотрим еще один интересный пример.

Задача квадратуры состоит в аппроксимации интеграла непрерывной функции. Предположим, что это функция `f(x)`. Как показано на рис. 1.4, интеграл функции `f(x)` от `a` до `b` — это площадь между графиком `f(x)` и осью абсцисс от прямой `x = a` до прямой `x = b`.

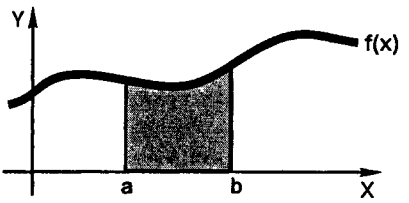


Рис. 1.4. Задача квадратуры

Существует два основных способа аппроксимации значения интеграла. Первый — разделить интервал от `a` до `b` на фиксированное число отрезков, а затем аппроксимировать площадь на каждом из них по правилу трапеций или по правилу Симпсона.

```
double fleft = f(a), fright, area = 0.0;
double width = (b-a) / INTERVALS;
for [x = (a + width) to b by width] {
  fright = f(x);
  area = area + (fleft + fright) * width / 2;
  fleft = fright;
}
```

Каждая итерация вычисляет площадь малой фигуры по правилу трапеций и добавляет ее к общему значению площади. Переменная `width` — ширина каждой трапеции. Отрезки перебираются слева направо, поэтому правое значение каждой итерации становится левым значением следующей итерации.

Второй способ аппроксимации интеграла — использовать парадигму "разделяй и властвуй" и переменное число интервалов. В частности, сначала вычисляют значение `m` — середину отрезка между `a` и `b`. Затем аппроксимируют площадь трех областей под кривой, определенной функцией `f()`: от `a` до `m`, от `m` до `b` и от `a` до `b`. Если сумма меньших площадей равна большей площади с некоторой заданной точностью `EPSILON`, то аппроксимацию можно считать достаточной. Если нет, то большая задача — от `a` до `b` — делится на две подзадачи — от `a` до `m` и от `m` до `b`, и процесс повторяется. Этот способ называется *адаптивной квадратурой*, поскольку алгоритм адаптируется к форме кривой. Его можно запрограммировать так.

```
double quad(double left,right,fleft,fright,lrarea) {
  double mid = (left + right) / 2;
  double fmid = f(mid);
  double larea = (fleft+fmid) * (mid-left) / 2;
  double rarea = (fmid+fright) * (right-mid) / 2;
  if (abs((larea+rarea) - lrarea) > EPSILON) {
    # рекурсия для интегрирования обоих значений
    larea = quad(left, mid, fleft, fmid, larea);
    rarea = quad(mid, right, fmid, fright, rarea);
  }
  return (larea + rarea);
}
```

Интеграл функции `f(x)` от `a` до `b` аппроксимируется таким вызовом функции:

```
area = quad(a, b, f(a), f(b), (f(a)+f(b))*(b-a)/2);
```

В функции снова используется правило трапеции. Значения функции `f()` в крайних точках отрезка и приближенная площадь этого интервала передаются в каждый вызов функции `quad`, чтобы не вычислять их больше одного раза.

Итеративную программу нельзя распараллелить, поскольку тело цикла и считывает, и записывает значение переменной `area`. Тем не менее в рекурсивной программе вызовы функции `quad` независимы при условии, что вычисление функции `f(x)` не дает побочных эффектов. В частности, аргументы функции `quad` передаются по значению, и в ее теле нет присваи-

вания глобальным переменным. Таким образом, для задания параллельного выполнения рекурсивных вызовов функции можно использовать оператор co.

```
co larea = quad(left, mid, fleft, fmid, larea);
// rarea = quad(mid, right, fmid, fright, rarea);
oc
```

Это единственное изменение, необходимое для того, чтобы сделать данную программу рекурсивной. Поскольку оператор co не заканчивается до тех пор, пока не будут завершены оба вызова функций, значения переменных larea и rarea вычисляются до того, как функция quad возвратит их сумму.

В операторах co программ умножения матриц содержатся списки инструкций, выполняемых для каждого значения счетчиков (i и j). В операторе co, приведенном выше, содержатся два вызова функций, разделенных знаками //. Первая форма оператора co используется для выражения итеративного параллелизма, вторая — рекурсивного.

Итак, программу с несколькими рекурсивными вызовами функций можно легко преобразовать в параллельную рекурсивную программу, если вызовы независимы. Однако существует чисто практическая проблема: параллельно выполняемых операций может стать слишком много. Каждый оператор co в приведенной выше программе создает два процесса, по одному для каждого вызова функции. Если глубина рекурсии велика, то появится большое число процессов, возможно, слишком большое для параллельного выполнения. Решение этой проблемы состоит в *сокращении*, или *отсечении*, дерева рекурсии при достаточном количестве процессов, т.е. переключении с параллельных рекурсивных вызовов на последовательные. Эта тема рассматривается в упражнениях и далее в этой книге.

1.6. Производители и потребители: каналы ОС Unix

Процесс-производитель выполняет вычисления и выводит поток результатов. Процесс-потребитель вводит и анализирует поток значений. Многие программы в той или иной форме являются производителями и/или потребителями. Сочетание становится особенно интересным, если производители и потребители объединены в *конвейер* — последовательность процессов, в которой каждый из них потребляет данные выхода предшественника и производит данные для последующего процесса. Классическим примером являются конвейеры в операционной системе Unix, рассматриваемые здесь. Другие примеры приводятся в последующих главах.

Обычно прикладной процесс в ОС Unix считывает данные из *стандартного файла ввода* stdin и записывает в *стандартный файл вывода* stdout. Обычно файл ввода — это клавиатура терминала, с которого вызвано приложение, а файл вывода — дисплей этого терминала. Но одной из наиболее мощных функций, предложенных в ОС Unix, была возможность привязки стандартных "устройств" ввода-вывода к различным типам файлов. В частности, файлы stdin и/или stdout могут быть связаны с файлом данных или с "файлом" особого типа, который называется каналом.

Канал — это буфер (очередь типа FIFO, работающая по принципу "First in — first out", т.е. "первым вошел, первым вышел") между процессом-производителем и процессом-потребителем. Он содержит связанную последовательность символов. Новые значения дописываются к ней, когда производитель выполняет запись в канал. Символы удаляются, когда процесс-потребитель считывает их из канала.

Прикладная программа в ОС Unix только читает данные из файла stdin, не заботясь о том, откуда в действительности они туда попали. Если файл stdin связан с клавиатурой, на вход поступают символы, набранные на клавиатуре. Если файл stdin связан с определенным файлом, вводится последовательность символов из этого файла. Если файл stdin связан с каналом, то вводится последовательность символов, записанных в этот канал. Аналогично приложение выполняет запись в файл stdout, не заботясь о том, куда в действительности поступают данные.

Каналы ОС Unix обычно определяются с помощью одного из командных языков, например csh (C shell — "оболочка C"). В частности, печатные страницы оригинала этой книги создавались с помощью команды на языке csh, похожей на следующую:

```
sed -f Script $* | tbl |eqn | groff Macros -
```

Этот конвейер содержит четыре команды: 1) sed, потоковый текстовый редактор; 2) tbl, процессор таблиц; 3) eqn, процессор уравнений и 4) groff, программа, создающая данные в формате Postscript из исходных файлов в формате troff. Каждая пара команд разделена вертикальной чертой, обозначающей канал в C shell.

На рис. 1.5 показана структура этого конвейера. Каждая команда является процессом-*фильтром*. Вход фильтра sed образован файлом редактирующих команд (Script) и аргументами командной строки ($*), которыми в данном случае являются соответствующие исходные файлы текста книги. Выход редактора sed передается программе tbl, направляющей свои выходные данные программе eqn, а та передает свой выход программе groff. Фильтр groff читает файл Macros для этой книги, считывает и обрабатывает свой стандартный вход, а затем отсылает выход на принтер в офисе автора.

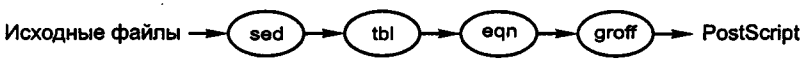


Рис. 1.5. Конвейер процессов

Каждый поток на рис. 1.5 реализован связанным буфером: синхронизированной очередью значений типа FIFO. Процесс-производитель ожидает (при необходимости), пока в буфере появится свободное место, затем добавляет в конец буфера новую строку. Процесс-потребитель ожидает (при необходимости), пока в буфере не появится строка данных, затем забирает ее. В части 1 показано, как реализовать такие буферы с использованием разделяемых переменных и различных примитивов синхронизации (флагов, семафоров и мониторов). В части 2 представлены каналы взаимодействия и примитивы пересылки сообщений send (отослать) и receive (получить). Затем будет показано, как с их использованием программируются фильтры, а с помощью буферов реализуются каналы и передача сообщений.

1.7. Клиенты и серверы: файловые системы

Между производителем и потребителем существует однонаправленный поток информации. Этот вид межпроцессного взаимодействия часто встречается в параллельных программах и не имеет аналогов в последовательных, поскольку в последовательной программе только один поток управления, тогда как производители и потребители — независимые процессы с собственными потоками управления и собственными скоростями выполнения.

Еще одной типичной схемой в параллельных программах является взаимосвязь типа клиент-сервер. Процесс-*клиент* запрашивает сервис, затем ожидает обработки запроса. Процесс-*сервер* многократно ожидает запрос, обрабатывает его, затем посылает ответ. Как показано на рис. 1.6, существует двунаправленный поток информации: от клиента к серверу и обратно. Отношения между клиентом и сервером в параллельном программировании аналогичны отношениям между программой, вызывающей подпрограмму, и самой подпрограммой в последовательном программировании. Более того, как подпрограмма может быть вызвана из нескольких мест программы, так и у сервера обычно есть много клиентов. Запросы каждого клиента должны обрабатываться независимо, однако параллельно может обрабатываться несколько запросов, подобно тому, как одновременно могут быть активны несколько вызовов одной и той же процедуры.

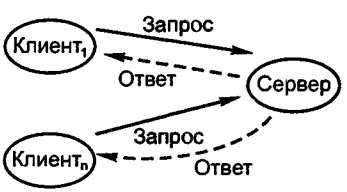


Рис. 1.6. Клиенты и серверы

Взаимодействие типа клиент-сервер встречается в операционных системах, объектно-ориентированных системах, сетях, базах данных и многих других программах. Типичный пример — чтение и запись файла. Для определенности предположим, что есть модуль файлового сервера, обеспечивающий две операции с файлом: `read` (читать) и `write` (писать). Когда процесс-клиент хочет получить доступ к файлу, он вызывает операцию чтения или записи в соответствующем модуле файлового сервера.

На однопроцессорной машине или в другой системе с разделяемой памятью файловый сервер обычно реализуется набором подпрограмм (для операций `read`, `write` и т.д.) и структурами данных, представляющими файлы (например, дескрипторами файлов). Следовательно, взаимодействие между процессом-клиентом и файлом обычно реализуется вызовом соответствующей процедуры. Однако, если файл разделяемый, важно, чтобы запись в него велась одновременно только одним процессом, а читаться он может одновременно несколькими. Эта разновидность задачи — пример так называемой задачи о "читателях и писателях", классической задачи параллельного программирования, которая ставится и решается в главе 4, а также упоминается в последующих главах.

В распределенной системе клиенты и серверы обычно расположены на различных машинах. Например, рассмотрим запрос по World Wide Web, который возникает, когда пользователь открывает новый адрес URL в окне программы-броузера. Web-броузер является клиентским процессом, выполняемым на машине пользователя. Адрес URL косвенно указывает на другую машину, на которой расположена Web-страница. Сама Web-страница доступна для процесса-сервера, выполняемого на другой машине. Этот процесс-сервер может уже существовать или может быть создан; в любом случае он читает Web-страницу, определяемую адресом URL, и возвращает ее на машину клиента. В действительности при преобразовании адреса URL могут использоваться или создаваться дополнительные процессы на промежуточных машинах по пути следования.

Клиенты и серверы программируются одним из двух способов в зависимости от того, выполняются они на одной или на разных машинах. В обоих случаях клиенты являются процессами. На машине с разделяемой памятью сервер обычно реализуется набором подпрограмм. Для защиты критических секций и определения очередности выполнения эти подпрограммы обычно реализуются с использованием взаимных исключений и условной синхронизации. На сетевых машинах или машинах с распределенной памятью сервер реализуется одним или несколькими процессами, которые обычно выполняются не на клиентских машинах. В обоих случаях сервер часто представляет собой многопоточную программу с одним потоком для каждого клиента.

В частях 1 и 2 представлены многочисленные приложения типа клиент-сервер, включая файловые системы, базы данных, программы резервирования памяти, управления диском, а также две классические задачи — об "обедающих философах" и о "спящем парикмахере". В части 1 показано, как реализовать серверы в виде подпрограмм, используя для синхронизации семафоры или мониторы. В части 2 — как реализовать серверы в виде процессов, взаимодействующих с клиентами с помощью пересылки сообщений, удаленных вызовов процедур или рандеву.

1.8. Взаимодействующие равные: распределенное умножение матриц

Ранее было показано, как реализовать параллельное умножение матриц с помощью процессов, разделяющих переменные. Здесь представлены два способа решения этой задачи с использованием процессов, взаимодействующих с помощью пересылки сообщений. Более сложные алгоритмы представлены в главе 9. Первая программа использует управляющий процесс и массив независимых рабочих процессов. Во второй программе рабочие процессы равны и их взаимодействие обеспечивается круговым конвейером. Рис. 1.7 иллюстрирует структуру схем взаимодействия этих процессов. Как показано в части 2, они часто встречаются в распределенных параллельных вычислениях.

а) взаимодействие управляющий-рабочие б) круговой конвейер

Рис. 1.7. Умножение матриц с использованием передачи сообщений

На машинах с распределенной памятью каждый процессор имеет доступ только к собственной локальной памяти. Это значит, что программа не может использовать глобальные переменные, поэтому любая переменная должна быть локальной для некоторого процесса и может быть доступной только этому процессу или процедуре. Следовательно, для взаимодействия процессы должны использовать передачу сообщений.

Допустим, что нам необходимо получить произведение матриц a и b, а результат поместить в матрицу c. Предположим, что каждая из них имеет размер n×n и существует n процессоров. Можно использовать массив из n рабочих процессов, поместив по одному на каждый процессор и заставив каждый рабочий процесс вычислять одну строку результирующей матрицы c. Программа для рабочих процессов будет выглядеть следующим образом.

```
process worker[i = 0 to n-1] {
  double a[n];   # строка i матрицы a
  double b[n,n]; # вся матрица b
  double c[n];   # строка i матрицы c
  receive начальные значения вектора a и матрицы b;
  for [j = 0 to n-1] {
    c[j] = 0.0;
    for [k = 0 to n-1]
      c[j] = c[j] + a[k] * b[k,j];
  }
  send вектор-результат c управляющему процессу;
}
```

Рабочий процесс i вычисляет строку i результирующей матрицы c. Чтобы это сделать, он должен получить строку i исходной матрицы a и всю исходную матрицу b. Каждый рабочий процесс сначала получает эти значения от отдельного управляющего процесса. Затем рабочий процесс вычисляет свою строку результатов и отсылает ее обратно управляющему. (Или, возможно, исходные матрицы являются результатом предшествующих вычислений, а результирующая — входом для последующих; это пример распределенного конвейера.)

Управляющий процесс инициирует вычисления, собирает и выводит их результаты. В частности, сначала управляющий процесс посылает каждому рабочему соответствующую строку матрицы a и всю матрицу b. Затем управляющий процесс ожидает получения строк матрицы c от каждого рабочего. Схема управляющего процесса такова.

```
process coordinator {
  double a[n,n]; # исходная матрица a
  double b[n,n]; # исходная матрица b
  double c[n,n]; # результирующая матрица c
  инициализировать a и b;
  for [i = 0 to n-1] {
    send строку i матрицы a процессу worker[i];
    send всю матрицу b процессу worker[i];
  }
  for [i = 0 to n-1]
```

```
      receive строку i матрицы c от процесса worker[i];
    вывести результат, который теперь в матрице c;
}
```

Операторы send и receive, используемые управляющим процессом, — это *примитивы* (*элементарные действия*) *передачи сообщений*. Операция send упаковывает сообщение и пересылает его другому процессу; операция receive ожидает сообщение от другого процесса, а затем сохраняет его в локальных переменных. Подробно передача сообщений будет описана в главе 7 и использована в программировании многочисленных приложений в частях 2 и 3.

Как и ранее, предположим, что каждый рабочий процесс получает одну строку матрицы a и должен вычислить одну строку матрицы c. Однако теперь допустим, что у каждого процесса есть только один столбец, а не вся матрица b. Итак, в начальном состоянии рабочий процесс i имеет столбец i матрицы b. Имея лишь эти исходные данные, рабочий процесс может вычислить только значение c[i,i]. Для того чтобы рабочий процесс i мог вычислить всю строку матрицы c, он должен получить все столбцы матрицы b. Для этого можно использовать круговой конвейер (см. рис. 1.7, *б*), в котором по рабочим процессам циркулируют столбцы. Каждый рабочий процесс выполняет последовательность *раундов*; в каждом раунде он отсылает свой столбец матрицы b следующему процессу и получает другой ее столбец от предыдущего. Программа имеет следующий вид.

```
process worker[i = 0 to n-1] {
  double a[n];    # строка i матрицы a
  double b[n];    # один столбец матрицы b
  double c[n];    # строка i матрицы c
  double sum = 0.0; # для промежуточных произведений
  int nextCol = i;  # следующий столбец результатов
  receive строку i матрицы a и столбец i матрицы b;
  # вычислить c[i,i] = a[i,*] x b[*,i]
  for [k = 0 to n-1]
    sum = sum + a[k] * b[k];
  c[nextCol] = sum;
  #пустить по кругу столбцы и вычислить остальные c[i,*]
  for [j = 1 to n-1] {
    send мой столбец матрицы b следующему процессу;
    receive новый столбец матрицы b от предыдущего;
    sum = 0.0;
    for [k = 0 to n-1]
      sum = sum + a[k]*b[k];
    if (nextCol == 0)
      nextCol = n-1;
    else
      nextCol = nextCol-1;
    c[nextCol] = sum;
  }
  send вектор-результат c управляющему процессу;
}
```

В данной программе рабочие процессы упорядочены в соответствии с их индексами. (Для процесса n-1 следующим является процесс 0, а предыдущим для 0 — n-1.) Столбцы матрицы b передаются по кругу между рабочими процессами, поэтому каждый процесс в конце концов получит каждый столбец. Переменная nextCol отслеживает, куда в векторе c поместить очередное промежуточное произведение. Как и в первом вычислении, предполагается, что управляющий процесс отправляет строки матрицы a и столбцы матрицы b рабочим, а затем получает от них строки матрицы c.

Во второй программе использовано отношение между процессорами, которое называется *взаимодействующие равные* (interacting peers), или просто *равные*. Каждый рабочий процесс выполняет один и тот же алгоритм и взаимодействует с другими рабочими, чтобы вычислить свою часть необходимого результата. Дальнейшие примеры взаимодействующих равных мы увидим в частях 2 и 3. В одних случаях, как и здесь, каждый из рабочих процессов общается только с двумя своими соседями, в других — каждый из рабочих взаимодействует со всеми остальными процессами.

В первой из приведенных выше программ значения из матрицы `b` *дублируются* в каждом рабочем процессе. Во второй программе в любой момент времени у каждого процесса есть одна строка матрицы `a` и только один столбец матрицы `b`. Это снижает затраты памяти для каждого процесса, но вторая программа выполняется дольше первой, поскольку на каждой ее итерации каждый рабочий процесс должен отослать сообщение одному соседу и получить сообщение от другого. Данные программы иллюстрируют классическое противоречие между временем и пространством в вычислениях. В разделе 9.3 представлены другие алгоритмы для распределенного умножения матриц, иллюстрирующие дополнительные противоречия между временем и пространством.

1.9. Обзор программной нотации

В пяти предыдущих разделах были представлены примеры циклических схем в параллельном программировании: итеративный параллелизм, рекурсивный параллелизм, производители и потребители, клиенты и серверы, а также взаимодействующие равные. Многочисленные примеры этих схем еще будут приведены. В примерах также была введена программная нотация. В данном разделе дается ее обзор, хотя она очевидна из примеров.

Напомним, что параллельная программа содержит один или несколько процессов, а каждый процесс — это последовательная программа. Таким образом, наш язык программирования содержит механизмы и параллельного, и последовательного программирования. Нотация последовательных программ основана на базовых понятиях языков C, C++ и Java. В нотации параллельного программирования используются операторы `co` и декларации `process`. Они были представлены ранее и определяются ниже. В следующих главах будут определены механизмы синхронизации и межпроцессного взаимодействия.

1.9.1. Декларации

Декларация (*объявление*, или *определение*) переменной задает тип данных и перечисляет имена одной или нескольких переменных этого типа. При объявлении переменную можно инициализировать, например:

```
int i, j = 3;
double sum = 0.0;
```

Массив объявляется добавлением размера каждого измерения к имени массива. Диапазон индексов массива по умолчанию находится в пределах от `0` до значения, меньшего на 1, чем размер измерения. В качестве альтернативы можно непосредственно указать нижнюю и верхнюю границы диапазона. Массивы также можно инициализировать при их объявлении. Вот примеры:

```
int a[n];   # то же, что и "int a[0:n-1];"
int b[1:n]; # массив из n целых, b[1] ... b[n]
int c[1:n]=([n]0);  # вектор нулей
double c[n,n] = ([n] ([n] 1.0)); # матрица единиц
```

Каждая декларация сопровождается комментарием, который начинается знаком # (см. раздел 1.9.4). Последняя декларация говорит, что `c` — это матрица чисел двойной точности. Индексы каждого ее измерения находятся в пределах от `0` до `n-1`, а начальное значение каждого ее элемента — `1.0`.

1.9.2. Последовательные операторы

В операторе присваивания есть целевая переменная (слева), знак равенства и выражение (справа). Для операций инкремента (увеличения), декремента (уменьшения) и других присваиваний используется короткая форма этого оператора. Примеры:

```
a[n] = a[n] + 1;  # то же, что "a[n]++;"
x = (y+z) * f(x); # f(x) - вызов функции
```

Из управляющих операторов используются `if`, `while` и `for`. Простой оператор `if` имеет такой вид.

```
if (условие)
  оператор;
```

Здесь `условие` — это булево выражение (со значением "истина" или "ложь"), а `оператор` — одиночный оператор. Если при истинности условия нужно выполнить несколько операторов, они отделяются фигурными скобками.

```
if (условие) {
  оператор1;
  ...
  операторN;
}
```

В последующих главах такой список операторов часто обозначается через `S`. Оператор `if/then/else` имеет такой вид.

```
if (условие)
  оператор1;
else
  оператор2;
```

Операторы здесь также могут представлять собой списки операторов в фигурных скобках.

Оператор `while` имеет следующий общий вид.

```
while (условие) {
  оператор1;
  ...
  операторN;
}
```

[3]

Если `условие` имеет значение "истина", то выполняются вложенные операторы (тело цикла), а затем оператор `while` повторяется. Цикл `while` завершается, если `условие` имеет значение "ложь". Если в теле цикла только один оператор, фигурные скобки опускаются.

Операторы `if` и `while` идентичны соответствующим операторам в языках C, C++ и Java, но оператор `for` записывается более компактно. Его общий вид таков.

```
for [квантификатор1, ..., квантификаторM] {
  оператор1;
  ...
  операторN;
}
```

Каждый *квантификатор* вводит новую индексную переменную (параметр цикла), инициализирует ее и указывает диапазон ее значений. Квадратные скобки вокруг квантификаторов используются для определения диапазона значений, как и в декларациях массивов.

---

[3] В действительности его общий вид — `while (условие) оператор;`. Это же относится и к следующему оператору `for`. — *Прим. перев.*

Предположим, что a[n] — это массив целых чисел. Тогда следующий оператор инициализирует каждый элемент массива a[i] значением i.

```
for [i = 0 to n-1]
  a[i] = i;
```

Здесь i — новая переменная; ее не обязательно определять выше в программе. Область видимости переменной i — тело данного цикла for. Ее начальное значение 0, и она принимает по порядку значения от 0 до n-1.

Предположим, что m[n,n] — массив целых чисел. Рассмотрим оператор for с двумя квантификаторами.

```
for [i = 0 to n-1, j = 0 to n-1]
  m[i,j] = 0;
```

Этому оператору эквивалентны вложенные операторы for.

```
for [i = 0 to n-1]
  for [j = 0 to n-1]
    m[i,j] = 0;
```

В обоих случаях n2 значений матрицы m инициализируются нулями. Рассмотрим еще два примера квантификаторов.

```
[i = 1 to n by 2]       #нечетные значения от 1 до n
[i = 0 to n-1 st i!=x] #каждое значение, кроме i==x
```

Обозначение st во втором квантификаторе — это сокращение слов "such that" ("такой, для которого").

Операторы for записываются с использованием синтаксиса, приведенного выше, по нескольким причинам. Во-первых, этим подчеркивается отличие наших операторов for от тех же операторов в языках C, C++ и Java. Во-вторых, такая нотация предполагает их использование с массивами, у которых индексы заключаются в квадратные, а не круглые скобки. В-третьих, наша запись упрощает программы, поскольку избавляет от необходимости объявлять индексную переменную. (Сколько раз вы забывали это сделать?) В-четвертых, зачастую удобнее использовать несколько индексных переменных, т.е. записывать несколько квантификаторов. И, наконец, те же формы квантификаторов используются в операторах co и декларациях process.

1.9.3. Параллельные операторы, процессы и процедуры

По умолчанию операторы выполняются последовательно, т.е. один за другим. Оператор co (concurrent — параллельный, происходящий одновременно) указывает, что несколько операторов могут выполняться параллельно. В одной форме оператор co имеет несколько *ветвей* (arms).

```
co оператор1;
// ...
// операторN;
oc
```

Каждая ветвь содержит оператор (или список операторов). Ветви отделяются символом параллелизма //. Оператор, приведенный выше, означает следующее: начать параллельное выполнение всех операторов, затем ожидать их завершения. Оператор co, таким образом, завершается после выполнения всех его операторов.

В другой форме оператор co использует один или несколько квантификаторов, которые указывают, что набор операторов должен выполняться параллельно для каждой комбинации

значений параметров цикла. Например, следующий тривиальный оператор инициализирует массивы a[n] и b[n] нулями.

```
co[i = 0 to n-1] {
  a[i] = 0; b[i] = 0;
}
```

Этот оператор создает n процессов, по одному для каждого значения переменной i. Область видимости счетчика — описание процесса, и у каждого процесса свое, отличное от других, значение переменной i. Две формы оператора co можно смешивать. Например, одна ветвь может иметь квантификатор в квадратных скобках, а другая — нет.

Декларация процесса является, по существу, сокращенной формой оператора co с одной ветвью и/или одним квантификатором. Она начинается ключевым словом process и именем процесса, а заканчивается ключевым словом end. Тело процесса содержит определения локальных переменных, если такие есть, и список операторов.

В следующем простом примере определяется процесс foo, который суммирует числа от 1 до 10, записывая результат в глобальную переменную x.

```
process foo {
  int sum = 0;
  for [i = 1 to 10]
    sum += i;
  x = sum;
}
```

Декларация process записывается на синтаксическом уровне декларации procedure; это не оператор, в отличие от co. Кроме того, объявляемые процессы выполняются в фоновом режиме, тогда как выполнение оператора, следующего за оператором co, начинается после завершения процессов, созданных этим co.

Еще один простой пример: следующий процесс записывает значения от 1 до n в стандартный файл вывода.

```
process bar1 {
  for [i = 1 to n]
    write(i);  # то же, что "printf("%d\n",i);"
}
```

Массив процессов объявляется добавлением квантификатора (в квадратных скобках) к имени процесса.

```
process bar2[i = 1 to n] {
  write(i);
}
```

И bar1, и bar2 записывают в стандартный вывод значения от 1 до n. Однако порядок, в котором их записывает массив процессов bar2, *недетерминирован*, поскольку массив bar2 состоит из n отдельных процессов, выполняемых в произвольном порядке. Существует n! различных порядков, в которых этот массив процессов мог бы записать числа (n! — число перестановок n значений).

Процедуры и функции объявляются и вызываются так же, как это делается в языке C, например, так.

```
int addOne(int v) { # функция возвращает целое число
  return (v + 1);
}
main() {  # "void"-процедура
  int n, sum;
  read(n);  # прочитать целое число из stdin
  for [i = 1 to n]
    sum = sum + addOne(i);
```

```
    write("Окончательным значением является ", sum);
}
```

Если входное значение n равно 5, эта программа выведет такую строку.

```
Окончательным значением является 20
```

1.9.4. Комментарии

Комментарии записываются двумя способами. Однострочные комментарии начинаются символом # и завершаются в конце строки. Многострочные комментарии начинаются символами /* и оканчиваются символами */. Для однострочных комментариев используется символ #, поскольку символ однострочных комментариев // языков C++ и Java уже давно использовался в параллельном программировании как разделитель ветвей в параллельных операторах.

Утверждение — это предикат, определяющий условие, которое должно выполняться в некоторой точке программы. (Утверждения подробно описаны в главе 2.) Утверждения можно рассматривать как предельно точные комментарии, поэтому они записываются в отдельных строках, начинающихся двумя символами #:

```
## x > 0
```

Данный комментарий утверждает, что значение x положительно.

Историческая справка

Как уже отмечалось, параллельное программирование возникло в 1960-х годах после появления независимых аппаратных контроллеров (каналов). Операционные системы были первыми программными системами, организованными как многопоточные параллельные программы. Исследование и первоначальные прототипы привели на рубеже 1960-х и 1970-х годов к современным операционным системам. Первые книги по операционным системам появились в начале 1970-х годов.

Создание компьютерных сетей привело в 1970-х годах к разработке распределенных систем. Изобретение в конце 1970-х сети Ethernet существенно ускорило темпы развития. Почти сразу появилось изобилие новых языков, алгоритмов и приложений; их создание стимулировалось развитием аппаратного обеспечения. Например, как только рабочие станции и локальные сети стали относительно дешевыми, была разработана модель вычислений типа клиент-сервер; развитие сети Internet привело к рождению языка Java, Web-броузеров и множества новых приложений.

Первые мультипроцессоры появились в 1970-х годах, и наиболее заметными из них были SIMD-мультипроцессоры Illiac, разработанные в университете Иллинойса. Первые машины были дорогими и специализированными, однако в течение многих лет трудоемкие научные вычисления выполнялись на векторных процессорах. Изменения начались в середине 1980-х годов с изобретения гиперкубовых машин в Калифорнийском технологическом институте и их коммерческой реализации фирмой Intel. Затем фирма Thinking Machines представила Connection Machine с массовым параллелизмом. Кроме того, фирма Cray Research и другие производители векторных процессоров начали производство многопроцессорных версий своих машин. В течение нескольких лет появлялись и вскоре исчезали многочисленные компании и машины. Однако сейчас группа производителей машин достаточна стабильна, а высокопроизводительные вычисления стали почти синонимом обработки с массовым параллелизмом.

В исторических справках следующих глав описываются разработки, связанные с самими этими главами. Ниже приведено несколько общих ссылок, касающихся аппаратного обеспечения, операционных систем, распределенных систем и параллельных вычислений.

Это книги, к которым автор чаще всего обращался за справкой при написании данной книги. (Чтобы побольше узнать о какой-то теме, можно использовать хорошую поисковую программу в Internet.)

На данный момент классической книгой по архитектуре компьютеров является книга [Hennessy, Patterson, 1996]. Если вы хотите узнать больше о кэш-памяти, соединительных сетях и работе мультипроцессоров, начинайте с нее. В книге [Hwang, 1993] описана архитектура высокопроизводительных компьютеров, а книга [Almasi, Gottlieb, 1994] дает обзор приложений синхронных параллельных вычислений, программного обеспечения и архитектуры.

По операционным системам наиболее широко известны учебники [Tanenbaum, 1992] и [Silberschatz, Peterson, Galvin, 1998]. Книга [Tanenbaum, 1995] также дает превосходный обзор систем распределенного программирования. Книга [Mullender, 1993] содержит великолепный набор глав по всем темам распределенных систем, включая надежность и отказоустойчивость, обработку транзакций, файловые системы, системы реального времени и безопасность. Два дополнительных учебника — [Bacon, 1998] и [Bernstein, Lewis, 1993] — охватывают многопоточные параллельные системы, включая системы распределенных баз данных.

За последние несколько лет было написано множество книг по синхронным параллельным вычислениям. Два конкурирующих издания — [Kumar, Grama, Gupta, Karypis, 1994] и [Quinn, 1994] — описывают и анализируют параллельные алгоритмы для решения многочисленных комбинаторных и численных задач. Обе книги до некоторой степени охватывают темы программного и аппаратного обеспечения, но особое внимание уделяется разработке и анализу параллельных алгоритмов.

Еще четыре книги посвящены программным аспектам параллельных вычислений. В [Brinch Hansen, 1995] рассматриваются интересные вычислительные задачи и решаются с помощью небольшого числа парадигм программирования; задачи описаны исключительно ясно. В книге [Foster, 1995] представлены идеи и средства параллельного программирования, в особенности для машин с распределенной памятью. В книге есть отличные описания двух наиболее актуальных инструментальных средств — High Performance Fortran (HPF) и Message Passing Interface (MPI). В [Wilson, 1995] описаны четыре важные модели программирования (параллелизм по данным, разделяемые переменные, передача сообщений, генеративное взаимодействие) и показано, как с их помощью решать научные и инженерные задачи. В одном из приложений этой книги превосходно изложена краткая история основных событий в параллельных вычислениях. Последняя книга [Wilkinson, Allen, 1999] описывает множество приложений и показывает, как писать параллельные программы для их решения. Основное внимание в книге уделено передаче сообщений, но описываются также вычисления с разделяемыми переменными.

Книга [Foster, Kesselman, 1999] представляет новый многообещающий подход к распределенным быстродействующим вычислениям, использующим так называемую вычислительную сеть. (Этот подход описан в главе 12.) Книга начинается исчерпывающим введением в вычислительные сети, а затем рассматривает приложения, инструменты программирования, сервис и инфраструктуру. Главы книги написаны экспертами, которые работают над тем, чтобы вычислительные сети стали реальностью.

Литература

Almasi, G. S., and A. Gottlieb. 1994. *Highly Parallel Computing.* 2nd ed. Menlo Park, CA: Benjamin/Cummings.

Bacon, J. 1998. *Concurrent Systems: Operating Systems, Database and Distributed Systems: An Integrated Approach.* 2nd ed. Reading, MA: Addison-Wesley.

Bernstein, A. J., and P. M. Lewis. 1993. *Concurrency in Programming and Database Systems.* Boston, MA: Jones and Bartlett.

Brinch Hansen, P. 1995. *Studies in Computational Science.* Englewood Cliffs, NJ: Prentice-Hall.

Foster, I. 1995. *Designing and Building Parallel Programs.* Reading, MA: Addison-Wesley.

Foster, I., and C. Kesselman, eds. 1999. *The Grid: Blueprint for a New Computing Infrastructure.* San Francisco, CA: Morgan Kaufmann.

Hennessy, J. L., and D. A. Patterson. 1996. *Computer Architecture: A Quantitative Approach,* 2nd ed. San Francisco, CA: Morgan Kaufmann.

Hwang, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* New York, NY: McGraw-Hill.

Kumar, V., A. Grama, A. Gupta, and G. Karypis. 1994. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Menlo Park, CA: Benjamin/Cummings.

Mullender, S., ed. 1993. *Distributed Systems,* 2nd ed. Reading, MA: ACM Press and Addison-Wesley.

Quinn, M. J. 1994. *Parallel Computing: Theory and Practice.* New York, NY: McGraw-Hill.

Silberschatz, A., J. Peterson, and P. Galvin. 1998. *Operating System Concepts,* 5th ed. Reading, MA: Addison-Wesley.

Tanenbaum, A. S. 1992. *Modern Operating Systems.* Englewood Cliffs, NJ: Prentice-Hall.

Tanenbaum, A. S. 1995. *Distributed Operating Systems.* Englewood Cliffs, NJ: Prentice-Hall.

Wilkinson, B., and M. Allen. 1999. *Parallel Programming: Techniques and Applications Using* Networked Workstations and РАЛ= 2,21*arallel Computers.* Englewood Cliffs, NJ: Prentice-Hall.

Wilson, G. V. 1995. *Practical Parallel Programming.* Cambridge, MA: MIT Press.

Упражнения

1.1. Определите характеристики доступных вам многопроцессорных машин. Сколько процессоров в каждой машине и каковы их рабочие частоты? Насколько велик размер их кэш-памяти, как она организована? Каково время доступа? Какой используется протокол согласования памяти? Как организована связующая сеть? Каково время удаленного доступа к памяти или передачи сообщения?

1.2. Многие задачи можно решить более эффективно с помощью параллельной, а не последовательной программы (конечно, при наличии соответствующего аппаратного обеспечения). Рассмотрите программы, которые вы писали раньше, и выберите две из них, которые можно переписать как параллельные. Одна из них должна быть итеративной, а другая — рекурсивной. Затем: а) запишите кратко условия задач и б) разработайте псевдокод параллельных программ, решающих поставленные задачи.

1.3. Рассмотрите умножение матриц в разделе 1.4:

   а) напишите последовательную программу для решения этой задачи. Размер матрицы `n` должен быть аргументом командной строки. Инициализируйте каждый элемент матриц `a` и `b` значением `1.0` (тогда значение каждого элемента результирующей матрицы `c` будет `n`);

   б) напишите параллельную программу для решения этой задачи. Вычислите полосы результата параллельно, используя `P` рабочих процессов. Размер матрицы `n` и число рабочих процессов `P` должны быть аргументами командной строки. Вновь инициализируйте каждый элемент матриц `a` и `b` значением `1.0`;

   в) сравните производительность ваших программ. Поэкспериментируйте с разными значениями параметров `n` и `P`. Подготовьте график результатов и объясните, что вы заметили;

г) преобразуйте ваши программы для умножения прямоугольных матриц. Размер матрицы a должен быть `p×q`, а матрицы `b` — `q×r`. Тогда размер результирующей матрицы будет `p×r`. Повторите часть *в* данного упражнения для новых программ.

1.4. В программах умножения матриц при вычислении промежуточных произведений используется умножение пар элементов и сложение результатов. Все умножения можно выполнять параллельно. Пары произведений тоже можно складывать параллельно:

а) создайте двоичное дерево выражений для иллюстрации работы этого процесса. Предположим, что векторы имеют длину `n`, и для простоты допустим, что `n` является степенью числа 2. Листья дерева должны быть элементами вектора (строки матрицы a и столбца матрицы `b`). Другие узлы дерева должны быть отмечены операциями сложения или умножения;

б) многие современные процессоры являются так называемыми сверхскалярными процессорами. Это значит, что они могут начать выполнение сразу нескольких инструкций. Рассмотрим машину, которая может начать выполнять сразу две инструкции. Напишите псевдокод на уровне языка ассемблера для реализации двоичного дерева выражения, построенного вами в части *а*. Предположим, что существуют инструкции для загрузки, сложения и умножения регистров и доступно любое число регистров. Максимизируйте число пар инструкций, выполнение которых можно начать одновременно;

в) предположим, что загрузка регистра занимает 1 такт работы процессора, сложение — тоже 1, а умножение — 8 тактов. Каким будет время выполнения вашей программы?

1.5. В первой строке первой программы для параллельного умножения матриц (см. раздел 1.4) находится оператор `co` для строк, а во второй строке — оператор `for` для столбцов. Допустим, что оператор `co` изменен на `for`, `for` — на `co`, а остальная часть программы осталась без изменений. Программа вычисляет строки последовательно, а столбцы для каждой строки — параллельно:

а) будет ли программа верна? Точнее, будут ли вычисляться те же результаты?

б) будет ли программа настолько же эффективной? Объясните свой ответ.

1.6. Точки на единичной окружности с центром в начале координат определяются функцией `f(x) = sqrt(1-x2)`.[4] Напомним, что площадь круга вычисляется по формуле $\pi r2$, где `r` — радиус. Используйте адаптивную квадратурную программу из раздела 1.5 для аппроксимации значения π путем вычисления площади первого (верхнего правого) квадранта единичной окружности и умножения результата на 4. (Можно также проинтегрировать от `0.0` до `1.0` функцию `f(x) = 4/(1+x2)`.)

1.7. Рассмотрите задачу квадратуры, описанную в разделе 1.5:

а) напишите четыре программы для ее решения: 1) последовательную итеративную программу, в которой используется фиксированное число интервалов, 2) последовательную рекурсивную программу с адаптивной квадратурой, 3) параллельную итеративную программу с фиксированным числом интервалов и 4) параллельную рекурсивную программу, использующую адаптивную квадратуру. Проинтегрируйте функцию с графиком интересной формы, например `sin(x)*exp(x)`. Ваши программы должны использовать параметры командной строки для интервала значений `x`, числа отрезков (при фиксированной квадратуре) и значения `EPSILON` (при адаптивной квадратуре);

[4] Точнее, эта функция определяет точки верхней полуокружности. — *Прим. перев.*

б) поэкспериментируйте со своими программами при различных значениях аргументов. Какова длительность их выполнения? Насколько точны ответы? Быстро ли сходятся программы? Объясните свои наблюдения.

1.8. Программа параллельной адаптивной квадратуры (см. раздел 1.5) создает большое число процессов, обычно превышающее число процессоров в системе:

а) измените программу, чтобы она создавала приблизительно `T` процессов, где `T` — аргумент командной строки, определяющий пороговое значение. Используйте глобальную переменную для хранения числа созданных вами процессов. (В этой задаче предположим, что глобальную переменную можно безопасно увеличивать.) Если внутри тела функции `quad` создано больше, чем `T` процессов, используйте последовательную рекурсию. В противном случае используйте параллельную рекурсию и каждый раз добавляйте 2 к глобальному счетчику;

б) реализуйте и протестируйте параллельную программу из текста книги и из вашего ответа к пункту *а*. Выберите для интегрирования интересную функцию, поэкспериментируйте с различными значениями переменных `T` и `EPSILON`. Сравните производительность обеих программ.

1.9. Напишите последовательную рекурсивную программу для реализации алгоритма быстрой сортировки массива из `n` чисел. Преобразуйте свою программу для использования рекурсивного параллелизма. Будьте внимательны, чтобы обеспечить независимость параллельных вызовов. Реализуйте обе программы и сравните их производительность.

1.10. Соберите данные о каналах ОС Unix. Как они реализованы в вашей системе? Каков максимальный размер канала? Как синхронизируются операции `read` и `write`? Проверьте, можете ли вы провести эксперимент, который приведет к блокировке операции `write` из-за переполнения канала. Можете ли вы создать параллельную программу, которая "зависнет", т.е. все процессы будут ждать друг друга?

1.11. Большая часть вычислительных возможностей серверов сейчас используется для электронной почты, Web-страниц, файлов и т.д. Какие типы серверов используются в ваших вычислительных средствах? Выберите один из серверов (если есть) и выясните, как на нем организовано программное обеспечение. Какие он выполняет процессы (потоки)? Как они планируются? Что представляют собой клиенты? Выделяется ли в нем по одному потоку на клиентский запрос, существует ли фиксированное число серверных потоков, или это происходит как-то иначе? Какие данные разделяются потоками сервера? Когда и зачем потоки сервера синхронизируются друг с другом?

1.12. Рассмотрите две программы для распределенного умножения матриц из раздела 1.8:

а) сколько сообщений отсылается и принимается каждой программой? Каковы размеры сообщений? Не забудьте, что во второй программе есть и управляющий процесс;

б) измените программу, чтобы она использовала `P` рабочих процессов, где `P` — делитель числа `n`. В частности, каждый рабочий процесс должен вычислять не одну строку, а `n/P` строк (или столбцов) результата;

в) сколько сообщений отсылается программой для части *б* и какого они размера?

1.13. Матрица `T` является транспонированной матрицей `M`, если для всех значений `i` и `j` выполняется равенство `T[i,j] = M[j,i]`:

а) напишите параллельную программу, которая с использованием разделяемых переменных транспонирует матрицу `M` размером `n`×`n`. Используйте `P` рабочих процессов. Для простоты предположим, что число `n` кратно `P`;

б) напишите параллельную программу транспонирования квадратной матрицы M, в которой используется пересылка сообщений. Вновь используйте P рабочих процессов и предположите, что n кратно P. Выясните, как распределить данные и собрать результаты;

в) измените ваши программы, чтобы они обрабатывали случай, когда n не кратно P;

г) поэкспериментируйте с вашими программами при различных значениях n и P. Какова их производительность?

1.14. Семейство программ `grep` ОС Unix выполняет поиск шаблонов в файлах. Напишите упрощенную версию программы `grep`, которая использует два аргумента: цепочку символов и имя файла. Все строки файла с найденной цепочкой программа должна выводить в файл `stdout`:

а) измените программу, чтобы она обрабатывала один за другим два файла (добавьте третий аргумент для указания имени файла);

б) измените программу, чтобы она выполняла поиск в двух файлах параллельно. Оба процесса должны выводить данные в файл `stdout`;

в) поэкспериментируйте с вашими программами к пунктам *а* и *б*. Их выход должен отличаться, по крайней мере, в некоторые моменты времени. Более того, выход параллельной программы может не всегда быть одним и тем же. Проверьте, можете ли вы наблюдать это явление.

Часть 1

Программирование с разделяемыми переменными

В последовательных программах часто используются разделяемые переменные, например, для хранения глобальных структур данных, но многое доказывает, что лучше обходиться *без них*.

Вместе с тем, параллельные программы всецело зависят от разделяемых компонентов, поскольку процессы могут работать над одной задачей, только взаимодействуя. А единственный способ взаимодействия — возможность для одного процесса записывать во *что-то*, откуда другой процесс читает. Этим чем-то может быть разделяемая переменная или разделяемый канал связи. Поэтому взаимодействие программируется как запись и чтение разделяемых переменных или как передача и прием сообщений.

Взаимодействие повышает необходимость синхронизации. Существует два основных ее типа: взаимное исключение и синхронизация условий. Взаимное исключение встречается, когда два процесса должны по очереди обращаться к таким разделяемым объектам, как записи в системе заказа авиабилетов. Синхронизация условий происходит, когда одному процессу приходится ждать другой процесс, например, когда процесс-потребитель ожидает данные от процесса-производителя.

В части 1 показано, как писать параллельные программы, в которых взаимодействие и синхронизация процессов выполняются с помощью разделяемых переменных. Рассмотрены различные многопоточные и параллельные приложения. Программы с разделяемыми переменными чаще всего выполняются на машинах с разделяемой памятью, поскольку в них каждый процессор может получить непосредственный доступ к каждой переменной. Однако программирование с разделяемыми переменными можно использовать и в машинах с распределенной памятью, если оно поддерживается программной реализацией так называемой *распределенной разделяемой памяти*; как это делается, показано в главе 10.

Основные понятия процессов и синхронизации представлены с помощью ряда небольших примеров в главе 2. В первой половине этой главы описаны способы распараллеливания программ, проиллюстрирована необходимость синхронизации, определены элементарные действия и представлен оператор для программирования синхронизации `await`. Во второй половине главы рассмотрена семантика параллельных программ и представлено несколько таких ключевых концепций, как взаимное вмешательство, глобальные инварианты, свойства безопасности и справедливость — с этим мы еще не раз встретимся в следующих главах.

В главе 3 рассмотрены два основных типа синхронизации — барьеры и блокировки, и показано, как реализовать их с помощью инструкций, которые есть у любого процессора. Блокировки используются для решения классической задачи о критической секции, которая встречается в большинстве параллельных программ. Барьеры — это фундаментальный для параллельных программ метод синхронизации. В последних двух разделах этой главы рас-

смотрены и проиллюстрированы приложения двух важных моделей параллельного программирования: "параллелизм по данным" и "портфель задач".

В главе 4 описаны семафоры, которые упрощают программирование взаимного исключения и синхронизации условий (сигнализации). В этой главе наряду с несколькими другими представлены две классические задачи параллельного программирования — об "обедающих философах" и о "читателях и писателях". В конце главы описано и продемонстрировано использование библиотеки потоков стандарта POSIX (Pthreads), которая на машинах с разделяемой памятью поддерживает потоки и семафоры.

В главе 5 описаны мониторы — механизм программирования более высокого уровня, который был впервые предложен в 1970-х годах. Интерес к нему через некоторое время снизился, но после реализации в языке Java он снова становится популярным. Мониторы также используются для организации и синхронизации кода в операционных системах и других многопоточных программных системах. Использование мониторов проиллюстрировано в нескольких интересных примерах этой главы. Тут представлены задачи коммуникационных буферов, "читателей и писателей", таймеров, "спящего парикмахера" (еще одна классическая задача) и планирования диска. В разделе 5.4 описаны потоки и синхронизированные методы языка Java, а также представлены различные способы защиты данных в языке Java. В разделе 5.5 показано, как программировать мониторы с использованием библиотеки Pthreads.

В главе 6 показано, как реализовать процессы, семафоры и мониторы на одиночных процессорах и мультипроцессорах с разделяемой памятью. Основу любой реализации языка параллельного программирования или библиотеки процедур составляет *ядро* из структур данных и процедурных примитивов. В конце данной главы показано, как реализовать мониторы, используя семафоры.

Глава 2

Процессы и синхронизация

Параллельным программам присуща более высокая сложность по сравнению с последовательными. Они соотносятся с последовательными программами, как шахматы с шашками или бридж с "дураком": и те и другие интересны, но первые гораздо интеллектуальнее последних.

В этой главе исследуется "игра" в параллельное программирование, здесь подробнее рассмотрены ее правила, фигуры и стратегии. Эти правила являются формальными инструментами, которые помогают понимать и разрабатывать правильные программы. Фигуры — это конструкции языка для описания параллельных вычислений, а стратегии — полезные методы программирования.

В предыдущей главе были представлены процессы и синхронизация, а также приведены примеры их использования. Здесь они рассматриваются подробнее. В разделе 2.1 кратко описывается семантика (смысл) параллельных программ и представлены пять основных понятий: состояние программы, неделимое действие, история, свойства безопасности и живучести. В разделах 2.2 и 2.3 эти понятия поясняются двумя примерами — поиск шаблона в файле и поиск в массиве максимального значения. Изучаются также способы распараллеливания программ, рассматривается необходимость неделимых действий и синхронизации. В разделе 2.4 определяются неделимые действия (примитивы) и вводится оператор `await` как средство выражения примитивов и синхронизации. В разделе 2.5 показано, как программировать синхронизацию, которая встречается в программах типа "производитель-потребитель".

В разделе 2.6 представлен краткий обзор аксиоматической семантики последовательных и параллельных программ. Новая фундаментальная проблема, возникающая в параллельных программах, — это возможность взаимного влияния (вмешательства). В разделе 2.7 описаны четыре метода его устранения: непересекающиеся множества переменных, ослабленные утверждения, глобальные инварианты и синхронизация. Наконец, в разделе 2.8 показано, как доказывать выполнение свойств безопасности, и определены стратегии планирования и понятие справедливости.

Многие концепции представлены в этой главе подробно, поэтому их, возможно, нелегко понять при первом прочтении. Но, пожалуйста, будьте настойчивы, изучайте примеры и при необходимости возвращайтесь к этой главе. Представленные в ней концепции важны, поскольку обеспечивают основу для разработки и понимания параллельных программ. Дисциплинированный подход важен для последовательных программ и обязателен для параллельных, поскольку порядок, в котором выполняются процессы, не детерминирован. В любом случае, приступим к игре!

2.1. Состояние, действие, история и свойства

Состояние параллельной программы состоит из значений переменных программы в некоторый момент времени. Переменные могут быть явно определенными программистом или неявными (вроде программного счетчика каждого процесса), хранящими скрытую информацию о состоянии. Параллельная программа начинает выполнение в некотором исходном состоянии. Каждый процесс программы выполняется независимо, и по мере выполнения он проверяет и изменяет состояние программы.

Процесс выполняет последовательность операторов. Оператор, в свою очередь, реализуется последовательностью *неделимых действий*. Эти действия проверяют или изменяют со-

стояние программы *неделимым образом*. Примерами неделимых действий являются непрерываемые машинные инструкции, которые загружают и сохраняют слова памяти.

Выполнение параллельной программы приводит к *чередованию* последовательностей неделимых действий, производимых каждым процессом. Конкретное выполнение каждой программы может быть рассмотрено как *история* $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$, где s_0 — начальное состояние. Переходы между состояниями осуществляются неделимыми действиями, изменяющими состояние. Историю также называют *трассой* последовательности состояний. Даже параллельное выполнение можно представить в виде линейной истории, поскольку параллельная реализация набора неделимых действий эквивалентна их выполнению в некотором последовательном порядке. Изменение состояния, вызванное неделимым действием, неразделимо, и, следовательно, на него не могут повлиять неделимые действия, производимые примерно в это же время.

Каждое выполнение параллельной программы порождает историю. Для всех, кроме самых тривиальных программ, число возможных историй громадно. Дело в том, что следующим в истории может стать неделимое действие любого процесса. Следовательно, существует много способов чередования действий, даже если выполнение программы всегда начинается в одном и том же исходном состоянии. Кроме того, в каждом процессе обычно есть условные операторы, и, следовательно, возможны различные действия при различных изменениях в состоянии.

Цель синхронизации — исключить нежелаемые истории параллельной программы. Взаимное исключение состоит в комбинировании неделимых действий, реализуемых непосредственно аппаратным обеспечением в виде последовательностей действий, которые называются *критическими секциями*. Они должны быть неделимыми, т.е. их нельзя прервать действиями других процессов, которые ссылаются на те же переменные. *Синхронизация по условию (условная синхронизация)* означает, что действие будет осуществлено, когда состояние будет удовлетворять заданному логическому условию. Обе формы синхронизации могут приостанавливать процессы, ограничивая набор последующих неделимых действий.

Свойством программы называется атрибут, который является истинным при любой возможной истории программы и, следовательно, при всех ее выполнениях. Есть два типа свойств: безопасность и живучесть. *Свойство безопасности* заключается в том, что программа никогда не попадает в "плохое" состояние (при котором некоторые переменные могут иметь нежелательные значения). *Свойство живучести* означает, что программа в конце концов всегда попадает в "хорошее" состояние, т.е. состояние, в котором все переменные имеют желаемые значения.

Примером свойства безопасности является *частичная корректность* (*правильность*). Программа частично корректна (правильна), если правильно ее заключительное состояние (при условии, что программа завершается). Если программе не удается завершить выполнение, она может никогда не выдать правильный результат, но не существует такой истории, при которой программа завершается, не выдавая правильного результата. *Завершимость* — пример свойства живучести. Программа завершается, если завершается каждый цикл или вызов процедуры, т.е. длина каждой истории конечна. *Тотальная* (*полная*) *корректность* программы — это свойство, объединяющее частичную корректность и завершимость: программа полностью корректна, если она всегда завершается, выдавая при этом правильный результат.

Взаимное исключение — это пример свойства безопасности в параллельной программе. При плохом состоянии два процесса такой программы одновременно выполняют действия в разных критических секциях. Возможность в конце концов войти в критическую секцию — пример свойства живучести в параллельной программе. В хорошем состоянии каждый процесс выполняется в своей критической секции.

Как же демонстрировать, что данная программа обладает желаемым свойством? Обычный подход состоит в *тестировании*, или *отладке*. Его можно охарактеризовать фразой "запусти программу и посмотри, что получится". Это соответствует перебору некоторых возможных историй программы и проверке их приемлемости. Недостаток такой проверки состоит в том, что каждый тест касается только одной истории выполнения, а ограниченное число тестов вряд ли способно продемонстрировать отсутствие плохих историй.

Второй подход — использование *операторных рассуждений*, которые можно назвать "исчерпывающий анализ случаев" (перебираются все возможные истории выполнения программы). Для этого анализируются способы чередования неделимых действий процессов. К сожалению, в параллельной программе число возможных историй обычно очень велико (поэтому метод "изнурителен"[5]). Предположим, что параллельная программа содержит n процессов и каждый из них выполняет последовательность из m неделимых действий. Тогда число различных историй программы составит $(n \cdot m)\,!\,/\,(m!^n)$. Для программы из трех процессов, каждый из которых выполняет всего две неделимые операции, возможны 90 различных историй! (Числитель в формуле — это количество возможных перестановок из $n \cdot m$ действий. Но, поскольку каждый процесс выполняет последовательность действий, для него возможен только один порядок следования m действий; знаменатель отбрасывает все варианты с неправильным порядком следования. Эта формула дает количество, равное числу способов перемешать n колод по m карт в каждой, при условии, что относительный порядок карт в каждой колоде сохраняется.)

Третий подход — применение *утвердительных рассуждений* (assertional reasoning); его можно назвать "абстрактный анализ". В этом методе формулы логики предикатов называются *утверждениями* и используются для описания наборов состояний — например, всех состояний, у которых `x > 0`. Неделимые действия рассматриваются как *предикатные преобразователи*, поскольку они меняют состояние программы, удовлетворяющее одному предикату, на состояние, удовлетворяющее другому. Преимуществом данного подхода является компактное представление состояний и их преобразований. Но еще важнее то, что он приводит к методу построения и анализа программ, согласно которому объем работы прямо пропорционален числу неделимых действий в программе.

Используем метод утверждений как инструмент построения и анализа решений многих нетривиальных задач. При разработке алгоритмов также будет применяться метод операторных рассуждений. Наконец, многие программы этой книги были протестированы. Однако в результате тестирования можно только обнаружить наличие ошибок, а не гарантировать их отсутствие. Кроме того, параллельные программы очень сложны в тестировании и отладке, поскольку, во-первых, трудно остановить сразу все процессы и проверить их состояние, и, во-вторых, в общем случае каждое выполнение программы приводит к новой истории.

2.2. Распараллеливание: поиск образца в файле

В главе 1 рассмотрено несколько типов приложений и показано, как их можно реализовать с помощью параллельных программ. Теперь возьмем одну простую задачу и подробно изучим способы ее распараллеливания.

Рассмотрим задачу поиска всех экземпляров шаблона `pattern` в файле `filename`. Переменная `pattern` — это заданная строка; переменная `filename` — имя файла. Эта задача без труда разрешима в ОС Unix на командном уровне с помощью одной из команд семейства `grep`, например:

```
grep pattern filename
```

В результате создается один процесс. Он выполняет нечто подобное следующей последовательной программе.

```
string line;
прочитать строку ввода из stdin в line;
while (!EOF) {  # EOF - это конец файла
  искать pattern в line;
  if (pattern есть в line)
```

[5] Игра слов: "exhaustive" означает как "исчерпывающий", так и "истощающий", "изнурительный". — *Прим. перев.*

```
        вывести line;
      прочитать следующую строку ввода;
    }
```

Теперь желательно выяснить два основных вопроса: можно ли распараллелить эту программу? Если да, то как?

Основное требование для возможности распараллеливания любой программы состоит в том, что она должна содержать независимые части, как это описано в разделе 1.4. Две части взаимно зависимы, если каждая из них порождает результаты, необходимые для другой; это возможно, только если они считывают и записывают разделяемые переменные. Следовательно, две части программы *независимы*, если они не выполняют чтение и запись одних и тех же переменных. Более точное определение таково.

(2.1) **Независимость параллельных процессов.** Пусть *множество чтения* части программы — это переменные, которые она считывает, но не изменяет. Пусть *множество записи* части программы — это переменные, в которые она записывает (и, возможно, читает их). Две части программы являются *независимыми*, если пересечение их множеств записи пусто.

Чтение или запись любой переменной неделимо. Это относится как к простым переменным (таким как целые), которые записываются в отдельные слова памяти, так и к отдельным элементам массивов или структур (записей).

Из предшествующего определения следует, что две части программы независимы, если обе они только считывают разделяемые переменные, или каждая часть считывает переменные, отличные от тех, которые другая часть записывает. Иногда две части программы могут безопасно выполняться параллельно, даже производя запись в одни и те же переменные. Однако это возможно, если не важен порядок, в котором происходит запись. Например, если несколько процессов периодически обновляют графический экран, и любой порядок выполнения обновлений не портит вида экрана.

Вернемся к задаче поиска шаблона в файле. Какие части программы независимы и, следовательно, могут быть выполнены параллельно? Программа начинается чтением первой строки ввода; это *должно* быть выполнено перед всеми остальными действиями. После этого программа входит в цикл поиска шаблона, выводит строку, если шаблон был найден, а затем считывает новую строку. Вывести строку до того, как в ней был выполнен поиск шаблона, нельзя, поэтому первые две строки цикла выполнить параллельно невозможно. Однако можно прочитать следующую строку входа во время поиска шаблона в предыдущей строке и возможной ее печати. Следовательно, рассмотрим другую, параллельную, версию предыдущей программы.

```
string line;
прочитать входную строку из stdin в line;
while (!EOF) {
  co искать pattern в line;
    if (pattern есть в line)
      вывести line;
  // прочитать следующую строку ввода и записать ее в line;
  oc;
}
```

Отметим, что первая ветвь оператора co является последовательностью операторов. Но независимы ли эти два процесса программы? Ответ — нет, поскольку первый читает line, а другой записывает в нее. Поэтому, если второй процесс выполняется быстрее первого, он будет перезаписывать строку до того, как ее успеет проверить первый процесс.

Как было отмечено, части программы могут выполняться параллельно только в том случае, если они читают и записывают различные переменные. Предположим, что второй процесс записывает не в ту переменную, которую проверяет первый процесс, и рассмотрим следующую программу.

```
string line1, line2;
прочитать строку ввода из stdin в line1;
while (!EOF) {
  co искать pattern в line1;
    if (pattern есть в line1)
      вывести line1;
  // прочитать следующую строку ввода и записать ее в line2;
  oc;
}
```

Теперь эти два процесса работают с разными строками, записанными в переменные `line1` и `line2`. Следовательно, процессы могут выполняться параллельно. Но правильна ли эта программа? Ясно, что нет, поскольку первый процесс все время ищет в `line1`, тогда как второй постоянно записывает в `line2`, которая никогда не рассматривается.

Решение относительно простое: поменяем роли строк данных в конце каждого цикла, чтобы первый процесс всегда проверял последнюю прочитанную из файла строку, а второй процесс всегда записывал в другую переменную. Этому соответствует следующая программа.

```
string line1, line2;
прочитать строку ввода из stdin в line1;
while (!EOF) {
  co искать pattern в line1;
    if (pattern есть в line1)
      вывести line1;
  // прочитать следующую строку ввода в line2;
  oc;
  line1 = line2;
}
```

Здесь в конце каждого цикла и после завершения каждого процесса содержимое `line2` копируется в `line1`. Процессы внутри оператора `co` теперь независимы, но их действия связаны из-за последнего оператора цикла, который копирует `line2` в `line1`.

Параллельная программа, приведенная выше, верна, но совершенно неэффективна. Во-первых, в последней строке цикла содержимое переменной `line2` копируется в переменную `line1`. Это последовательное действие отсутствует в первой программе, и в общем случае оно требует копирования огромного количества символов, а это — накладные расходы. Во-вторых, в теле цикла содержится оператор `co`, а это означает, что при каждом повторении цикла `while` будут создаваться, выполняться и уничтожаться по два процесса. "Копирование" можно сделать намного эффективнее, использовав массив с двумя строками. Индексы в каждом из процессов должны указывать на различные строки массива, а последняя строка определяется просто обменом значений индексов. Однако и в этом случае создавать процессы весьма накладно, поскольку создание и уничтожение процесса занимает намного больше времени, чем вызов процедуры, и еще больше, чем выполнение линейного участка кода (за подробностями обращайтесь к главе 6).

Итак, мы подошли к последнему вопросу этого раздела. Существует ли еще один путь распараллеливания программы, позволяющий не использовать оператор `co` внутри цикла? Как вы наверняка уже догадались, ответ — да. В частности, вместо использования оператора `co` внутри цикла `while`, можно поместить циклы `while` в каждую ветвь оператора `co`. В листинге 2.1 показано решение, использующее этот метод. Данная программа является примером схемы типа "производитель-потребитель", представленной в разделе 1.6. Здесь первый процесс является производителем, а второй — потребителем. Они взаимодействуют с помощью разделяемой переменной `buffer`. Отметим, что объявления переменных `line1` и `line2` теперь стали локальными для процессов, поскольку строки уже не разделяются процессами.

Стиль программы, приведенной в листинге 2.1, называется "while внутри co", в отличие от стиля "co внутри while", использованного в предыдущих программах этого раздела. Преимущество стиля "while внутри co" состоит в том, что процессы создаются только однажды, а не при каждом повторении цикла. Недостатком является необходимость использовать два буфера и программировать синхронизацию. Операторы, предшествующие обращению к разделяемому буферу buffer и следующие за ним, указывают тип необходимой синхронизации. В разделе 2.5 будет показано, как программируется эта синхронизация, но сначала нужно подробно рассмотреть вопросы синхронизации в целом и неделимые действия в частности.

Листинг 2.1. Поиск шаблонов в файле

```
string buffer; # содержит одну строку ввода
bool done = false; #используется для сигнализации о завершении
co # процесс 1: найти шаблоны
  string line1;
  while (true) {
    ожидать заполнения буфера или значения true переменной done;
    if (done) break;
    line1 = buffer;
    сигнализировать, что буфер пуст;
    искать pattern в line1;
    if (pattern есть в line1)
      напечатать line1;
  }
// # процесс 2: прочитать новые строки
  string line2;
  while (true) {
    прочитать следующую строку ввода в line2;
    if (EOF) {done = true; break; }
    ожидать опустошения буфера
    buffer = line2;
    сигнализировать о заполнении буфера;
  }
oc;
```

2.3. Синхронизация: поиск максимального элемента массива

Рассмотрим другую задачу, которая требует синхронизации процессов. Она состоит в поиске максимального элемента массива a[n]. Предположим, что n положительно и все элементы массива — положительные целые числа.

Поиск максимального элемента в массиве a — это пример задачи накопления (сведения). В данном случае сохраняется (накапливается) максимальный из просмотренных элементов, или, иными словами, все значения сводятся к их максимуму. Пусть m — переменная для хранения максимального значения. Цель программы в логике предикатов можно описать так.

```
(∀ j: 1<= j <= n: m >= a[j]) ^
(∃ j: 1<= j <= n: m == a[j]).
```

Первая строка гласит, что при завершении программы значение переменной m должно быть не меньше любого значения в массиве a. Вторая строка означает, что переменная m должна быть равна *некоторому* значению в массиве a.

Для решения данной задачи можно использовать такую последовательную программу.

```
int m = 0;
for [i = 0 to n-1] {
  if (a[i] > m)
    m = a[i];
}
```

Эта программа итеративно просматривает все значения в массиве a; если находится какое-нибудь значение больше текущего максимума, оно присваивается m. Поскольку предполагается, что значения элементов массива положительны, можно без опасений инициализировать переменную m значением 0.

Теперь рассмотрим способы распараллеливания приведенной программы. Предположим, что цикл полностью распараллелен с помощью параллельной проверки всех элементов массива.

```
int m = 0;
co [i = 0 to n-1]
  if (a[i] > m)
    m = a[i];
```

Эта программа некорректна, поскольку процессы не являются независимыми: каждый из них и читает, и записывает переменную m. В частности, допустим, что все процессы выполняются с одинаковой скоростью, и, следовательно, каждый из них сравнивает свой элемент массива a[i] с переменной m в одно и то же время. Все процессы определят, что неравенство выполняется (поскольку все элементы массива a больше начального значения переменной m, равного нулю). Следовательно, все процессы попытаются обновить значение переменной m. Аппаратное обеспечение памяти будет выполнять обновление в порядке некоторой очереди, поскольку запись элемента в память — неделимая операция. Окончательным значением переменной m будет значение a[i], присвоенное ей последним процессом.

В программе, показанной выше, чтение и запись переменной m являются отдельными операциями. Для работы с таким уровнем параллельности можно использовать синхронизацию для совмещения отдельных действий в одной неделимой операции. Это делает следующая программа.

```
int m = 0;
co [i = 0 to n-1]
  ⟨if (a[i] > m)
    m = a[i];⟩
```

Угловые скобки в этом фрагменте кода указывают, что каждый оператор if выполняется как неделимая операция, т.е. проверяет текущее значение m и в соответствии с условием обновляет его в *одном, неделимом действии*. (Подробно нотация с угловыми скобками будет описана в следующем разделе.)

К сожалению, последняя программа — это почти то же самое, что и последовательная. В последовательной программе элементы массива a проверяются в установленном порядке — от a[0] до a[n-1]. В последней же программе элементы массива a проверяются в произвольном порядке, поскольку в произвольном порядке выполняются процессы. Но из-за синхронизации проверки все еще выполняются по одной.

Основные задачи в данном приложении — обеспечить, чтобы обновления переменной m были неделимыми операциями, а значение m было действительно максимальным. Допустим, что сравнения выполняются параллельно, но обновления производятся по одному, как в следующей программе.

```
int m = 0;
co [i = 0 to n-1]
  if (a[i] > m)
    ⟨m = a[i];⟩
```

Правильна ли эта версия? Нет, поскольку эта программа в действительности является тем же, что и первая параллельная программа: каждый процесс может сравнить свое значение элемента массива a с переменной m и затем обновить значение переменной m. И хотя здесь указано, что обновление m является неделимой операцией, фактически это осуществляется аппаратным обеспечением памяти.

Итак, как же лучше всего решить эту задачу? Выход состоит в сочетании последних двух программ. Можно безопасно выполнять параллельные сравнения, поскольку это действия, которые только читают переменные. Но необходимо обеспечить, чтобы при завершении программы значение m действительно было максимальным. Это достигается в следующей программе.

```
int m = 0;
co [i = 0 to n-1]
  if (a[i] > m)      #проверка значения m
    ⟨if (a[i] > m)   #перепроверка значения m
      m = a[i];⟩
```

Идея состоит в том, чтобы сначала проверить неравенство, а затем, если оно выполняется, провести *еще одну проверку* перед обновлением значения переменной. Она может показаться лишней, но это не так. Например, если некоторый процесс обновил значение m, то часть других процессов определит, что их значение a[i] меньше нового значения m, и не будет выполнять тело оператора if. После дальнейших обновлений еще меньше процессов определят, что условие в первой проверке истинно. Следовательно, если проверки сами по себе выполняются в некотором случайном порядке, а не параллельно, это повышает вероятность того, что процессам не нужно будет производить вторую проверку.

Эта частная задача — не из тех, которые выгодно решать с помощью параллельной программы, если только она не выполняется на SIMD-машине, построенной специально для эффективного выполнения мелкомодульных программ. Однако в данном разделе есть три ключевых момента. Во-первых, синхронизация необходима для получения правильных результатов, если процессы и считывают, и записывают разделяемые переменные. Во-вторых, неделимость действий задана угловыми скобками; это рассмотрено подробно в следующем разделе, а затем показано, как реализовать неделимые действия (фактически это пример критических секций). В-третьих, описанный здесь метод двойной проверки перед обновлением разделяемой переменной весьма полезен, как мы увидим в дальнейших примерах, особенно когда существует вероятность, что первая проверка дает ложный результат, и, следовательно, вторая проверка не нужна.

2.4. Неделимые действия и операторы ожидания

Как упоминалось выше, выполнение параллельной программы можно представить как чередование неделимых действий, выполняемых отдельными процессами. При взаимодействии процессов не все чередования допустимы. Цель синхронизации — предотвратить нежелательные чередования. Это осуществляется путем объединения мелкомодульных неделимых операций в крупномодульные (составные) действия или задержки выполнения процесса до достижения программой состояния, удовлетворяющего некоторому предикату. Первая форма синхронизации называется *взаимным исключением*; вторая — *условной синхронизацией*. В данном разделе рассмотрены аспекты неделимых действий и представлена нотация для определения синхронизации.

2.4.1. Мелкомодульная неделимость

Напомним, что неделимое действие выполняет неделимое преобразование состояния. Это значит, что любое промежуточное состояние, которое может возникнуть при выполнении этого действия, не должно быть видимым для других процессов. *Мелкомодульное* недели-

мое действие — это действие, реализуемое непосредственно аппаратным обеспечением, на котором выполняется программа.

В последовательной программе неделимыми оказываются операторы присваивания, поскольку при их выполнении нет промежуточных состояний, видимых программе (за исключением, возможно, случаев, когда происходит ошибка, определяемая аппаратным обеспечением). Однако в параллельных программах оператор присваивания не является неделимым действием, поскольку он может быть реализован в виде последовательности мелкомодульных машинных инструкций. В качестве примера рассмотрим следующую программу и предположим, что мелкомодульные неделимые действия — это чтение и запись переменных.

```
int y = 0, z = 0;
co x = y+z; // y = 1; z = 2; oc;
```

Если выражение `x = y+z` реализовано загрузкой значения `y` в регистр и последующим прибавлением значения `z`, то конечными значениями переменной `x` могут быть `0`, `1`, `2` или `3`. Это происходит потому, что мы можем получить как начальные значения `y` и `z`, так и их конечные значения или некоторую комбинацию, в зависимости от степени выполнения второго процесса. Еще одна особенность приведенной программы в том, что конечным значением `x` может быть и 2, хотя невозможно остановить программу и увидеть состояние, в котором сумма `y+z` имеет значение 2.

Предполагается, что машины обладают следующими характеристиками.

- Значения базовых типов (например `int`) хранятся в элементах памяти (например словах), которые считываются и записываются неделимыми операциями.
- Значения обрабатываются так: их помещают в регистры, там к ним применяют операции и затем записывают результаты обратно в память.
- Каждый процесс имеет собственный набор регистров. Это реализуется или путем предоставления каждому процессу отдельного набора регистров, или путем сохранения и восстановления значений регистров при выполнении различных процессов. (Это называется *переключением контекста*, поскольку регистры образуют контекст выполнения процесса.)
- Любые промежуточные результаты, появляющиеся при вычислении сложных выражений, сохраняются в регистрах или областях памяти, принадлежащих исполняемому процессу, например, в его стеке.

В этой модели машины, если выражение `e` в одном процессе не обращается к переменной, измененной другим процессом, вычисление выражения всегда будет неделимой операцией, даже если для этого необходимо выполнить несколько мелкомодульных действий. Это происходит потому, что при вычислении выражения `e` ни одно значение, от которого зависит `e`, не изменяет своего значения, и ни один процесс не может видеть временные значения, которые создаются при вычислении выражения. Аналогично, если присваивание `x = e` в одном процессе не ссылается на переменные, изменяемые другим процессом (например, ссылается только на локальные переменные), то выполнение присваивания будет неделимой операцией.

К сожалению, многие операторы в параллельных программах, ссылающиеся на разделяемые переменные, не удовлетворяют этим условиям непересекаемости. Однако часто выполняются более мягкие условия.

(2.2) **Условие "не больше одного".** *Критической ссылкой* в выражении называется ссылка на переменную, изменяемую другим процессом. Предположим, что любая критическая ссылка — это ссылка на простую переменную, которая хранится в элементе памяти и может быть считана и записана автоматически. Оператор присваивания `x = e` удовлетворяет условию "не больше одного", если либо выражение `e` содержит не больше одной критической ссылки, а переменная `x` не считывается другим процессом, либо выражение `e` не содержит критических ссылок, а другие процессы могут считывать переменную `x`.

Это условие называется "не больше одного", поскольку в таком случае возможна только одна разделяемая переменная, и на нее ссылаются не более одного раза. Аналогичное определение применяется к выражениям, которые не являются операторами присваивания. Такое выражение удовлетворяет условию "не больше одного", если содержит не более одной критической ссылки.

Если оператор присваивания удовлетворяет требованиям условия "не больше одного", то выполнение оператора присваивания будет *казаться* неделимой операцией, поскольку однаединственная разделяемая переменная в выражении будет записываться или считываться только один раз. Например, если выражение `e` не содержит критических ссылок, а переменная `x` — простая переменная, читаемая другими процессами, то они не могут распознать, вычисляется ли выражение неделимым образом. Аналогично, если `e` содержит одну критическую ссылку, то процесс, выполняющий присваивание, не сможет различить, каким образом изменяется значение переменной; он увидит только некоторое конечное значение.

Чтобы пояснить определение условия "не больше одного", приведем несколько примеров. В следующей программе оба присваивания удовлетворяют этому условию.

```
int x = 0, y = 0;
co x = x+1; // y = y+1; oc;
```

Здесь нет критических ссылок ни в один процесс, поэтому конечным значением и `x`, и `y` будет `1`.

Оба присваивания в следующей программе также удовлетворяют условию.

```
int x = 0, y = 0;
co x = y+1; // y = y+1; oc;
```

Первый процесс ссылается на `y` (одна критическая ссылка), но переменная `x` не читается вторым процессом, и во втором процессе нет критических ссылок. Конечным значением переменной `x` будет или `1`, или `2`, а конечным значением `y` — `1`. Первый процесс увидит переменную `y` или перед ее увеличением, или после, но в параллельной программе он никогда не знает, какое из значений он видит, поскольку порядок выполнения программы недетерминирован.

В следующем примере ни одно присваивание не соответствует требованию "не больше одного".

```
int x = 0, y = 0;
co x = y+1; // y = x+1; oc;
```

Выражение в каждом процессе содержит критическую ссылку, и каждый процесс присваивает значение переменной, считываемой другим процессом. Действительно, конечными значениями переменных `x` и `y` могут быть `1` и `2`, `2` и `1`, или даже `1` и `1` (если процессы считывают значения переменных `x` и `y` до присвоения им значений). Однако, поскольку каждое присваивание ссылается только один раз и только на одну переменную, изменяемую другим процессом, конечными будут те значения, которые действительно существовали в некотором состоянии. Это отличается от примера, приведенного ранее, в котором выражение `y+z` ссылалось на две переменные, изменяемые другим процессом.

2.4.2. Задание синхронизации: оператор ожидания

Возможно, выражение или оператор присваивания не удовлетворяет условию "не больше одного", однако необходимо выполнить его как неделимое. В более общем случае в одном неделимом действии необходимо выполнить последовательность операторов. В любом случае нужен механизм синхронизации, позволяющий задать *крупномодульное* неделимое действие, — последовательность мелкомодульных неделимых операций, которая выглядит как неделимая.

В качестве конкретного примера представим, что база данных содержит два значения `x` и `y`, которые всегда должны быть одинаковы в том смысле, что ни один процесс, использующий базу данных, не должен видеть состояния, в котором `x` и `y` различаются. Следовательно, если процесс изменяет `x`, он должен изменить и `y` в том же самом неделимом действии.

Еще один пример: пусть один процесс вставляет элементы в очередь, представленную связанным списком. Другой процесс удаляет элементы из списка при условии, что они там есть.

Одна переменная указывает на начало списка, а другая — на его конец. Вставка и удаление элементов требуют обработки двух значений. Например, для вставки элемента нужно изменить ссылку в предыдущем элементе и указатель конца списка так, чтобы они указывали на новый элемент. Если в списке содержится только один элемент, одновременные вставка и удаление могут вызвать конфликт и привести список к непригодному для использования состоянию. Таким образом, вставка и удаление элемента должны быть неделимыми действиями. К тому же, если список пуст, необходимо отложить выполнение операции удаления до того, как в список будет вставлен элемент.

Неделимые действия задаются с помощью угловых скобок ⟨ и ⟩. Например, ⟨e⟩ указывает, что выражение e должно быть вычислено неделимым образом.

Синхронизация определяется с помощью оператора `await`:

(2.3) `⟨await (B) S;⟩`

Булево выражение `B` задает *условие задержки* (delay condition), а `S` — это список последовательных операторов, завершение которых гарантированно (например, последовательность операторов присваивания). Оператор `await` заключен в угловые скобки для указания, что он выполняется как неделимое действие. В частности, выражение `B` гарантированно имеет значение "истина"[6], когда начинается выполнение `S`, и ни одно промежуточное состояние в `S` не видно другим процессам. Например, выполнение кода

```
⟨await (s > 0) s = s-1;⟩
```

откладывается до момента, когда значение `s` станет положительным, а затем оно уменьшается на 1. Гарантируется, что перед вычитанием значение `s` положительно.

Оператор `await` является очень мощным, поскольку может быть использован для определения любых крупномодульных неделимых действий. Это делает его удобным для выражения синхронизации, поэтому будем использовать оператор `await` для разработки первоначальных решений задач синхронизации. Вместе с тем, выразительная мощность оператора `await` делает очень дорогой его реализацию в общей форме. Однако, как показано в этой и нескольких последующих главах, существует множество частных случаев оператора `await`, допускающих эффективную реализацию. Например, последний приведенный выше оператор `await` является частным случаем операции `P` над семафором `s` (см. главу 4).

Общая форма оператора `await` определяет как взаимное исключение, так и синхронизацию по условию. Для определения только взаимного исключения можно использовать сокращенную форму оператора `await`:

```
⟨S;⟩
```

Например, в следующем операторе значения `x` и `y` увеличиваются в неделимом действии:

```
⟨x = x+1; y = y+1;⟩
```

Промежуточное состояние, в котором `x` была увеличена на единицу, а `y` — еще нет, по определению не будет видимым для других процессов, ссылающихся на переменные `x` или `y`. Если последовательность `S` — это одиночный оператор присваивания, удовлетворяющий условию (2.2) "не больше одного", или если последовательность `S` реализована одной машинной инструкцией, то `S` будет выполнена как неделимая; таким образом, ⟨`S;`⟩ имеет тот же эффект, что и `S`.

Для задания только условной синхронизации сократим оператор `await` так:

```
⟨await (B);⟩
```

Например, следующим оператором выполнение процесса откладывается до момента, когда значение переменной `count` станет положительным.

```
⟨await (count > 0);⟩
```

---

[6] Условие задержки лучше было бы называть условием *окончания задержки*. — *Прим. ред.*

Если выражение B удовлетворяет условию "не больше одного", как в данном примере, то выражение ⟨await (B);⟩ может быть реализовано как

```
while (not B);
```

Это пример так называемого *цикла ожидания* (spin loop). Тело оператора while пусто, поэтому он просто зацикливается до тех пор, когда значением B станет "ложь".

Безусловное неделимое действие — это действие, которое не содержит в теле условия задержки B. Такое действие может быть выполнено немедленно, конечно, в соответствии с требованием неделимости его выполнения. Аппаратно реализуемые (мелкомодульные) действия, выражения в угловых скобках и операторы await, в которых условие опущено или является константой "истина", являются безусловными неделимыми действиями.

Условное неделимое действие — это оператор await с условием B. Такое действие не может быть выполнено, пока условие B не станет истинным. Если B ложно, оно может стать истинным только в результате действий других процессов. Таким образом, процесс, ожидающий выполнения условного неделимого действия, может оказаться задержанным непредсказуемо долго.

2.5. Синхронизация типа "производитель-потребитель"

В последнем решении раздела 2.2 для задачи поиска шаблонов в файле использованы процесс-производитель и процесс-потребитель. Производитель постоянно считывает строки ввода, определяет, какие из них содержат искомый шаблон, и передает их процессу-потребителю. Затем потребитель выводит строки, полученные от производителя. Взаимодействие между производителем и потребителем обеспечивается с помощью разделяемой переменной buffer. Метод синхронизации доступа к буферу остался неопределенным, и теперь его можно описать.

Здесь решается более простая задача типа "производитель-потребитель": копирование всех элементов массива от производителя к потребителю. Адаптация этого решения к конкретной задаче из раздела 2.2 оставляется читателю (см. упражнения в конце этой главы).

Даны два процесса: Producer (производитель) и Consumer (потребитель). Процесс Producer имеет локальный массив целых чисел a[n]; Consumer — b[n]. Предполагается, что массив a инициализирован. Цель — скопировать его содержимое в массив b. Поскольку процессы не разделяют массивы, для их взаимодействия нужны разделяемые переменные. Пусть переменная buf — это одиночная разделяемая целочисленная переменная, которая будет служить буфером взаимодействия.

Процессы Producer и Consumer должны получать доступ к переменной buf по очереди. Сначала Producer помещает первый элемент массива a в переменную buf, затем Consumer извлекает ее, потом Producer помещает в переменную buf следующий элемент массива a и так далее. Пусть разделяемые переменные p и c будут счетчиками числа помещенных и извлеченных элементов, соответственно. Их начальные значения — 0. Тогда условия синхронизации процессов Producer и Consumer могут быть записаны в следующем виде:

```
PC:   c <= p <= c+1
```

Значения переменных c и p могут отличаться не больше, чем на 1; это значит, что Producer поместил в буфер максимум на один элемент больше, чем Consumer извлек. Код этих двух процессов приведен в листинге 2.2.

Процессы Producer и Consumer используют переменные p и c (см. листинг 2.2) для синхронизации доступа к буферу buf. Операторы await применяются для приостановки процессов до тех пор, пока буфер не станет полным или пустым. Если истинно условие p == c, буфер пуст (последний помещенный в него элемент был извлечен), а если p > c — заполнен.

Если синхронизация реализуется описанным способом, говорят, что процесс находится в *состоянии активного ожидания*, или *зациклен*, поскольку он занят проверкой условия в операторе await, но все, что он делает, — это повторение цикла до тех пор, пока условие не вы-

полнится. Это обычный тип синхронизации, необходимый на самых низких уровнях программных систем, например, в операционных системах и сетевых протоколах. Активное ожидание рассматривается подробно в главе 3.

Листинг 2.2. Копирование массива от производителя к потребителю

```
int buf, p = 0, c = 0;

process Producer {
  int a[n];
  while (p < n) {
    ⟨await (p == c);⟩
    buf = a[p];
    p = p+1;
  }
}

process Consumer {
  int b[n];
  while (c < n) {
    ⟨await (p > c);⟩
    b[c] = buf;
    c = c+1;
  }
}
```

2.6. Обзор аксиоматической семантики

В конце раздела 2.1 было описано, как утвердительные рассуждения позволяют понять свойства параллельной программы. Но, что еще важнее, они могут помочь в разработке правильных программ. Поэтому утвердительные рассуждения будут интенсивно использоваться в оставшейся части книги. В этом и следующих двух разделах представлена формальная база для них. В дальнейших главах эти понятия будут использоваться неформально.

Основой для утвердительных рассуждений является так называемая *логика программирования* — формальная логическая система, которая обеспечивает порождение точных утверждений о выполнении программы. В этом разделе представлены основные понятия данной темы. В исторической справке указаны источники более подробной информации, включающей много различных примеров.

2.6.1. Формальные логические системы

Любая формальная логическая система состоит из правил, определенных в терминах следующих множеств:

- *символов*;
- *формул*, построенных из этих символов;
- выделенных формул, называемых *аксиомами*;
- *правил вывода.*

Формулами являются правильно построенные последовательности символов. Аксиомы — это особые формулы, которые *априори* предполагаются истинными. Правила вывода определяют, как получить истинные формулы из аксиом и других истинных формул. Правила вывода имеют вид

$$\frac{H_1, H_2, \ldots, H_n}{C}.$$

Каждая H_i является гипотезой, C — заключением. Значение правила вывода состоит в том, что если все гипотезы истинны, то и заключение истинно. И гипотезы, и заключение являются формулами или их схематическим представлением.

Доказательство в формальной логической системе — это последовательность строк, каждая из которых является аксиомой или может быть получена из предыдущих строк путем применения к ним правила вывода. *Теорема* — это любая строка в доказательстве. Таким образом, теоремы либо являются аксиомами, либо получены с помощью применения правил вывода к другим теоремам.

Сама по себе формальная логическая система является математической абстракцией — набором символов и отношений между ними. Логическая система становится интересной, когда формулы представляют утверждения о некоторой обсуждаемой области, а формулы, являющиеся теоремами, — истинные утверждения. Для этого нужно обеспечить интерпретацию формулы.

Интерпретация логики отображает каждую формулу в ложь или истину. Логика является *непротиворечивой* относительно интерпретации, если непротиворечивы все ее аксиомы и правила вывода. Аксиома непротиворечива, если она отображается в истину. Правило вывода непротиворечиво, если его следствие отображается в истину, при условии, что все его гипотезы отображаются в истину. Следовательно, если логика непротиворечива, то все теоремы являются истинными утверждениями в данной рассматриваемой области. В таком случае интерпретация называется *моделью* логики.

Понятие *полноты* дуально понятию непротиворечивости. Логика является *полной* относительно интерпретации, если формула, отображаемая в истину, является теоремой, т.е. в данной логике доказуема любая (истинная) формула. Таким образом, если *ФАКТЫ* — это множество истинных утверждений, выразимых формулами логики, а *ТЕОРЕМЫ* — множество теорем логики, то непротиворечивость означает, что *ТЕОРЕМЫ* $\subseteq$ *ФАКТЫ*, а полнота — *ФАКТЫ* $\subseteq$ *ТЕОРЕМЫ*. Полная и непротиворечивая логика позволяет доказать все ее истинные утверждения.

Как доказал немецкий математик Курт Гедель в своей знаменитой теореме о неполноте, любая логика, которая включает арифметику, не может быть полной. Однако логика, которая расширяет другую логику, может быть *относительно полной*. Это значит, что она не добавляет неполноту к той, которая присуща расширяемой логике. К счастью, относительной полноты вполне достаточно для логики программирования, представленной здесь, поскольку используемые свойства арифметики безусловно истинны.

2.6.2. Логика программирования

Логика программирования (ЛП) является формальной логической системой, которая позволяет устанавливать и обосновывать (доказывать) свойства программ.

Как и в любой формальной логической системе, в *ЛП* есть символы, формулы, аксиомы и правила вывода. Символы *ЛП* — это предикаты, фигурные скобки и операторы языка программирования. Формулы в *ЛП* называются *тройками*. Они имеют вид[7]

```
{P} S {Q}.
```

Предикаты `P` и `Q` определяют отношения между значениями переменных программы; `S` — это оператор или список операторов.

---

[7] Предикаты в тройках заключаются в фигурные скобки, поскольку это традиционный способ их записи в логиках программироваиия. Однако такие скобки используются в нашей программной нотации и для выделения последовательностей операторов. Во избежание возможных недоразумений предикаты в программе будут записываться с помощью символов `##`. Напомним, что одиночный символ `#` используется для однострочного комментария, т.е. предикат можно рассматривать как предельно точный и четкий комментарий.

Цель логики программирования состоит в том, чтобы обеспечить возможность обосновывать (доказывать) свойства выполнения программы. Следовательно, *интерпретация* тройки характеризует отношение между предикатами {P} и {Q} и списком операторов S.

(2.4) **Интерпретация тройки.** Тройка {P} S {Q} истинна при условии, что если выполнение S начинается в состоянии, удовлетворяющем P, и завершается, то результирующее состояние удовлетворяет Q.

Эта интерпретация называется *частичной корректностью*, которая является свойством безопасности в соответствии с определением из раздела 2.1. Она утверждает, что если начальное состояние программы удовлетворяет P, то заключительное состояние будет удовлетворять Q при условии, что выполнение S завершится. Соответствующее свойство живучести — это *тотальная* (*полная*) *корректность*, т.е. частичная корректность плюс завершимость (все истории конечны).

Предикаты P и Q в тройке часто называются *утверждениями*, поскольку они утверждают, что состояние программы должно удовлетворять предикату, чтобы интерпретация тройки была истинной. Таким образом, утверждение характеризует допустимое состояние программы. Предикат P называется *предусловием* S. Он описывает условие, которому должно удовлетворять состояние перед началом выполнения S. Предикат Q называется *постусловием* S. Он описывает состояние после выполнения S при условии, что выполнение S завершается. Двумя особыми утверждениями являются true (истина), характеризующее все состояния программы, и false (ложь), которое не описывает ни одного состояния программы.

Для того чтобы интерпретация (2.4) была моделью нашей логики программирования, аксиомы и правила вывода *ЛП* должны быть непротиворечивыми относительно (2.4). Это гарантирует, что все доказуемые в *ЛП* теоремы непротиворечивы. Например, такая тройка должна быть теоремой:

```
{x == 0} x = x+1; {x == 1}
```

Однако следующая тройка не будет теоремой, поскольку присваивание значения переменной x не может чудесным образом изменить значение y на 1:

```
{x == 0} x = x+1; {y == 1}
```

В дополнение к непротиворечивости логика должна быть (относительно) полной, чтобы истинные тройки в действительности были доказуемыми теоремами.

Важнейшей аксиомой логики программирования, такой как *ЛП*, является аксиома, связанная с присваиванием.

Аксиома присваивания. $\{P_{x \leftarrow e}\}$ x = e {P}

Запись $P_{x \leftarrow e}$ определяет *текстуальную подстановку*: заменить все свободные вхождения переменной x в предикат P выражением e. (Переменная является свободной в предикате, если она не входит в область действия квантора существования или всеобщности, имеющего переменную с тем же именем.) Аксиома присваивания гласит, что если нужно, чтобы присваивание приводило к состоянию, удовлетворяющему предикату P, то предшествующее состояние должно удовлетворять предикату P, в котором вместо переменной x записано выражение e.

В качестве примера для этой аксиомы приведем следующую тройку:

```
{1 == 1} x = 1; {x == 1}
```

Предусловие упрощается до предиката true, характеризующего все состояния. Таким образом, эта тройка означает, что, независимо от начального состояния, после присваивания x значения 1 получается состояние, удовлетворяющее предикату x == 1.

Более общий способ рассматривать присваивание — "идти вперед", т.е. начать с некоторого предиката, характеризующего текущее состояние, а затем записать предикат, истинный для состояния после выполнения присваивания. Например, если начать в состоянии, в котором x == 0, и прибавить 1 к x, то в результирующем состоянии значением x будет 1. Это описывается тройкой

```
{x == 0} x = 1; {x == 1}.
```

Аксиома присваивания описывает изменение состояния. Правила вывода в такой логике программирования, как *ЛП*, позволяют сочетать теоремы, полученные из частных случаев аксиомы присваивания. В частности, правила вывода используются для описания действия композиции операторов (списков операторов) и управляющих операторов, например `if` и `while`. Они также позволяют изменять предикаты в тройках.

На рис. 2.1 представлены четыре наиболее важных правила вывода. *Правило композиции* позволяет склеить тройки для двух операторов, выполняемых один за другим. Первая гипотеза в *правиле оператора if* характеризует результат выполнения оператора `S`, когда условие `B` истинно; вторая описывает, что является истинным, когда `B` ложно; заключение объединяет эти два случая. В качестве простого примера использования этих правил рассмотрим программу, которая присваивает переменной `m` максимальное из значений `x` и `y`.

```
{true}
m = x;
{m == x}
if (y > m)
    m = y;
{(m == x ∧ m >= y) or (m == y) ∧ m > x)}
```

При любом начальном состоянии первое присваивание приводит к состоянию, когда `m == x`. После выполнения оператора `if` значение переменной `m` равно `x` и не меньше `y`, или оно равно `y` и больше, чем `x`.

Правило композиции. $\dfrac{\{P\}\ S_1\ \{Q\},\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$

Правило оператора if. $\dfrac{\{P \wedge B\}\ S\ \{Q\},\ \{P \wedge \neg B\} \Rightarrow Q}{\{P\}\ \mathit{if}\ (B)\ S;\ \{Q\}}$

Правило оператора while. $\dfrac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \mathit{while}\ (B)\ S;\ \{I \wedge \neg B\}}$

Правило следования. $\dfrac{\{P' \Rightarrow P\},\ \{P\}\ S\ \{Q\},\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$

Рис. 2.1. Правила вывода в логике программирования ЛП

Для *правила оператора while* требуется *инвариант цикла* `I`. Это предикат, значение которого истинно перед каждым повторением цикла и после него. Если инвариант `I` и условие цикла `B` истинны перед выполнением тела цикла `S`, то выполнение `S` должно снова сделать `I` истинным. Таким образом, когда оператор цикла завершается, инвариант `I` остается истинным, а `B` становится ложным. В качестве примера приведем следующую программу, которая просматривает массив `a` и ищет в нем первое вхождение значения `x`. При условии, что `x` встречается в `a`, цикл завершается присвоением переменной `i` индекса первого вхождения.

```
i = 1;
{i == 1 ∧ (∀ j: 1 <= j < i: a[j] != x)}
while (a[i] != x)
  i = i+1;
{(∀ j: 1 <= j < i: a[j] !=x) ∧ a[i] == x}
```

Здесь инвариантом цикла является предикат с квантором. Он истинен перед выполнением цикла, поскольку область определения квантора пуста. Он также истинен перед каждым вы-

полнением тела цикла и после него. Когда выполнение оператора цикла завершается, `a[i]` равно `x`, и `x` ранее не встречался в массиве `a`.

Правило следования позволяет усиливать предусловия и ослаблять постусловия. В качестве примера рассмотрим истинную тройку:

```
{x == 0} x = x+1; {x == 1}.
```

По правилу следования истинной будет и такая тройка:

```
{x == 0} x = x+1; {x > 0}.
```

Во второй тройке постусловие *слабее*, чем в первой, поскольку оно характеризует больше состояний; значение `x` может как равняться 1, так и вообще быть неотрицательным.

2.6.3. Семантика параллельного выполнения

Оператор параллельного выполнения `co` (или объявление процесса) является управляющим оператором. Следовательно, его действие описывается правилом вывода, учитывающим воздействие параллельного выполнения. Процессы состоят из последовательных операторов и операторов синхронизации, таких как оператор `await`.

С точки зрения частичной корректности действие оператора `await`

```
⟨await (B) S;⟩
```

больше всего похоже на действие оператора `if`, для которого условие `B` истинно, когда начинается выполнение `S`. Следовательно, правило вывода для оператора `await` аналогично правилу вывода для оператора `if`.

Правило оператора await. $\dfrac{\{P \wedge B\}\ S\ \{Q\}}{\{P\} <await\ (B)\ S;>\ \{Q\}}$

Гипотеза гласит: "если выполнение `S` начинается в состоянии, когда и `P`, и `B` истинны, и `S` завершается, то `Q` будет истинным". Заключение позволяет сделать вывод о том, что оператор `await` приводит к состоянию, удовлетворяющему `Q`, если начинается в состоянии, удовлетворяющем `P` (при условии, что выполнение оператора `await` завершается). Правила вывода ничего не говорят о возможных задержках выполнения, поскольку задержки влияют на свойства живучести, а не на свойства безопасности.

Теперь рассмотрим влияние параллельного выполнения, заданного, например, таким оператором:

```
co S1; // S2; // ... // Sn; oc;
```

Предположим, что для каждого оператора истинно следующее выражение:

```
{Pi} Si {Qi}
```

В соответствии с интерпретацией троек (2.4) это означает, что если `Si` начинается в состоянии, удовлетворяющем `Pi`, и `Si` завершается, то состояние удовлетворяет `Qi`. Для того чтобы эта интерпретация оставалась верной при параллельном выполнении, процессы должны начинаться в состоянии, удовлетворяющем конъюнкции предикатов `Pi`. Если все процессы завершаются, заключительное состояние будет удовлетворять конъюнкции предикатов `Qi`. Таким образом, получается следующее правило вывода.

Правило оператора co. $\dfrac{\{P_i\}\ S_i\ \{Q_i\}\ \textit{свободны от взаимного вмешательства}}{\begin{array}{l}\{P_1 \wedge \ldots \wedge P_n\} \\ co\ S_1; // \ldots // S_n; oc \\ \{Q_1 \wedge \ldots \wedge Q_n\}\end{array}}$

Отметим фразу в гипотезе. Чтобы заключение было истинным, процессы и их обоснования не должны влиять друг на друга.

Один процесс *вмешивается* в другой (*влияет* на другой), если он выполняет присваивание, нарушающее утверждение в другом процессе. Утверждения характеризуют, *что* в процессе предполагается истинным до и после выполнения каждого оператора. Таким образом, если один процесс присваивает значение разделяемой переменной и тем самым делает недействительным предположение другого процесса, то обоснование другого процесса становится ложным.

В качестве примера рассмотрим следующую простую программу:

```
{x == 0}
co ⟨x = x+1;⟩ // ⟨x = x+2;⟩ oc
{x == 3}
```

Если выполнение программы начинается состоянием, в котором значение x равно 0, то при завершении программы значение x будет равно 3. Но что же будет истинным для каждого процесса? Нельзя предполагать, что значение x будет 0 в начале каждого из них, поскольку порядок выполнения операторов недетерминирован. В частности, на предположение, что в начале выполнения процесса x имеет значение 0, влияет другой процесс, если выполняется первым. Однако то, что является истинным, учитывается следующими утверждениями.

```
{x == 0}
co {x == 0 ∨ x == 2}
   ⟨x = x+1;⟩
   {x == 1 ∨ x == 3}
// {x == 0 ∨ x == 1}
   ⟨x = x+2;⟩
   {x == 2 ∨ x == 3}
oc
{x == 3}
```

Утверждения в каждом процессе учитывают оба возможных порядка выполнения. Отметим также, что конъюнкция предусловий — это действительно x == 0, а конъюнкция постусловий — x == 3.

Выше показан пример так называемой *схемы доказательства*. Перед каждым оператором и после него приведены утверждения, и все полученные тройки истинны. (В данной схеме три тройки: по одной для каждого процесса и одна для оператора co.) Следовательно, схема доказательства учитывает все ключевые части, необходимые для формального доказательства правильности данной программы.

Вернемся к формальному определению невмешательства. *Действие присваивания* — это оператор присваивания или оператор await, содержащий одно или несколько присваиваний. *Критическое утверждение* является предусловием или постусловием вне оператора await.

(2.5) **Взаимное невмешательство.** Пусть a — действие присваивания в некотором процессе, и его предусловием является pre(a). Пусть C — критическое утверждение в другом процессе. Если необходимо, переименуем локальные переменные в процессе C так, чтобы их имена отличались от имен локальных переменных в действии a и предусловии pre(a). Тогда действие a *не вмешивается* в (*не влияет* на) C, если следующее утверждение является теоремой логики программирования *ЛП*:

```
{C ∧ pre(a)} a {C}
```

Таким образом, критическое утверждение C является инвариантным относительно действия присваивания a. Предусловие a включено в (2.5), поскольку действие a может быть выполнено, если только процесс находится в состоянии, удовлетворяющем условию pre(a).

В качестве примера использования (2.5) рассмотрим последнюю из приведенных выше программ. Предусловие первого процесса является критическим утверждением. На него не влияет оператор присваивания во втором процессе, поскольку истинна такая тройка:

```
{(x == 0 ∨ x == 2) ∧ (x == 0 ∨ x == 1)}
x = x+2;
{x == 0 ∨ x == 2}
```

Первый предикат упрощается до `x == 0`, поэтому после прибавления 2 к `x` значение переменной `x` будет или `0`, или `2`. Эта тройка выражает следующий факт: если второй процесс выполняется перед первым, то в начале выполнения первого процесса переменная `x` будет иметь значение `2`. В данной программе есть еще три критических утверждения: постусловие первого процесса, пред- и постусловие второго процесса. Все доказательства взаимного невмешательства аналогичны приведенному.

2.7. Техника устранения взаимного вмешательства

Процессы в параллельной программе работают вместе над вычислением результата. Ключевое требование для получения правильной программы состоит в том, что процессы не должны влиять друг на друга. Совокупность процессов свободна от взаимного влияния, когда в одном процессе нет действий присваивания, вмешивающихся в критические утверждения другого процесса.

В данном разделе описаны четыре основных метода устранения взаимного вмешательства, которые можно использовать для разработки правильных параллельных программ: непересекающиеся множества переменных, ослабленные утверждения, глобальные инварианты и синхронизация. Эти методы широко применяются в оставшейся части книги. Все они включают запись утверждений и действий присваивания в форме, обеспечивающей истинность формулам (2.5).

2.7.1. Непересекающиеся множества переменных

Напомним, что *множество записи* процесса — это множество переменных, которым он присваивает значения (и, возможно, считывает их), а *множество чтения* процесса — это множество переменных, которые процесс считывает, но не изменяет. *Множество ссылок* процесса — это множество переменных, которые встречаются в утверждениях доказательства корректности данного процесса. Множество ссылок процесса часто (но не всегда) будет совпадать с объединением множеств чтения и записи. По отношению к взаимному вмешательству критические переменные — это переменные в утверждениях.

Если множество записи одного процесса не пересекается со множеством ссылок другого процесса и наоборот, то эти два процесса не могут влиять друг на друга. Формально это происходит потому, что в аксиоме присваивания используется текстуальная подстановка, которая не влияет на предикат, не содержащий ссылок на целевую переменную присваивания. (Локальные переменные в различных процессах, даже если и имеют одинаковые имена, все равно являются разными переменными; поэтому перед применением аксиомы присваивания их можно переименовать.)

В качестве примера рассмотрим следующую программу.

```
co x = x+1; // y = y+1; oc
```

Если начальные значения `x` и `y` равны `0`, то по аксиоме присваивания теоремами являются следующие утверждения:

```
{x == 0} x = x+1; {x == 1}
{y == 0} y = y+1; {y == 1}
```

Каждый процесс содержит один оператор присваивания и два утверждения; следовательно, нужно доказать четыре теоремы взаимного невмешательства. Каждая из них тривиально ис-

тинна, поскольку два процесса ссылаются на разные переменные, поэтому подстановки, связанные с аксиомой присваивания, будут пустыми.

Непересекающиеся множества записи и ссылок образуют основу для большинства параллельных итерационных алгоритмов, таких как алгоритм умножения матриц, описанный в разделе 1.4. Еще два примера: разные ветви дерева возможных ходов в игровой программе могут исследоваться параллельно, а множественные транзакции могут параллельно просматривать базу данных или обновлять разные отношения.

2.7.2. Ослабленные утверждения

Чтобы учесть воздействие параллельного выполнения, взаимного вмешательства можно иногда избежать с помощью ослабленных утверждений, даже когда множества записи и ссылок процессов перекрываются. *Ослабленное утверждение* допускает больше состояний программы, чем другое утверждение, которое могло бы быть истинным для изолированного процесса. В разделе 2.6 был приведен пример с такой программой.

```
{x == 0}
co {x == 0 ∨ x == 2}
   ⟨x = x+1;⟩
   {x == 1 ∨ x == 3}
// {x == 0 ∨ x == 1}
   ⟨x = x+2;⟩
   {x == 2 ∨ x == 3}
oc
{x == 3}
```

Здесь пред- и постусловия для каждого процесса слабее, чем могли бы быть при изолированном выполнении. В частности, для каждого изолированного процесса можно утверждать, что если x сначала имеет значение 0, то при завершении ее значение становится 1 (для первого процесса) или 2 (для второго). Однако в эти более сильные утверждения произойдет вмешательство.

Ослабленные утверждения имеют более реалистичные приложения, чем сверхупрощенная задача, рассмотренная выше. Например, допустим, что процесс планирует операции диска с перемещаемыми головками. Другие процессы добавляют операции в очередь. Когда диск незанят, планировщик проверяет очередь, выбирает по некоторому критерию наилучшую операцию и начинает ее выполнять. В действительности диск не всегда будет выполнять наилучшую операцию, даже если она была таковой во время просмотра очереди. Это происходит потому, что процесс может добавить еще одну, лучшую, операцию в очередь сразу после того, как был сделан выбор, и даже до того, как диск начал выполнение операции. Таким образом, "наилучшая" в данном случае — это свойство, зависящее от времени, хотя для задачи планирования вроде приведенной этого достаточно.

Еще один пример — многие параллельные алгоритмы приближенного решения дифференциальных уравнений имеют следующий вид. (Более конкретный пример приводится в главе 11.) Пространство задачи аппроксимируется конечной сеткой точек, скажем, grid[n,n]. Каждой точке или (что типичнее) блоку точек сетки назначается процесс, как в следующем фрагменте программы.

```
double grid[n,n];

process PDE [i = 0 to n-1, j = 0 to n-1] {
  while (не сошлось) {
    grid[i,j] = f(окружающие точки);
  }
]
```

Функция f, вычисляемая на каждой итерации, может быть, например, усреднением значений в четырех соседних точках той же строки и того же столбца. Во многих задачах значение, присваиваемое элементу grid[i,j] на одной итерации, зависит от значений соседних элементов на предыдущей. Таким образом, инвариант цикла должен характеризовать отношения между старыми и новыми значениями точек сетки.

Чтобы гарантировать, что инвариант цикла в каждом процессе не подвержен влиянию извне, процессы должны использовать две матрицы и после каждой итерации синхронизироваться. На каждой итерации каждый процесс PDE считывает значения из матрицы, вычисляет f, а затем записывает результат во вторую матрицу. Потом он ждет, пока остальные процессы вычислят новые значения для своих точек сетки. (В следующей главе показано, как реализовать этот тип синхронизации, называемый *барьером*.) Затем роли матриц меняются местами, и процессы выполняют следующую итерацию.

Второй способ синхронизации процессов состоит в их пошаговом выполнении, когда все процессы одновременно выполняют одни и те же действия. Этот тип синхронизации поддерживается в синхронных процессорах. Данный метод позволяет избежать взаимного вмешательства, поскольку каждый процесс считывает старые значения из массива grid до того, как какой-либо из процессов присвоит новое значение.

2.7.3. Глобальные инварианты

Еще один, очень эффективный способ избежать взаимного вмешательства состоит в использовании глобального инварианта для учета всех отношений между разделяемыми переменными. Как показано в начале главы 3, глобальный инвариант можно использовать при разработке решения *любой* задачи синхронизации.

Предположим, что I — предикат, который ссылается на глобальные переменные. Тогда I называется *глобальным инвариантом* по отношению ко множеству процессов, если: 1) I истинен в начале выполнения процессов, 2) I сохраняется каждым действием присваивания. Условие 1 удовлетворяется, если предикат I истинен в начальном состоянии каждого процесса. Условие 2 удовлетворяется, если для любого действия присваивания a предикат I истинен после выполнения a при условии, что I был истинным до выполнения a. Таким образом, эти два условия образуют пример использования математической индукции.

Предположим, что предикат I является глобальным инвариантом. Допустим, что любое критическое утверждение C в обосновании каждого процесса P_j имеет вид $I \wedge L$, где L — предикат относительно локальных переменных. В частности, каждая переменная, на которую ссылается предикат L, является либо локальной переменной для процесса P_j, либо глобальной, которой присваивает только процесс P_j. Если все критические утверждения можно представить в форме $I \wedge L$, то доказательства правильности процессов будут свободны от взаимного вмешательства. Дело в том, что: 1) предикат I инвариантен относительно каждого действия присваивания a, и 2) ни одно действие присваивания в одном процессе не может повлиять на локальный предикат L в другом процессе, поскольку левая часть присваивания a отличается от переменных в предикате L.

Если все утверждения используют глобальный инвариант и локальный предикат так, как описано выше, требование взаимного невмешательства (2.5) выполняется для каждой пары действий присваивания и критических утверждений. Кроме того, нам нужно проверить только тройки в каждом процессе, чтобы удостовериться, что каждое критическое утверждение имеет вышеописанный вид, а предикат I является глобальным инвариантом. Нам даже нет необходимости просматривать утверждения или операторы в других процессах. Фактически из массива идентичных процессов достаточно проверить только один. В любом случае нужно проверить лишь *линейное* число операторов и утверждений. Сравните это с необходимостью проверки *экспоненциального* количества историй программы (см. раздел 2.1).

Техника глобальных инвариантов широко используется в остальной части книги. Ее полезность и мощность мы продемонстрируем в конце данного раздела после знакомства с четвертым способом избежать взаимного вмешательства — синхронизацией.

2.7.4. Синхронизация

Последовательность операторов внутри оператора `await` для других процессов выступает как неделимая единица. Это значит, что, определяя, вмешиваются ли процессы друг в друга, можно не обращать внимание на результаты выполнения отдельных операторов. Достаточно выяснить, может ли *вся* последовательность вызывать вмешательство. Например, рассмотрим следующее неделимое действие:

```
⟨x = x+1; y = y+1;⟩
```

Ни одно присваивание само по себе не может вызвать вмешательства, поскольку никакой другой процесс не может увидеть состояния, в котором переменная `x` уже была увеличена на единицу, а `y` — еще нет. Причиной вмешательства может стать только пара присваиваний.

Внутренние состояния сегментов программы внутри угловых скобок также неделимы. Следовательно, ни одно утверждение о внутреннем состоянии программы не может подвергаться вмешательству со стороны другого процесса. Например, утверждение в середине следующего неделимого действия не является критическим.

```
{x == 0 ∧ y == 0}
⟨x = x+1; {x == 1 ∧ y == 0} y = y+1;⟩
{x == 1 ∧ y == 1}
```

Эти два свойства неделимых действий приводят к двум способам использования синхронизации для устранения взаимного влияния: взаимному исключению и условной синхронизации. Рассмотрим следующий фрагмент.

```
co P1: ... a; ...
// P2: ... S1; {C} S2; ...
oc
```

Здесь `a` — это оператор присваивания в процессе `P1`, а `S1` и `S2` — операторы в процессе `P2`. Критическое утверждение `C` является предусловием оператора `S2`.

Предположим, что `a` влияет на `C`. Один способ избавиться от этого — использовать взаимное исключение, чтобы "скрыть" утверждение `C` от оператора `a`. Для этого операторы `S1` и `S2` второго процесса можно собрать в единое неделимое действие.

```
⟨S1; S2;⟩
```

Теперь операторы `S1` и `S2` выполняются неделимым образом, и состояние `C` становится невидимым для других процессов.

Другой способ избежать взаимного влияния процессов — использовать условную синхронизацию, чтобы усилить предусловие оператора `a`. В частности, можно заменить `a` следующим условным неделимым действием.

```
⟨await (!C or B) a;⟩
```

Здесь `B` — предикат, характеризующий множество состояний, при которых выполнение `a` сделает `C` истинным. Таким образом, в данном выражении вмешательство устраняется либо ожиданием того, что `C` станет ложным (тогда оператор `S2` не сможет выполняться), либо обеспечением того, что выполнение `a` сделает `C` истинным (что будет приемлемым для выполнения `S2`).

2.7.5. Пример: еще раз о задаче копирования массива

В большинстве параллельных программ используется сочетание представленных выше методов. Здесь *все* они иллюстрируются в одной простой программе: задаче копирования массива

(см. листинг 2.2). Напомним, что эта программа использует разделяемый буфер buf для копирования содержимого массива a (в процессе производителя) в массив b (в процессе потребителя).

Процессы Producer и Consumer в листинге 2.2 по очереди получают доступ к переменной buf. Сначала Producer помещает первый элемент массива a в переменную buf, затем Consumer извлекает ее, а Producer помещает в переменную buf следующий элемент массива a и так далее. В переменных p и c ведется подсчет числа помещенных и извлеченных элементов, соответственно. Для синхронизации доступа к переменной buf используются операторы await. Когда выполняется условие p == c, буфер пуст (последний помещенный в него элемент был извлечен). Когда выполняется условие p > c, буфер заполнен.

Допустим, что вначале элементы массива a[n] содержат некоторый набор значений A[n]. (A[i] — это просто заменители произвольных действительных значений.) Цель — доказать, что при условии завершения программы, представленной выше, значения элементов b[n] будут совпадать с A[n], т.е. значениями элементов массива a. Это утверждение можно доказать с помощью следующего глобального инварианта.

$$PC\text{: } c <= p <= c+1 \wedge a[0:n-1] == A[0:n-1] \wedge (p == c+1) \Rightarrow (buf == A[p-1])$$

Процессы чередуют доступ к переменной buf, поэтому в любой момент значение p равно c или на 1 больше, чем c. Массив a не изменяется, так что значением a[i] всегда является A[i]. Наконец, когда буфер полон (т.е. p == c+1), он содержит A[p-1].

Предикат *PC* в начальном состоянии является истинным, поскольку переменные p и c равны нулю. Его истинность сохраняется каждым оператором присваивания, что показано в схеме доказательства (листинг 2.3). В листинге *IP* является инвариантом цикла в процессе Producer, а *IC* — в Consumer. Как показано, предикаты *IP* и *IC* связаны с предикатом *PC*.

Листинг 2.3 — это еще один пример схемы доказательства, поскольку перед каждым оператором и после него находятся утверждения, и каждая тройка в схеме доказательства истинна. Тройки в каждом процессе образуются непосредственно вокруг операторов присваивания в каждом процессе. Если каждый процесс непрерывно получает возможность выполняться, то операторы await завершаются, поскольку сначала истинно одно условие задержки, затем другое и так далее. Таким образом, каждый процесс завершается после n итераций. Когда программа завершается, постусловия обоих процессов истинны. Следовательно, заключительное состояние программы удовлетворяет предикату:

$$PC \wedge p == n \wedge IC \wedge c == n$$

Таким образом, массив b содержит копию массива a.

Утверждения в двух указанных процессах не оказывают взаимного влияния. Большинство из них являются комбинациями глобального инварианта *PC* и локальных предикатов. Следовательно, они соответствуют требованиям взаимного невмешательства, описанным в начале раздела о глобальных инвариантах. Четырьмя исключениями являются утверждения, которые определяют отношения между значениями разделяемых переменных p и c. Но они не подвержены влиянию из-за операторов await.

Роль операторов await в программе копирования массива — обеспечить чередование доступа к буферу процесса-потребителя и процесса-производителя. В устранении взаимного вмешательства чередование играет двоякую роль. Во-первых, оно гарантирует, что два процесса не получат доступ к переменной buf одновременно (это пример взаимного исключения). Во-вторых, оно гарантирует, что производитель не перезаписывает элементы (переполнение), а потребитель не читает их дважды ("антипереполнение") — это пример условной синхронизации.

Подводя итоги, отметим, что этот пример, несмотря на простоту, иллюстрирует все четыре способа избежать взаимного влияния. Во-первых, многие операторы и многие части утверждений в каждом процессе не пересекаются. Во-вторых, использованы ослабленные утверждения о разделяемых переменных; например, говорится, что условие buf == A[p-1]

истинно, только если истинно p == c+1. В-третьих, для выражения отношения между значениями разделяемых переменных использован глобальный инвариант *PC*; хотя при выполнении программы изменяется каждая переменная, это отношение не меняется! И, наконец, чтобы обеспечить взаимное исключение и условную синхронизацию, необходимые в этой программе, использована синхронизация, выраженная операторами await.

Листинг 2.3. Схема доказательства для программы копирования массива

```
int buf, p = 0, c = 0;
{PC: c <= p <= c+1 ∧ a[0:n-1] == A[0:n-1] ∧
          (p == c+1) ⇒ (buf == A[p-1]}

process Producer {
  int a[n]; # предполагается, что a[i] инициализирован A[i]
  {IP: PC ∧ p <= n}
  while (p < n) {
    {PC ∧ p < n}
    ⟨await (p == c);⟩ # пауза до опустошения буфера
    {PC ∧ p < n ∧ p == c}
    buf = a[p];
    {PC ∧ p < n ∧ p == c ∧ buf == A[p]}
    p = p+1;
    {IP}
  }
  {PC ∧ p == n}
}

process Consumer {
  int b[n];
{IC: PC ∧ c <= n ∧ b[0:c-1] == A[0:c-1]}
  while (c < n) {
    {IC ∧ c < n}
    ⟨await (p > c);⟩  # пауза до заполнения буфера
    {IC ∧ c < n ∧ p > c}
    b[c] = buf;
    {IC ∧ c < n ∧ p > c ∧ b[c] == A[c]}
    c = c+1;
    {IC}
  }
  {IC ∧ c == n}
}
```

2.8. Свойства безопасности и живучести

Свойство программы — это ее атрибут, истинный для любой возможной истории выполнения (см. раздел 2.1). Каждое интересующее нас свойство можно сформулировать как свойство безопасности или живучести. Свойство безопасности утверждает, что во время выполнения программы не происходит ничего плохого; свойство живучести утверждает, что в конечном счете происходит что-то хорошее. В последовательных программах основное свойство безопасности состоит в корректности заключительного состояния, а живучести — в завершимости про-

граммы. Для параллельных программ эти свойства одинаково важны. Однако есть и другие интересные свойства безопасности и живучести, применимые к параллельным программам.

Два важных свойства безопасности параллельных программ — взаимные исключения и отсутствие взаимоблокировок (deadlock). Для взаимного исключения плохо, когда несколько процессов одновременно выполняют критические секции операторов. Для взаимоблокировки плохо, когда все процессы ожидают условий, которые никогда не выполнятся.

Вот примеры свойств живучести параллельных программ: процесс в конце концов войдет в критическую секцию, запрос на обработку будет принят, а сообщение достигнет места назначения. Свойства живучести зависят от стратегии планирования, которая определяет, какое из неделимых действий должно быть выполнено следующим.

В данном разделе представлены два метода обоснования свойств безопасности. Затем описаны различные типы стратегий планирования процессора и их влияние на свойства живучести.

2.8.1. Доказательство свойств безопасности

Каждое выполняемое программой действие основано на ее состоянии. Если программа не удовлетворяет свойству безопасности, должно существовать некоторое "плохое" состояние, не удовлетворяющее этому свойству. Например, если не удовлетворяется свойство взаимного исключения, то должно быть некоторое состояние, в котором два (или более) процесса одновременно находятся в своих критических секциях. Или если процессы заблокировали друг друга (вошли в клинч), то должно быть состояние, в котором это происходит.

Эти замечания ведут к простому методу доказательства, что программа удовлетворяет свойству безопасности. Пусть предикат *BAD* описывает плохое состояние программы. Тогда программа удовлетворяет соответствующему свойству безопасности, если *BAD* ложен в каждом состоянии любой возможной истории выполнения программы. Чтобы по данной программе S показать, что предикат *BAD* не является истинным в любом состоянии, нужно доказать, что он ложен в начальном состоянии, во втором состоянии и т.д., причем состояние изменяется в результате выполнения неделимых действий.

Если программа не должна попадать в *BAD*-состояние, она всегда должна быть в *GOOD*-состоянии, где предикат *GOOD* эквивалентен $\neg BAD$. Следовательно, эффективный способ обеспечить свойства безопасности — определить предикат *BAD* и его отрицание, чтобы получить *GOOD*, а затем проверить, является ли предикат *GOOD* глобальным инвариантом, т.е. предикатом, истинным в каждом состоянии программы. Чтобы сделать предикат глобальным инвариантом, можно использовать синхронизацию (как это уже было и еще не раз будет показано в дальнейших главах).

Вышеописанный метод является общим для доказательства свойств безопасности. Существует связанный с ним, но несколько более специализированный метод, также весьма полезный. Рассмотрим следующий фрагмент программы.

```
co # процесс 1
   ...; {pre(S1)} S1; ...
// # процесс 2
...; {pre(S2)} S2; ...
oc
```

Здесь есть два оператора, по одному в каждом процессе, и два связанных с ними предусловия (предикаты, истинные перед выполнением каждого оператора). Предположим, что эти предикаты не подвержены вмешательству, а их конъюнкция ложна:

```
pre(S1) ∧ pre(S2) == false
```

Это значит, что оба процесса одновременно не могут выполнять данные операторы, поскольку предикат, который ложен, не характеризует *ни одного* состояния программы (пустое множество состояний, если хотите). Этот метод называется *исключением конфигураций*, поскольку

он исключает конфигурации программы, в которых первый процесс находится в состоянии pre(S1), а второй в это же время — в состоянии pre(S2).

В качестве примера рассмотрим схему доказательства корректности программы копирования массива (см. листинг 2.3). Оператор await в каждом процессе может стать причиной задержки. Процессы заблокируют друг друга ("войдут в клинч"), если оба будут приостановлены и ни один не сможет продолжить работу. Процесс Producer приостанавливается, если он находится в операторе await, а условие (окончания) задержки ложно; в этом состоянии истинным является следующий предикат.

$$PC \wedge p < n \wedge p \mathrel{!}= c$$

Следовательно, когда приостановлен процесс Producer, истинно условие p > c. Аналогично приостанавливается процесс Consumer, если он находится в операторе await, а условие (окончания) задержки ложно; это состояние удовлетворяет такому предикату.

$$IC \wedge c < n \wedge p <= c$$

Поскольку условия p > c и p <= c не могут быть истинными одновременно, процессы не могут одновременно находиться в этих состояниях, т.е. взаимоблокировка возникнуть не может.

2.8.2. Стратегии планирования и справедливость

Большинство свойств живучести зависит от *справедливости* (*fairness*), связанной с гарантиями, что каждый процесс получает шанс на продолжение выполнения независимо от действий других процессов. Каждый процесс выполняет последовательность неделимых действий. Неделимое действие в процессе называется *подходящим*, или *возможным* (*eligible*), если оно является следующим неделимым действием в процессе, которое может быть выполнено. При наличии нескольких процессов существует несколько возможных неделимых действий. *Стратегия планирования* (scheduling policy) определяет, какое из них будет выполнено следующим. В этом разделе определены три степени справедливости, которые могут быть обеспечены стратегией планирования.

Напомним, что безусловное неделимое действие — это действие, не имеющее условия задержки. Рассмотрим следующую простую программу, в которой процессы выполняют безусловные неделимые действия.

```
bool continue = true;
co while (continue);
// continue = false;
oc
```

Допустим, что стратегия планирования назначает процессор для процесса до тех пор, пока тот не завершится или не будет приостановлен (задержан). При одном процессоре данная программа не завершится, если вначале будет выполняться первый процесс. Однако, если второй процесс в конце концов получит право на выполнение, программа будет завершена. Данная ситуация отражена в следующем определении.

(2.6) **Безусловная справедливость** (unconditional fairness). Стратегия планирования *безусловно справедлива*, если любое подходящее безусловное неделимое действие в конце концов выполняется.

Для программы, приведенной выше, безусловно справедливой стратегией планирования на одном процессоре было бы циклическое (round-robin) выполнение, а на мультипроцессоре — синхронное.

Если программа содержит условные неделимые действия — операторы await с логическими условиями B, необходимо делать более сильные предположения, чтобы гарантировать продвижение процесса. Причина в том, что условное неделимое действие не может быть выполнено, пока условие B не станет истинным.

(2.7) **Справедливость в слабом смысле** (weak fairness). Стратегия планирования *справедлива в слабом смысле*, если: 1) она безусловно справедлива, и 2) каждое подходящее условное неделимое действие в конце концов выполняется, если его условие становится и затем остается истинным, пока его видит процесс, выполняющий условное неделимое действие.

Таким образом, если действие ⟨await (B) S;⟩ является подходящим и условие B становится истинным, то B остается истинным по крайней мере до окончания выполнения условного неделимого действия. Циклическая стратегия и стратегия квантования времени являются справедливыми в слабом смысле, если каждому процессу дается возможность выполнения. Причина в том, что любой приостановленный процесс в конце концов увидит, что условие (окончания) его задержки истинно.

Справедливости в слабом смысле, однако, недостаточно для гарантии, что любой подходящий оператор await в конце концов выполняется. Это связано с тем, что условие может изменить свое значение (от ложного к истинному и обратно к ложному), пока процесс приостановлен. В этом случае необходима более сильная стратегия планирования.

(2.8) **Справедливость в сильном смысле** (strong fairness). Стратегия планирования *справедлива в сильном смысле*, если: 1) она безусловно справедлива, и 2) любое подходящее условное неделимое действие в конце концов выполняется в предположении, что его условие бывает истинным бесконечно часто.

Условие бывает истинным бесконечно часто, если оно истинно бесконечное число раз в каждой истории (не завершающейся) программы. Чтобы стратегия планирования была справедливой в сильном смысле, действие должно выбираться для выполнения не только тогда, когда его условие ложно, но и когда оно истинно.

Чтобы увидеть различия между справедливостью в слабом и сильном смысле, рассмотрим следующую программу.

```
bool continue = true, try = false;
co while (continue) {try = true; try = false;}
// ⟨await (try) continue = false;⟩
oc
```

При стратегии, справедливой в сильном смысле, эта программа в конце концов завершится, поскольку значение переменной try бесконечно часто истинно. Однако при стратегии планирования, справедливой в слабом смысле, программа может не завершиться, поскольку значение переменной try также бесконечно часто является ложным.

К сожалению, невозможно разработать стратегию планирования процессора, которая была бы и практичной, и справедливой в сильном смысле. Еще раз рассмотрим программу, приведенную выше. На одном процессоре диспетчер, чередующий действия двух процессов, будет справедливым в сильном смысле, поскольку второй процесс будет видеть состояние, в котором значение переменной try истинно. Однако такой планировщик невозможно реализовать практически. Циклическое планирование и планирование с квантованием времени практичны, но в общем случае не являются справедливыми в сильном смысле, поскольку процессы выполняются в непредсказуемом порядке. Диспетчер мультипроцессора, выполняющего процессы параллельно, также практичен, но не является справедливым в сильном смысле. Причина в том, что второй процесс может проверять значение переменной try только тогда, когда оно ложно. Это, конечно, маловероятно, но теоретически возможно.

Для дальнейшего объяснения различных видов стратегий планирования вернемся к программе копирования массива в листингах 2.2 и 2.3. Как отмечалось выше, эта программа свободна от блокировок. Таким образом, программа будет завершена, поскольку каждый процесс регулярно получает возможность продвинуться в своем выполнении. Процесс будет продвигаться, поскольку стратегия справедлива в слабом смысле. Дело в том, что, когда один процесс делает истинным условие (окончания) задержки другого процесса, это условие остается истинным, пока другой процесс не будет продолжен и не изменит разделяемые переменные.

Оба оператора await в программе копирования массива имеют вид ⟨await (B);⟩, а условие B ссылается только на одну переменную, изменяемую другим процессом. Следовательно, оба оператора await могут быть реализованы циклами активного ожидания. Например, ⟨await (p == c);⟩ в процессе Producer может быть реализован следующим оператором.

```
while (p != c);
```

Программа завершится, если стратегия планирования безусловно справедлива, поскольку теперь нет условных неделимых действий и процессы чередуют доступ к разделяемому буферу. Однако в общем случае не верно, что безусловно справедливая стратегия планирования гарантирует завершение цикла активного ожидания. Причина в том, что при безусловно справедливой стратегии всегда может быть запланировано к выполнению неделимое действие, проверяющее условие цикла как раз тогда, когда оно истинно (как в приведенной выше программе).

Если все циклы активного ожидания программы зациклились навсегда, о программе говорят, что она вошла в *активный тупик* (livelock) — программа работает, но процессы не продвигаются. Активный тупик — это активно ожидающий аналог взаимоблокировки (клинча). Отсутствие активного тупика, как и отсутствие клинча, является свойством безопасности. Плохим считается состояние, в котором все процессы зациклены, но не выполняется ни одно из условий (окончания) задержки. С другой стороны, продвижение любого из процессов является свойством живучести. Хорошим в этом случае является то, что цикл активного ожидания отдельного процесса когда-нибудь завершится.

Историческая справка

Одной из первых и наиболее влиятельных статей по параллельному программированию была статья Эдсгера Дейкстры [Dijkstra, 1965]. В ней представлены задача критической секции и оператор parbegin, который позже стал называться оператором cobegin. Наш оператор co является обобщением cobegin. В другой работе Дейкстры [Dijkstra, 1968] были представлены задачи о производителе и потребителе, об обедающих философах и некоторые другие, которые рассматриваются в следующих главах.

Арт Бернстайн [Bernstein, 1966] был первым, кто определил условия, достаточные для независимости, и, значит, возможности параллельного выполнения двух процессов. Условия Бернстайна, как они и сейчас называются, выражены в терминах множеств ввода и вывода для каждого процесса; множество ввода содержит переменные, читаемые процессом, а множество вывода — переменные, записываемые процессом. Три условия Бернстайна для независимости двух процессов состоят в том, что пересечения множеств (вход, выход), (выход, вход) и (выход, выход) не пересекаются между собой. Условия Бернстайна также дают основу для анализа зависимости данных, выполняемого распараллеливающими компиляторами (эта тема описана в главе 12). В нашем определении независимости (2.1) использованы множества чтения и записи для каждого процесса, и переменная находится только в одном из этих множеств для каждого процесса. Это приводит к более простому, но эквивалентному условию.

В большинстве книг по операционным системам показано, как реализация оператора присваивания с использованием регистров и мелкомодульных неделимых действий приводит к задаче критических секций в параллельных программах. Современное аппаратное обеспечение гарантирует, что существует *некоторый* базовый уровень неделимости (обычно слово) для чтения и записи памяти. Статья Лампорта [Lamport, 1977b] содержит интересное обсуждение того, как реализовать неделимые чтение и запись "слов", если неделимым образом могут записываться и читаться только отдельные байты. Его решение не требует использования механизма блокировки для взаимных исключений. (Более новые результаты содержатся в статье [Peterson, 1983].)

Нотация угловых скобок для определения крупномодульных неделимых действий была введена Лампортом, а популяризована Дейкстрой [Dijkstra, 1977]. Овицки и Грис в статье [Owicki and Gries, 1976a] для задания неделимого действия применяли оператор

`await B then S end`. Нотация, использованная в данной книге, комбинирует угловые скобки и вариант оператора `await`. Термины *безусловное* и *условное неделимое действие* были введены Фредом Шнейдером в статье [Schneider and Andrews, 1986].

Существует богатый материал по логическим системам для доказательства свойств последовательных и параллельных программ. В книге [Schneider, 1997] прекрасно охвачены темы разделов 2.6–2.8 и многие другие, а также представлено множество исторических замечаний и ссылок. Нынешнее содержание этих разделов было собрано и переработано из глав 1 и 2 моей предыдущей книги [Andrews, 1991]. Этот материал, в свою очередь, был основан на более ранней работе [Schneider and Andrews, 1986]. Краткая история формальных логических систем и их приложений к проверке программ приведена ниже; более подробная информация и справки есть в книгах [Schneider, 1997] и [Andrews, 1991].

Формальная логика связана с формализацией и анализом методов систематических рассуждений. Ее истоки восходят к древним грекам. В течение следующих двух тысячелетий логикой занимались в основном философы. Однако открытие неевклидовой геометрии в XIX столетии способствовало возобновлению интереса в широких кругах математиков. Это привело к систематическому изучению самой математики как формальной логической системы и появлению новой области математики — математической логики, или, как ее еще называют, метаматематики. Между 1910 и 1913 годами Альфред Норт Уайтхед и Бертран Рассел опубликовали объемный труд *Principia Mathematica*, в котором была представлена система выведения всей математики из логики (так было объявлено авторами). Почти сразу после этого Давид Гильберт попытался доказать, что система, представленная в *Principia Mathematica*, полна и непротиворечива. Однако в 1931 году Курт Гедель показал, что в ней существуют истинные утверждения, которые нельзя доказать ни в этой, ни в *любой* аналогичной аксиоматической системе.

Результат о неполноте, полученный Геделем, установил ограниченность формальных логических систем, но не уничтожил интерес к логике и работу в ее области. Совсем наоборот. Доказательство свойств программ — это лишь одно из многих ее приложений. Формальная логика описана во всех стандартных учебниках по математической логике, или метаматематике. Облегченное развлекательное вступление в эту область можно найти в книге Дугласа Хофстадтера (Douglas Hofstadter), получившей Пулитцеровскую премию, *Gödel, Escher, Bach: An Eternal Golden Braid* [Hofstadter, 1979]. В ней объяснены результаты, полученные Геделем, и показано, как логика связана с вычислимостью и искусственным интеллектом.

Роберта Флойда [Floyd, 1967] принято считать первым, кто предложил технику доказательства правильности программ. В его методе используется связывание предиката с каждой дугой блок-схемы так, что если через эту ветвь проходит выполнение программы, предикат является истинным. Вдохновленный работой Флойда, Хоар [Hoare, 1969] разработал первую формальную логику для доказательства свойств частичной корректности последовательных программ. Он ввел понятие тройки, интерпретации троек, аксиомы и правила взаимного вывода для последовательных операторов (они образовывали подмножество Алгола). Любая логическая система для последовательного программирования, основанная на этом стиле, с тех пор называется "хоаровской логикой". Логика программирования *ЛП* также является примером такой логики.

Первая работа по доказательству свойств параллельных программ тоже была основана на представлении в виде блок-схем [Ashcroft and Manna, 1971]. Примерно в то же время Хоар [Hoare, 1972] расширил свою логику частичной корректности для последовательных программ правилами вывода для параллелизма и синхронизации. Синхронизация задавалась условными критическими областями (аналогичными операторам `await`). Однако логика была не полной, поскольку утверждения в одном процессе не могли ссылаться на переменные в другом процессе, а глобальные инварианты не могли ссылаться на локальные переменные.

Сьюзан Овицки в диссертации, выполненной под руководством Дэвида Гриса, впервые разработала полную логику для доказательства свойств частичной корректности параллельных программ [Owicki, 1975; Owicki and Gries, 1976a, 1976b]. Эта логика включала операторы `cobegin` и синхронизацию с помощью разделяемых переменных, семафоров или условных критических областей. В этой работе описано свойство "не больше одного" (2.2), представле-

на и формализована идея свободы от взаимного вмешательства, проиллюстрированы методы исключения взаимного вмешательства, описанные в разделе 2.7. Автору посчастливилось находиться в Корнельском университете во время выполнения этой работы.

Работа Овицки и Гриса касается только трех свойств безопасности: частичной корректности, взаимного исключения и отсутствия блокировок. Лампорт [Lamport, 1977a] независимо разработал идею, аналогичную свободе от взаимного вмешательства, — монотонные утверждения, как часть общего метода доказательства свойств как безопасности, так и живучести. В его статье также появились сами термины *безопасность* и *живучесть*. Способ доказательства свойства безопасности с использованием глобального инварианта, описанный в разделе 2.8, основан на методе Лампорта. Метод исключения конфигураций принадлежит Шнейдеру [Schneider and Andrews, 1986] и является обобщением метода Овицки и Гриса.

Книга [Francez, 1986] содержит исчерпывающее обсуждение справедливости и ее связи с завершимостью, синхронизацией и защищенными вычислениями. Термины для безусловно справедливой, справедливой в слабом и сильном смысле стратегий планирования взяты из этой книги, в которой есть широкая библиография. Указанные стратегии планирования впервые были определены и формализованы в [Lehman, 1981], хотя и с несколько другой терминологией.

Логика программирования (*ЛП*) — это логика для доказательства свойств безопасности. Один путь построения формальных доказательств свойств живучести состоит в расширении *ЛП* двумя временными операторами: "впредь" (henceforse) и "в конце концов" (eventually). Они позволяют устанавливать утверждения о последовательностях состояний, а значит, и о будущем. Такая *темпоральная* (временная) *логика* была введена в статье Пнуэли [Pnueli, 1977]. Овицки и Лампорт [Owicki and Lamport, 1982] показали, как использовать темпоральную логику и инварианты для доказательства свойств живучести параллельных программ. Обзор темпоральной логики и ее использования в доказательствах живучести содержится в книге Шнейдера [1997].

Литература

Andrews, G. R. 1991. *Concurrent Programming: Principles and Practice.* Menlo Park, CA: Benjamin/ Cummings.

Ashcroft, E. and Z. Manna. 1971. Formalization of properties of parallel programs. *Machine Intelligence* 6: 17–41.

Bernstein, A. J. 1966. Analysis of programs for parallel processing. *IEEE Trans. on Computers EC-15*, 5 (October): 757–762.

Dijkstra, E. W, 1965. Solution of a problem in concurrent programming control. *Comm. ACM* 8, 9 (September): 569.

Dijkstra, E. W. 1968. Cooperating sequential processes. In F. Genuys, ed. *Programming Languages.* New York: Academic Press, pp. 43–112. (Э. Дейкстра. Взаимодействующие последовательные процессы. В сб. "Языки программирования" под ред. Ф. Женюи — М.: Мир, 1972.)

Dijkstra, E. W. 1977. On two beautiful solutions designed by Martin Rem. EWD 629. Reprinted in E. W. Dijkstra. *Selected Writings on Computing: A Personal Perspective.* New York: Springer-Verlag, 1982, pp. 313–318.

Floyd, R. W. 1967. Assigning meanings to programs. *Proc. Amer. Math. Society Symp. in Applied Mathematics* 19: 19–31.

Francez, N. 1986. *Fairness.* New York: Springer-Verlag.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Comm. ACM* 12, 10 (October): 576–580, 583.

Hoare, C. A. R. 1972. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, eds. *Operating Systems Techniques.* New York: Academic Press.

Hofstadter, D. J. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid.* New York: Vintage Books.

Lamport, L. 1977a. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engr. SE-3*, 2 (March): 125–143.

Lamport, L. 1977b. Concurrent reading and writing. *Cumm. ACM* 20, 11 (November): 806–811.

Lehman, D., A. Pnueli, and J. Stavii., 1981. Impartiality, justice, and fairness: The ethics of concurrent termination. *Proc. Eighth Cotton, on Automata, Langs., and Prog.*, Lecture Notes in Computer Science Vol. 115. New York: Springer-Verlag. 264–277.

Owicki, S. S. 1975. Axiomatic proof techniques for parallel programs. TR 75–251. Doctoral dissertation, Ithaca, NY: Cornell University.

Owicki, S. S., and D. Gries. 1976a. An axiomatic proof technique for parallel programs. *Acta Informatica* 6: 319–340.

Owicki, S. S., and D. Cries. 1976b. Verifying properties of parallel programs: An axiomatic approach. *Comm. ACM* 19, 5 (May): 279–285.

Owicki, S., and L. Lamport. 1982. Proving liveness properties of concurrent programs. *ACM Trans. on Prog. Languages and Systems* 4, 3 (July): 455–495.

Peterson, G. L. 1983. Concurrent reading while writing. *ACM. Trans, on Prog. Languages and Systems* 5, 1 (January): 46–55.

Pnueli, A. 1977. The temporal logic of programs. *Proc. 18th Symp. on the Foundations of Computer Science*, November. 46–57.

Schneider, F. B. 1997. *On Concurrent Programming.* New York: Springer.

Schneider, F. B., and G. R. Andrews. 1986. Concepts for concurrent programming. In *Current Trends in Concurrency.* Lecture Notes in Computer Science Vol. 224. New York: Springer-Verlag, 669–716.

Упражнения

2.1. Разберите эскиз программы в листинге 2.1, выводящей все строки с шаблоном `pattern` в файл:

   а) разработайте недостающий код для синхронизации доступа к буферу `buffer`. Для программирования синхронизации используйте оператор `await`;

   б) расширьте программу так, чтобы она читала два файла и выводила все строки, содержащие шаблон `pattern`. Определите независимые операции и используйте отдельный процесс для каждой из них. Напишите весь необходимый код синхронизации.

2.2. Рассмотрите решение задачи копирования массива в листинге 2.2. Измените код так, чтобы переменная `p` была локальной для процесса-производителя, а `c` — для потребителя. Следовательно, эти переменные нельзя использовать для синхронизации доступа к буферу `buf`. Вместо них используйте для синхронизации процессов две новые булевы переменные `empty` и `full`. В начальном состоянии переменная `empty` имеет значение `true`, `full` — `false`. Добавьте новый код для процессов потребителя и производителя. Для программирования синхронизации используйте оператор `await`.

2.3. Команда ОС Unix `tee` вызывается выполнением:

```
tee filename
```

   Эта команда читает стандартный ввод и записывает его в стандартный вывод и в файл `filename`, т.е. создает две копии ввода:

   а) запишите последовательную программу для реализации этой команды;

   б) распараллельте вашу последовательную программу, чтобы она использовала три процесса: чтения из стандартного ввода, записи в стандартный вывод и записи в файл `filename`. Используйте стиль "`co` внутри `while`";

в) измените свое решение из пункта *б*, используя стиль "`while` внутри `co`". В частности, создайте процессы один раз. Используйте двойную буферизацию, чтобы можно было параллельно читать и писать. Для синхронизации доступа к буферам используйте оператор `await`.

2.4. Рассмотрите упрощенную версию команды ОС Unix `diff` для сравнения двух текстовых файлов. Новая команда вызывается так:

```
differ filename1 filename2
```

Она просматривает два файла и печатает все строки, которые в этих файлах отличаются. Команда выводит в стандартный вывод каждую пару несовпадающих строк:

```
lineNumber: line from file 1
lineNumber: line from file 2
```

Если один из файлов длиннее другого, то команда также выводит каждую дополнительную строку более длинного файла (по одной):

а) напишите последовательную программу для реализации команды `differ`;

б) распараллельте свою последовательную программу, чтобы использовались три процесса: два для чтения файлов и один для записи в стандартный вывод. Используйте стиль "`co` внутри `while`";

в) измените свое решение из пункта *б*, используя стиль "`while` внутри `co`". В частности, создайте процесс один раз. Используйте двойную буферизацию, чтобы можно было параллельно читать и записывать. Для синхронизации доступа к буферам используйте оператор `await`.

2.5. Даны массивы целых чисел `a[1:m]` и `b[1:n]`. Предположим, что каждый из них отсортирован по возрастанию, и значения в каждом массиве не повторяются:

а) разработайте последовательную программу подсчета числа разных значений в *обоих* массивах;

б) определите независимые операции в вашей программе и измените ее, чтобы они выполнялись параллельно. Ответ сохраните в разделяемой переменной. Для задания параллельности используйте оператор `co`, а для синхронизации, которая может понадобиться, — `await`.

2.6. Предположим, есть дерево, представленное связанной структурой. Точнее, каждый узел дерева является структурой, состоящей из трех полей: значения и указателей на левое и правое поддеревья. Пустой указатель представлен константой `null`. Напишите рекурсивную параллельную программу для вычисления суммы значений всех узлов дерева. Общее время работы программы должно быть порядка высоты дерева.

2.7. Допустим, что дан проинициализированный массив целых чисел `a[1:n]`:

а) напишите итеративную параллельную программу для вычисления суммы элементов массива `a` с использованием `PR` процессов. Каждый процесс должен работать на полосе массива. Считайте, что `n` кратно `PR`;

б) напишите рекурсивную параллельную программу для вычисления суммы элементов массива. Для деления задачи используйте стратегию "разделяй и властвуй", уменьшая размер задачи вдвое на каждом шаге рекурсии. Останавливайте рекурсию, когда размер задачи станет не больше некоторого порога `T`. Для каждого базового случая используйте последовательный итеративный алгоритм.

2.8. Очередь часто представляют связанным списком. Допустим, переменные `head` и `tail` указывают на первый и последний элементы списка. Каждый элемент содержит поле данных и указатель на следующий элемент. Пустой указатель имеет значение `null`:

а) напишите процедуры: 1) поиска в списке первого элемента (если такой есть), содержащего значение данных d; 2) вставки нового элемента в конец списка; 3) удаления элемента из начала списка. Процедуры поиска и удаления при неудачном выполнении должны возвращать значение null;

б) допустим, что к связанному списку имеют доступ несколько процессов. Определите множества записи и чтения для каждой процедуры в соответствии с (2.1). Определите, какие комбинации процедур могут выполняться параллельно, а какие — по одной, т.е. неделимым образом;

в) добавьте код синхронизации к трем процедурам для задания синхронизации, определенной в ответе *б*. Сделайте свои неделимые действия как можно более мелкими, и не задерживайте выполнение процедур без необходимости. Для программирования синхронизации используйте оператор await.

2.9. Рассмотрите фрагмент кода в разделе 2.6, который присваивает переменной m значение, максимальное из x и y. В тройке для оператора if этого фрагмента кода использовано правило оператора if (см. рис. 2.1). Какими будут предикаты P, Q и B в правиле оператора if для данного приложения?

2.10. Рассмотрите следующую программу:

```
int x = 0, y = 0;
co x = x+1; x = x+2;
// x = x+2; y = y-x;
oc
```

а) допустим, каждый оператор присваивания реализуется одной машинной инструкцией и поэтому неделим. Сколько существует возможных историй? Каковы возможные конечные значения переменных x и y?

б) предположим, что каждый оператор присваивания реализуется тремя неделимыми действиями, которые загружают регистр, складывают или вычитают значение из регистра, а затем сохраняют результат. Сколько возможных историй существует теперь? Каковы возможные заключительные значения переменных x и y?

2.11. Рассмотрите следующую программу:

```
int u = 0, v = 1, w = 2, x;
co x = u + v + w;
//u = 3;
//v = 4;
//w = 5;
oc
```

Предположим, что неделимыми действиями являются чтение и запись отдельных переменных:

а) каковы возможные заключительные значения переменной x, если значение выражения u + v + w вычисляется слева направо?

б) каковы возможные заключительные значения переменной x, если значение выражения u + v + w может быть вычислено в любом порядке?

2.12. Рассмотрите следующую программу:

```
int x = 2, y = 3;
co ⟨x = x + y;⟩ // ⟨y = x * y;⟩
```

а) каковы возможные заключительные значения переменных x и y?

б) допустим, что угловые скобки убраны, а каждый оператор присваивания реализуется тремя неделимыми действиями: чтение переменной, сложение или умножение и запись в переменную. Каковы теперь возможные заключительные значения переменных x и y?

2.13. Рассмотрите следующую программу:

```
S1: x = x + y;
S2: y = x - y;
S3: x = x - y;
```

Предположим, что начальное значение x равно 2, а y — 5. Укажите возможные заключительные значения x и y для каждого из следующих фрагментов. Объясните свои ответы:

а) `S1; S2; S3;`

б) `co ⟨S1;⟩ // ⟨S2;⟩ // ⟨S3;⟩ oc`

в) `co ⟨await (x > y) S1; S2;⟩ // ⟨S3;⟩ oc`

2.14. Рассмотрите следующую программу:

```
int x = 1, y = 1;
co ⟨x = x + y;⟩
// y = 0;
// x = x - y;
oc
```

а) объясните, удовлетворяет ли эта программа требованиям свойства (2.2) "не больше одного";

б) укажите возможные заключительные значения переменных x и y. Объясните свой ответ.

2.15. Рассмотрите следующую программу:

```
int x = 0, y = 10;
co while (x != y) x = x + 1;
// while (x != y) y = y - 1;
oc
```

а) объясните, удовлетворяет ли эта программа требованиям свойства (2.2) "не больше одного";

б) объясните, завершится ли программа. Всегда? Иногда? Никогда?

2.16. Рассмотрите следующую программу:

```
int x = 0;
co ⟨await (x != 0) x = x - 2;⟩
// ⟨await (x != 0) x = x - 3;⟩
// ⟨await (x == 0) x = x + 5;⟩
oc
```

Разработайте схему доказательства, которая покажет, что заключительным значением переменной x является 0. Используйте способ ослабленных утверждений. Определите, какие утверждения критичны по определению (2.5), и покажите, что они не подвержены вмешательству.

2.17. Рассмотрите следующую программу:

```
int x = 0;
co ⟨await (x >= 3) x = x - 2;⟩
// ⟨await (x >= 2) x = x - 3;⟩
// ⟨await (x == 1) x = x + 5;⟩
oc
```

Для каких начальных значений переменной x программа завершится, если применяется справедливая в слабом смысле стратегия планирования? Какими будут конечные значения? Объясните свой ответ.

2.18. Рассмотрите следующую программу:

```
int x = 0;
co ⟨await (x > 0) x = x - 1;⟩
// ⟨await (x < 0) x = x + 2;⟩
// ⟨await (x == 0) x = x -1 ;⟩
oc
```

Для каких начальных значений переменной x программа завершится, если применяется справедливая в слабом смысле стратегия планирования? Какими будут заключительные значения? Объясните свой ответ.

2.19. Рассмотрите следующую программу:

```
int x = 10, y = 0;
co while (x != y) x = x - 1; y = y + 1;
// ⟨await (x == y);⟩ x = 8; y = 2;
oc
```

Объясните, что заставляет программу завершаться. Когда программа завершается, каковы заключительные значения x и y?

2.20. Пусть a[1:m] и b[1:n] — массивы целых чисел, причем m > 0 и n > 0. Напишите предикаты для выражения следующих свойств:

а) все элементы массива a меньше всех элементов массива b;

б) массив a или массив b содержит одно значение 0, но не оба массива вместе;

в) неверно, что оба массива a и b содержат нули;

г) значения в массиве b — те же, что и в массиве a, но расположены в обратном порядке (для этой части предполагается, что n == m);

д) каждый элемент массива a является элементом массива b;

е) некоторый элемент массива a больше, чем некоторый элемент массива b и наоборот.

2.21. Оператор if-then-else имеет вид:

```
if (B)
  S1;
else
  S2;
```

Если условие B истинно, то выполняется оператор S1, иначе — S2. Сформулируйте правило вывода для этого оператора. За идеями обращайтесь к правилу оператора if.

2.22. Рассмотрите следующий оператор for:

```
for [i = 1 to n]
  S;
```

Перепишите оператор, применяя цикл while и явные операторы присваивания счетчику цикла i. Затем используйте аксиому присваивания и правило оператора while (см. рис. 2.1) для разработки правила вывода для такого оператора for.

2.23. Оператор repeat

```
repeat
  S;
until (B);
```

циклически выполняет оператор S до тех пор, пока логическое выражение B не станет истинным в конце некоторой итерации:

а) разработайте правило вывода для оператора repeat;

б) используя ответ к пункту *а*, разработайте схему доказательства, которая демонстрирует, что оператор `repeat` эквивалентен следующим:

```
S; while (!B) S;
```

2.24. Рассмотрите следующее предусловие и оператор присваивания.

```
{x >= 4} ⟨x = x - 4;⟩
```

Для каждой из следующих троек покажите, влияет ли на нее указанный оператор:

а) {x >= 0} ⟨x = x + 5;⟩ {x >= 5};

б) {x >= 0} ⟨x = x + 5;⟩ {x >= 0};

в) {x >= 10} ⟨x = x + 5;⟩ {x >= 11};

г) {x >= 10} ⟨x = x + 5;⟩ {x >= 12};

д) {x *нечетно*} ⟨x = x + 5;⟩ {x *четно*};

е) {x *нечетно*} ⟨y = x + 1;⟩ {y *четно*};

ж) {y *нечетно*} ⟨y = y + 1;⟩ {y *четно*};

з) {x *кратно* 3} y = x; {y *кратно* 3}.

2.25. Рассмотрите следующую программу:

```
int x = V1, y = V2;
x = x + y;
y = x - y;
x = x - y;
```

Добавьте утверждения перед каждым оператором и после него, чтобы охарактеризовать результат работы этой программы. В частности, каковы заключительные значения x и y?

2.26. Рассмотрите следующую программу:

```
int x, y;
co x = x - 1; x = x + 1;
// y = y + 1; y = y - 1;
oc
```

Покажите, что тройка {x == y} S {x == y} является теоремой, где S — это оператор co. Приведите все подробные доказательства взаимного невмешательства.

2.27. Рассмотрите следующую программу:

```
int x = 0;
co ⟨x = x+2;⟩ // ⟨x = x+3;⟩ // ⟨x = x+4;⟩ oc
```

Докажите, что тройка {x == 0} S {x == 9} является теоремой, где S — это оператор co. Используйте способ ослабленных утверждений.

2.28 а) Напишите параллельную программу, которая присваивает значение "истина" булевой переменной `allzero`, если в целочисленном массиве `a[1:n]` все значения — нули. Иначе программа должна присвоить `allzero` значение "ложь". Для параллельной проверки всех элементов используйте оператор co;

б) разработайте схему доказательства, которая продемонстрирует правильность вашего решения. Покажите, что все процессы свободны от взаимного влияния.

2.29 а) Разработайте программу поиска максимального элемента в целочисленном массиве `a[1:n]` путем параллельного просмотра его четных и нечетных элементов.

б) Разработайте схему доказательства, которая продемонстрирует правильность вашего решения. Докажите, что все процессы свободны от взаимного влияния.

2.30. Даны три целочисленные функции: `f(i)`, `g(j)` и `h(k)`. Область определения каждой функции — неотрицательные целые числа. Значения каждой функции возрастают, например, `f(i) < f(i+1)` для всех `i`. Среди значений трех функций есть хотя бы одно общее (это задача о времени ближайшей общей встречи, где области значений функций представляют моменты времени возможной встречи трех людей):

а) напишите параллельную программу для присвоения переменным `i`, `j`, `k` наименьших целых значений, при которых `f(i) == g(j) == h(k)`. Для задания параллельных сравнений используйте оператор `co` (в этой программе используется мелкомодульный параллелизм);

б) разработайте схему доказательства, которая продемонстрирует правильность вашего решения. Покажите, что все процессы свободны от взаимного вмешательства.

2.31. Предположим, что обе тройки $\{P_1\}$ S_1 $\{Q_1\}$ и $\{P_2\}$ S_2 $\{Q_2\}$ истинны и свободны от взаимного вмешательства. Допустим, что оператор S_1 содержит оператор `await` вида ⟨`await (B) T`⟩. Пусть оператор S_1' — это оператор S_1, в котором оператор `await` заменен выражением:

```
while (!B);
T;
```

Ответьте на следующие независимые вопросы:

а) останется ли после этого истинной тройка $\{P_1\}$ S_1' $\{Q_1\}$? Обоснуйте ответ. Она истинна всегда? Иногда? Никогда?

б) останутся ли после этого тройки $\{P_1\}$ S_1' $\{Q_1\}$ и $\{P_2\}$ S_2 $\{Q_2\}$ свободными от взаимного вмешательства? Обоснуйте ответ. Они свободны от взаимного влияния всегда? Иногда? Никогда?

2.32. Рассмотрите следующую программу:

```
int s = 1;

process foo[i = 1 to 2] {
  while (true) {
    ⟨await (s > 0) s = s-1;⟩
    Si;
    ⟨s = s+1;⟩
  }
}
```

Предположим, что S_i — это список операторов, не изменяющий разделяемую переменную `s`:

а) разработайте схемы доказательства для этих двух процессов. Покажите, что доказательства правильности процессов свободны от взаимного влияния. Затем используйте схемы доказательств и метод исключения конфигураций (раздел 2.8), чтобы показать, что операторы S_1 и S_2 не могут выполняться одновременно и программа свободна от взаимоблокировок; (*Указание.* В программе и схемах доказательств понадобятся вспомогательные переменные. Эти переменные должны отслеживать положение каждого процесса.)

б) какая стратегия планирования необходима для обеспечения того, что процесс, приостановленный первым оператором `await`, в конце концов продолжается? Объясните ответ.

2.33. Рассмотрите следующую программу:

```
int x = 10, c = true;
co ⟨await x == 0⟩; c = false;
// while (c) ⟨x = x - 1⟩;
oc
```

а) завершится ли программа при стратегии планирования, справедливой в слабом смысле? Объясните;

б) завершится ли программа при стратегии планирования, справедливой в сильном смысле? Объясните;

в) добавьте в оператор co третью ветвь:

```
while (c) {if (x < 0) ⟨x = 10⟩;}
```

Повторите пункты *а* и *б* для этой программы из трех процессов.

2.34. Задача о восьми ферзях состоит в том, чтобы расположить 8 ферзей на шахматной доске так, чтобы ни один из них не атаковал другого, т.е. не находился с ним на одной вертикали, горизонтали или диагонали.

Напишите параллельную рекурсивную программу, порождающую все 92 решения этой задачи. (*Указание.* Используйте рекурсивную процедуру для размещения ферзей и вторую процедуру для проверки допустимости выбранного решения.)

2.35. Задача об устойчивом паросочетании (о стабильных браках) состоит в следующем. Пусть есть массивы процессов `Man[1:n]` и `Woman[1:n]`. Каждый мужчина оценивает женщин числами от 1 до n, и каждая женщина так же оценивает мужчин. (Получаются перестановки целых чисел от 1 до n.) *Паросочетание* — это взаимно однозначное соответствие между мужчинами и женщинами. Паросочетание *устойчиво*, если для любых двух мужчин `Man[i]`, `Man[j]` и двух женщин `Woman[p]` и `Woman[q]`, соответствующих им в этом паросочетании, выполняются оба следующих условия.

1. `Man[i]` оценивает `Woman[p]` выше, чем `Woman[q]`, или `Woman[q]` оценивает `Man[j]` выше, чем `Man[i]`.
2. `Man[j]` оценивает `Woman[q]` выше, чем `Woman[p]`, или `Woman[p]` оценивает `Man[i]` выше, чем `Man[j]`.

Таким образом, паросочетание неустойчиво, если найдутся мужчина и женщина, предпочитающие друг друга своей текущей паре. Решением задачи об устойчивом паросочетании является множество n-паросочетаний, каждое из которых устойчиво:

а) сформулируйте предикат, который описывает цель задачи об устойчивом паросочетании;

б) напишите параллельную программу для решения данной задачи. Обязательно объясните стратегию вашего решения. Также приведите пред- и постусловия для каждого процесса и инварианты для каждого цикла.

Глава 3

Блокировки и барьеры

Напомним, что в параллельных программах используются два основных типа синхронизации: взаимное исключение и условная синхронизация. В этой главе рассмотрены критические секции и барьеры, показывающие, как программировать эти типы синхронизации. Задача критической секции связана с программной реализацией неделимых действий; эта задача возникает в большинстве параллельных программ. Барьер — это точка синхронизации, которой должны достигнуть все процессы перед тем, как продолжится выполнение одного из них. Барьеры необходимы во многих синхронных параллельных программах.

Взаимное исключение обычно реализуется посредством *блокировок*, которые защищают критические секции кода. В разделе 3.1 определена задача критической секции и представлено крупномодульное решение, использующее оператор `await` для реализации блокировки. В разделе 3.2 разработаны мелкомодульные решения с использованием так называемых *циклических блокировок* (spin lock). В разделе 3.3 представлены три решения со справедливой стратегией: алгоритмы разрыва узла, билета и поликлиники. Эти разные решения иллюстрируют различные подходы к решению данной задачи и имеют разные быстродействия и свойства справедливости. Решения задачи критической секции также важны, поскольку их можно использовать для реализации операторов `await` и, следовательно, произвольных неделимых действий. Как это сделать, показано в конце раздела 3.2.

Во второй половине данной главы представлены три способа выполнения параллельных вычислений: барьерная синхронизация, алгоритмы, параллельные по данным, и так называемый *портфель задач*. Как было отмечено ранее, многие задачи можно решить синхронными итерационными алгоритмами, в которых несколько идентичных процессов итеративно обрабатывают разделяемый массив. Алгоритмы этого типа называются *алгоритмами, параллельными по данным*, поскольку разделяемые данные обрабатываются параллельно и синхронно. В таком алгоритме каждая итерация обычно зависит от результатов предыдущей итерации. Следовательно, в конце итерации более быстрый процесс должен ожидать более медленные, чтобы начать следующую итерацию. Такой тип точки синхронизации называется *барьером*. В разделе 3.4 описываются различные способы реализации барьерной синхронизации и обсуждается согласование их быстродействия. В разделе 3.5 приведено несколько примеров алгоритмов, параллельных по данным и использующих барьеры, а также кратко описана архитектура синхронных мультипроцессоров (SIMD-машин), которые специально приспособлены для реализации алгоритмов, параллельных по данным. Поскольку SIMD-машины выполняют инструкции в блокированном шаге на каждом процессоре, они автоматически реализуют барьеры после каждой машинной инструкции.

В разделе 3.6 представлен другой полезный метод выполнения параллельных вычислений, называемый *портфелем задач*. Этот подход можно использовать для реализации рекурсивного параллелизма и итерационного параллелизма при фиксированном числе независимых задач. Важное свойство парадигмы "портфель задач" состоит в том, что она облегчает сбалансированную загрузку, т.е. гарантирует, что все процессоры выполняют примерно равные объемы работы. В парадигме "портфель задач" использованы блокировки для реализации "портфеля" и барьероподобная синхронизация для определения того, что вычисления окончены.

В программах этой главы используется *активное ожидание* — реализация синхронизации, при которой процесс циклически проверяет некоторое условие до тех пор, пока оно не станет

истинным. Достоинство синхронизации с активным ожиданием состоит в том, что для ее реализации достаточно машинных инструкций современных процессоров. Активное ожидание неэффективно при разделении процессора несколькими процессами (когда их выполнение перемежается), но, когда каждый процесс выполняется на отдельном процессоре, оно эффективно. Многие годы для высокопроизводительных научных вычислений использовались большие мультипроцессоры. В настоящее время на рабочих станциях и даже на персональных компьютерах становятся обычными небольшие мультипроцессоры (с двумя–четырьмя ЦПУ). Как будет показано в разделе 6.2, в ядре операционных систем для мультипроцессоров используется синхронизация с активным ожиданием. Более того, синхронизацию с активным ожиданием использует само аппаратное обеспечение — например, при передаче данных по шинам памяти и локальным сетям.

Библиотеки программ для мультипроцессорных машин включают процедуры для блокировок и иногда для барьеров. Эти библиотечные процедуры реализуются с помощью методов, описанных в данной главе. Две такие библиотеки представлены в конце главы 4 (Pthreads) и главе 12 (OpenMP).

3.1. Задача критической секции

Задача критической секции — это одна из классических задач параллельного программирования. Она стала первой всесторонне изученной проблемой, но интерес к ней не угасает, поскольку критические секции кода есть в большинстве параллельных программ. Кроме того, решение этой задачи можно использовать для реализации произвольных операторов `await`. В данном разделе задача определена и разработано ее крупномодульное решение. В следующих двух разделах построены мелкомодульные решения, иллюстрирующие различные способы решения этой задачи с использованием разных типов машинных инструкций.

В задаче критической секции `n` процессов многократно выполняют сначала критическую, а затем некритическую секцию кода. Критической секции предшествует протокол входа, а следует за ней протокол выхода. Таким образом, предполагается, что процесс имеет следующий вид:

```
process CS[i = 1 to n] {
    while (true) {
        протокол входа;
        критическая секция;
        протокол выхода;
        некритическая секция;
    }
}
```

Каждая критическая секция является последовательностью операторов, имеющих доступ к некоторому разделяемому объекту. Каждая некритическая секция — это еще одна последовательность операторов. Предполагается, что процесс, вошедший в критическую секцию, обязательно когда-нибудь из нее выйдет; таким образом, процесс может завершиться только вне критической секции. Наша задача — разработать протоколы входа и выхода, которые удовлетворяют следующим четырем свойствам.

(3.1) **Взаимное исключение**. В любой момент времени только один процесс может выполнять свою критическую секцию.

(3.2) **Отсутствие взаимной блокировки (живая блокировка)**. Если несколько процессов пытаются войти в свои критические секции, хотя бы один это осуществит.

(3.3) **Отсутствие излишних задержек**. Если один процесс пытается войти в свою критическую секцию, а другие выполняют свои некритические секции или завершены, первому процессу разрешается вход в критическую секцию.

(3.4) **Возможность входа**. Процесс, который пытается войти в критическую секцию, когда-нибудь это сделает.

Первые три свойства являются свойствами безопасности, четвертое — свойством живучести. Для взаимного исключения плохим является состояние, когда два процесса находятся в своих критических секциях. Для отсутствия взаимной блокировки плохим является состояние, когда все процессы ожидают входа, но ни один не может этого сделать. (В решении с активным ожиданием это называется *отсутствием живой блокировки*, поскольку процессы работают, но навсегда зациклены.) Для отсутствия излишних задержек плохим является состояние, когда один-единственный процесс, пытающийся войти в критическую секцию, не может этого сделать, даже если все остальные процессы находятся вне критических секций. Возможность входа является свойством живучести, поскольку зависит от стратегии планирования (об этом ниже).

Тривиальный способ решения задачи критической секции состоит в ограничении каждой критической секции угловыми скобками, т.е. в использовании безусловных операторов `await`. Из семантики угловых скобок сразу следует условие (3.2) — взаимное исключение. Другие три свойства будут удовлетворяться при безусловно справедливой стратегии планирования, поскольку она гарантирует, что процесс, который пытается выполнить неделимое действие, соответствующее его критической секции, в конце концов это сделает, независимо от действий других процессов. Однако при таком "решении" возникает вопрос о том, как реализовать угловые скобки.

Все четыре указанных свойства важны, однако наиболее существенным является взаимное исключение. Таким образом, сначала мы сосредоточимся на нем, а затем узнаем, как обеспечить выполнение остальных. Для описания свойства взаимного исключения необходимо определить, находится ли процесс в своей критической секции. Чтобы упростить запись, построим решение для двух процессов, `CS1` и `CS2`; оно легко обобщается для `n` процессов.

Пусть `in1` и `in2` — логические переменные с начальным значением "ложь". Когда процесс `CS1` (`CS2`) находится в своей критической секции, переменной `in1` (`in2`) присваивается значение "истина". Плохое состояние, которого мы будем стараться избежать, — если и `in1`, и `in2` имеют значение "истина". Таким образом, нам нужно, чтобы для любого состояния выполнялось отрицание условия плохого состояния.

MUTEX: $\neg$(`in1` $\wedge$ `in2`)

Как сказано в разделе 2.7, предикат *MUTEX* должен быть глобальным инвариантом. Для этого он должен выполняться в начальном состоянии и после каждого присваивания переменным `in1` и `in2`. В частности, перед тем, как процесс `CS1` войдет в критическую секцию, сделав тем самым `in1` истинной, он должен убедиться, что `in2` ложна. Это можно реализовать с помощью следующего условного неделимого действия.

```
⟨await (!in2) in1 = true;⟩
```

Процессы симметричны, поэтому во входном протоколе процесса `CS2` используется аналогичное условное неделимое действие. При выходе из критической секции задерживаться ни к чему, поэтому защищать операторы, которые присваивают переменным `in1` и `in2` значение "ложь", нет необходимости.

Решение показано в листинге 3.1. По построению программа удовлетворяет условию взаимного исключения. Взаимная блокировка здесь не возникнет: если бы каждый процесс был заблокирован в своем протоколе входа, то обе переменные, и `in1`, и `in2`, были бы истинными, а это противоречит тому, что в данной точке кода обе они ложны. Излишних задержек также нет, поскольку один процесс блокируется, только если другой находится в критической секции, поэтому нежелательные паузы при выполнении программы не возникают. (Все эти свойства программ можно доказать формально, использовав метод исключения конфигураций, представленный в разделе 2.8.)

Наконец, рассмотрим свойство живучести: процесс, который пытается войти в критическую секцию, в конце концов сможет это сделать. Если процесс `CS1` пытается войти, но не может, то переменная `in2` истинна, и процесс `CS2` находится в критической секции. По предположению процесс в конце концов выходит из критической секции, поэтому переменная `in2` когда-нибудь станет ложной, а переменная защиты входа процесса `CS1` — истинной.

Если процессу CS1 вход все еще не разрешен, это значит, что либо диспетчер несправедлив, либо процесс CS2 снова достиг входа в критическую секцию. В последнем случае описанный выше сценарий повторяется, так что когда-нибудь переменная in2 станет ложной. Таким образом, либо переменная in2 становится ложной бесконечно часто, либо процесс CS2 завершается, и переменная in2 принимает значение "ложь" и остается в таком состоянии. Для того чтобы процесс CS2 в любом случае входил в критическую секцию, нужно обеспечить справедливую в сильном смысле стратегию планирования. (Аргументы для процесса CS2 симметричны.) Напомним, однако, что справедливая в сильном смысле стратегия планирования непрактична, и вернемся к этому вопросу в разделе 3.3.

Листинг 3.1. Задача критической секции: крупномодульное решение

```
bool in1 = false, in2 = false;
## MUTEX: ¬(in1 ∧ in2) -- глобальный инвариант

process CS1 {
  while (true) {
    ⟨await (!in2) in1 = true;⟩ /* вход */
     критическая секция;
    in1 = false;              /* выход */
    некритическая секция;
  }
}

process CS2 {
  while (true) {
    ⟨await (!in1) in2 = true;⟩ /* вход */
    критическая секция;
    in2 = false;              /* выход */
    некритическая секция;
  }
}
```

3.2. Критические секции: активные блокировки

В крупномодульном решении, приведенном в листинге 3.1, используются две переменные. При обобщении данного решения для n процессов понадобятся n переменных. Однако существует только два интересующих нас состояния: или некоторый процесс находится в своей критической секции, или ни одного там нет. Независимо от числа процессов, для того, чтобы различить эти два состояния, достаточно одной переменной.

Пусть lock — логическая переменная, которая показывает, находится ли процесс в критической секции, т.е. lock истинна, когда одна из in1 или in2 истинна, в противном случае lock ложна. Таким образом, получим следующее условие:

```
lock == (in1 ∨ in2)
```

Используя lock вместо in1 и in2, можно реализовать протоколы входа и выхода программы 3.1 так, как показано в листинге 3.2.

Преимущество протоколов входа и выхода, показанных в листинге 3.2, по отношению к протоколам в листинге 3.1 состоит в том, что их можно использовать для решения задачи критической секции при любом числе процессов. Все они будут разделять переменную lock и выполнять одни и те же протоколы.

Листинг 3.2. Критические секции на основе блокировок

```
bool lock = false;

process CS1 {
  while (true) {
    ⟨await (!lock) lock = true;⟩ /* вход */
    критическая секция;
    lock = false;                /* выход */
    некритическая секция;
  }
}

process CS2 {
  while (true) {
    ⟨await (!lock) lock = true;⟩ /* вход */
    критическая секция;
    lock = false;                /* выход */
    некритическая секция;
  }
}
```

3.2.1. "Проверить-установить"

Использование переменной `lock` вместо `in1` и `in2`, показанное в листинге 3.2, очень важно, поскольку почти у всех машин, особенно у мультипроцессоров, есть специальная инструкция для реализации условных неделимых действий. Здесь применяется инструкция, называемая "проверить-установить".[8] В следующем разделе будет использована еще одна подобная инструкция — "извлечь и сложить". Дополнительные инструкции описываются в упражнениях.

Инструкция "проверить-установить" (test and set — `TS`) в качестве аргумента получает разделяемую переменную `lock` и возвращает логическое значение. В неделимом действии инструкция `TS` считывает и сохраняет значение переменной `lock`, присваивает ей значение "истина", а затем возвращает сохраненное предыдущее значение переменной `lock`. Результат действия инструкции `TS` описывается следующей функцией:

(3.6)
```
bool TS(bool lock) {
  ⟨ bool initial = lock; /* сохранить начальное значение */
    lock = true;         /* установить lock */
    return initial; ⟩    /* возвратить начальное значение */
}
```

Используя инструкцию `TS`, можно реализовать крупномодульный вариант программы 3.2 по алгоритму, приведенному в листинге 3.3. Условные неделимые действия в программе 3.2 заменяются циклами. Циклы не завершаются, пока переменная `lock` не станет ложной, т.е. инструкция `TS` возвращает значение "ложь". Поскольку все процессы выполняют одни и те же протоколы, приведенное решение работает при любом числе процессов. Использование блокирующей переменной, как в листинге 3.3, обычно называется *циклической блокировкой* (spin lock), поскольку процесс постоянно повторяет цикл, ожидая снятия блокировки.

Программа в листинге 3.3 имеет следующие свойства. Взаимное исключение (3.2) обеспечено: если несколько процессов пытаются войти в критическую секцию, только один из них первым изменит значение переменной `lock` с ложного на истинное, следовательно, только один из

[8] Напомним, что термин "установить" (без указания значения) обычно применяется в смысле "присвоить значение `true` (или 1)", а "сбросить" — "присвоить `false` (или 0)". — *Прим. ред.*

процессов успешно завершит свой входной протокол и войдет в критическую секцию. Отсутствие взаимной блокировки (3.3) следует из того, что, если оба процесса находятся в своих входных протоколах, то lock ложна, и, следовательно, один из процессов войдет в свою критическую секцию. Нежелательные задержки (3.4) не возникают, поскольку, если оба процесса находятся вне своих критических секций, lock ложна, и, следовательно, один из процессов может успешно войти в критическую секцию, если другой выполняет некритическую секцию или был завершен.

Листинг 3.3. Критические секции на основе инструкции "проверить-установить"

```
bool lock = false;              /* разделяемая переменная */

process CS[i = 1 to n] {
  while (true) {
    while (TS(lock)) skip;      /* протокол входа */
    критическая секция;
    lock = false;               /* протокол выхода */
    некритическая секция;
  }
}
```

С другой стороны, выполнение свойства возможности входа (3.5) не гарантируется. Если используется справедливая в сильном смысле стратегия планирования, то попытки процесса войти в критическую секцию завершатся успехом, поскольку переменная lock бесконечно часто будет принимать значение "ложь". При справедливой в слабом смысле стратегии планирования, которая встречается чаще всего, процесс может навсегда зациклиться в протоколе входа. Однако это может случиться, только если другие процессы все время успешно входят в свои критические секции, чего на практике быть не должно. Следовательно, решение в листинге 3.3 должно удовлетворять условию справедливой стратегии планирования.

Решение задачи критической секции, аналогичное приведенному в листинге 3.3, может быть реализовано на любой машине, если у нее есть инструкция, проверяющая и изменяющая разделяемую переменную в одном неделимом действии. Например, в некоторых машинах есть инструкция инкремента (увеличения), которая увеличивает целое значение переменной и устанавливает код условия, указывающий, положительно ли значение результата. Используя эту инструкцию, можно построить протокол входа, основанный на переходе от нуля к единице. В упражнениях рассмотрены несколько типичных инструкций такого рода. (Это любимый вопрос экзаменаторов!)

Построенное решение с циклической блокировкой и те решения, которые вы, возможно, составите сами, обладают свойством, которое стоит запомнить.

(3.7) **Протоколы выхода в решении с циклической блокировкой.** В решении задачи критической секции с циклической блокировкой протокол выхода должен просто присваивать разделяемым переменным их начальные значения.

В начальном состоянии (см. листинг 3.1) обе переменные in1 и in2 ложны, как и переменная lock (см. листинги 3.2 и 3.3).

3.2.2. "Проверить-проверить-установить"

Хотя решение в листинге 3.3 верно, эксперименты на мультипроцессорных машинах показывают его низкую производительность, если несколько процессов соревнуются за доступ к критической секции. Причина в том, что каждый приостановленный процесс непрерывно обращается к разделяемой переменной lock. Эта "горячая точка" вызывает

конфликт при обращении к памяти, который снижает производительность модулей памяти и шин, связывающих процессор и память.

К тому же инструкция TS при каждом вызове записывает значение в lock, даже если оно не изменяется. Поскольку в мультипроцессорных машинах с разделяемой памятью для уменьшения числа обращений к основной памяти используются кэши, TS выполняется гораздо дольше, чем простое чтение значения разделяемой переменной. (Когда переменная записывается одним из процессоров, ее копии нужно обновить или сделать недействительными в кэшах других процессоров.)

Затраты на обновление содержимого кэш-памяти и конфликты при обращении к памяти можно сократить, изменив протокол входа. Вместо того, чтобы выполнять цикл, пока инструкция TS не возвратит значение "истина", можно использовать следующий протокол.

```
while (lock) skip;  /* пока lock установлена, повторять цикл */
while (TS(lock)) {  /* попытаться захватить lock */
  while (lock) skip;/* повторять цикл, если не удалось */
}
```

Этот протокол называется "*проверить-проверить-установить*", поскольку процесс просто проверяет lock до тех пор, пока не появится возможность выполнения TS. В двух дополнительных циклах lock просто проверяется, так что ее значение можно прочесть из кэш-памяти, не влияя на другие процессоры. Таким образом, конфликты при обращении к памяти сокращаются, но не исчезают. Если флажок блокировки lock сброшен, то как минимум один, а возможно, и все приостановленные процессы могут выполнить инструкцию TS, хотя продолжаться будет только один из них. Ниже мы опишем способы дальнейшего сокращения конфликтов обращения к памяти.

В листинге 3.4 представлено полное решение задачи критической секции, использующее входной протокол "проверить-проверить-установить". Как и ранее, протокол выхода просто очищает переменную lock.

Листинг 3.4. Критические секции на основе протокола "проверить-проверить-установить"

```
bool lock = false;        /* разделяемая блокировка */

process CS[i = 1 to n] {
  while (true) {
    while (lock) skip;    /* протокол входа */
    while (TS(lock)) {
      while (lock) skip;
    }
    критическая секция;
    lock = false;         /* протокол выхода */
    некритическая секция;
  }
}
```

3.2.3. Реализация операторов await

Любое решение задачи критической секции можно использовать для реализации безусловного неделимого действия ⟨S;⟩, скрывая внутренние контрольные точки от других процессов. Пусть *CSenter* — входной протокол критической секции, а *CSexit* — соответствующий выходной. Тогда действие ⟨S;⟩ можно реализовать так:

```
CSenter;
S;
CSexit;
```

Здесь предполагается, что все секции кода процессов, которые изменяют или ссылаются на переменные, изменяемые в S (или изменяют переменные, на которые ссылается S), защищены аналогичными входными и выходными протоколами. В сущности, скобки ⟨ и ⟩ заменены процедурами *CSenter* и *CSexit*.

Приведенный выше образец кода можно использовать в качестве "кирпичика" для реализации операторов ⟨await (B) S;⟩. Напомним, что условное неделимое действие приостанавливает процесс, пока условие B не станет истинным, после чего выполняется S. Когда начинается выполнение S, условие B должно быть истинным. Чтобы обеспечить неделимость всего действия, можно использовать протокол критической секции, скрывая промежуточные состояния в S. Для циклической проверки условия B, пока оно не станет истинным, можно использовать следующий цикл.

```
CSenter;
while (!B) { ??? }
S;
CSexit;
```

Здесь предполагается, что критические секции всех процессов, изменяющих переменные, используемые в B или S, или использующих переменные, изменяемые в S, защищены такими же протоколами входа и выхода.

Остается выяснить, как реализовать тело цикла, указанного выше. Если тело выполняется, значит, условие B было ложным. Следовательно, единственный способ сделать условие B истинным — изменить в другом процессе значения переменных, входящих в это условие. Предполагается, что все операторы, изменяющие эти переменные, находятся в критических секциях, поэтому, ожидая, пока условие B выполнится, нужно выйти из критической секции. Но для обеспечения неделимости вычисления B и выполнения S перед повторным вычислением условия B необходимо вновь войти в критическую секцию. Возможным уточнением указанного выше протокола может быть следующее.

(3.8)
```
CSenter;
while (!B) { CSexit; CSenter; }
S;
CSexit;
```

Данная реализация сохраняет семантику условных неделимых действий при условии, что протоколы критических секций гарантируют взаимное исключение. Если используемая стратегия планирования справедлива в слабом смысле, то процесс, выполняющий (3.8), в конце концов завершит цикл при условии, что B когда-нибудь станет (и останется) истинным. Если стратегия планирования справедлива в сильном смысле, цикл завершится, если условие B становится истинным бесконечно часто.

Программа (3.8) верна, но не эффективна, поскольку процесс, выполняющий ее, повторяет "жесткий" цикл, постоянно выходя из критической секции и входя в нее, но не может продвинуться дальше, пока какой-нибудь другой процесс не изменит переменных в условии B. Это приводит к конфликту обращения к памяти, поскольку каждый приостановленный процесс постоянно обращается к переменным, используемым в протоколах критической секции и условии B.

Чтобы сократить количество конфликтов обращения к памяти, процесс перед повторной попыткой войти в критическую секцию должен делать паузу. Пусть *Delay* — некоторый код, замедляющий выполнение процесса. Тогда программу (3.8) можно заменить следующим протоколом, реализующим условное неделимое действие.

(3.9)
```
CSenter;
while (!B) { CSexit; Delay; CSenter; }
S;
CSexit;
```

Кодом *Delay* может быть, например, пустой цикл, который выполняется случайное число раз. (Во избежание конфликтов памяти в цикле в коде *Delay* следует использовать только локальные переменные.) Этот тип протокола "отхода" ("back-off") полезен и в самих протоколах *CSenter*; например, его можно использовать вместо `skip` в цикле задержки простого протокола "проверить-установить" (см. листинг 3.3).

Если `S` состоит из одного оператора `skip`, протокол (3.9) можно упростить, опустив `S`. Если условие `B` удовлетворяет свойству "не больше одного" (2.2), то оператор ⟨`await (B);`⟩ можно реализовать в следующем виде.

```
while (!B) skip;
```

Как упоминалось в начале этой главы, синхронизация с активным ожиданием часто применяется в аппаратном обеспечении. Фактически протокол, аналогичный (3.9), используется для синхронизации доступа в локальных сетях Ethernet. Чтобы передать сообщение, контроллер Ethernet отправляет его в сеть и следит, не возник ли при передаче конфликт с сообщениями, отправленными примерно в это же время другими контроллерами. Если конфликт не обнаружен, то считается, что передача завершилась успешно. В противном случае контроллер делает небольшую паузу, а затем повторяет передачу сообщения. Чтобы избежать состояния гонок, в котором два контроллера постоянно конфликтуют из-за того, что делают одинаковые паузы, их длительность выбирается случайным образом из интервала, который удваивается при каждом возникновении конфликта. Такой протокол называется *двоичным экспоненциальным протоколом отхода*. Эксперименты показывают, что протокол такого типа полезен также в (3.9) и во входных протоколах критических секций.

3.3. Критические секции: решения со справедливой стратегией

Решения задачи критической секции с циклической блокировкой обеспечивают взаимное исключение, отсутствие взаимных блокировок, активных тупиков и нежелательных пауз. Однако для обеспечения свойства возможности входа (3.5) им необходима справедливая в сильном смысле стратегия планирования. Как сказано в разделе 2.8, стратегии планирования, применяемые на практике, являются справедливыми только в слабом смысле. Маловероятно, что процесс, пытающийся войти в критическую секцию, никогда этого не сделает, однако может случиться, что несколько процессов будут без конца состязаться за вход. В частности, решения с циклической блокировкой не управляют порядком, в котором несколько приостановленных процессов пытаются войти в критические секции.

В данном разделе представлены три решения задачи критической секции со справедливой стратегией планирования: алгоритмы разрыва узла, поликлиники и билета. Они зависят только от справедливой в слабом смысле стратегии планирования, например, от кругового (round-robin) планирования, при котором каждый процесс периодически получает возможность выполнения, а условия задержки, став истинными, остаются таковыми. Алгоритм разрыва узла достаточно прост для двух процессов и не зависит от специальных машинных инструкций, но сложен для n процессов. Алгоритм билета прост для любого числа процессов, но требует специальной инструкции "извлечь и сложить". Алгоритм поликлиники — это вариант алгоритма билета, для которого не нужны специальные машинные инструкции. Поэтому он более сложен (хотя и проще, чем алгоритм разрыва узла для n процессов).

3.3.1. Алгоритм разрыва узла

Рассмотрим решение задачи критической секции для двух процессов (листинг 3.1). Его недостаток в том, что оно не решает, какой из процессов, пытающихся войти в критическую секцию, туда действительно попадет. Например, один процесс может войти в критическую

секцию, выполнить ее, затем вернуться к протоколу входа и снова успешно войти в критическую секцию. Чтобы решение было справедливым, должна соблюдаться очередность входа в критическую секцию, если несколько процессоров пытаются туда войти.

Алгоритм разрыва узла (также называемый алгоритмом Питерсона) — это вариант протокола критической секции (см. листинг 3.1), который "разрывает узел", когда два процесса пытаются войти в критическую секцию. Для этого используется дополнительная переменная, которая фиксирует, какой из процессов вошел в критическую секцию последним.

Чтобы пояснить алгоритм разрыва узла, вернемся к крупномодульной программе в листинге 3.1. Сейчас цель — реализовать условные неделимые действия в протоколах входа с использованием только простых переменных и последовательных операторов. Для начала рассмотрим реализацию оператора `await`, в которой сначала выполняется цикл, пока не будет снята блокировка, а затем присваивание. Протокол входа процесса `CS1` должен выглядеть следующим образом.

```
while (in2) skip;
in1 = true;
```

Протокол входа процесса `CS2` аналогичен:

```
while (in1) skip;
in2 = true;
```

Соответствующий протокол выхода процесса `CS1` должен присвоить значение "ложь" переменной `in1`, а `CS2` — переменной `in2`.

В этом "решении" есть проблема — два действия в протоколе входа не выполняются неделимым образом, поэтому не обеспечено взаимное исключение. Например, желательным постусловием для цикла задержки в процессе `CS1` является `in2 == false`. К сожалению, на него влияет операция присваивания `in2 = true;`, поскольку оба процесса, вычислив свои условия задержки примерно в одно и то же время, могут обнаружить, что оба условия выполняются.

Когда завершается цикл `while`, каждый из процессов должен быть уверен в том, что другой не находится в критической секции. Поэтому рассмотрим протоколы входа с обратным порядком следования операторов. Для процесса `CS1`:

```
in1 = true;
while (in2) skip;
```

Аналогично и для `CS2`:

```
in2 = true;
while (in1) skip;
```

Но этим проблема не решается. Взаимное исключение гарантируется, но появляется возможность взаимной блокировки: если обе переменные `in1` и `in2` истинны, то ни один из циклов ожидания не завершится. Однако есть простой способ избежать взаимной блокировки — использовать дополнительную переменную, чтобы "разорвать узел", если приостановлены оба процесса.

Пусть `last` — целочисленная переменная, которая показывает, какой из процессов `CS1` и `CS2` начал выполнять протокол входа последним. Если оба процесса пытаются войти в критические секции, т.е. `in1` и `in2` истинны, выполнение последнего из них приостанавливается. Это приводит к крупномодульному решению, показанному в листинге 3.5.

Алгоритм программы в листинге 3.5 очень близок к мелкомодульному решению, для которого не нужны операторы `await`. В частности, если все операторы `await` удовлетворяют условию "не больше одного" (2.2), то их можно реализовать в виде циклов активного ожидания. К сожалению, операторы `await` в листинге 3.5 обращаются к двум переменным, каждую из которых изменяет другой процесс. Однако в данном случае нет необходимости в неделимом вычислении условий задержки. Докажем это.

Предположим, процесс `CS1` вычисляет свое условие задержки и обнаруживает, что оно истинно. Если `CS1` обнаружил, что `in2` ложна, то теперь `in2` может быть истинной. Но в этом случае процесс `CS2` только что присвоил переменной `last` значение 2; следовательно, условие задержки остается истинным, если даже значение переменной `in2` изменилось. Если

обнаружено, что last == 2 истинно, то условие остается истинным, поскольку переменная last не изменится, пока процесс CS1 не выполнит свою критическую секцию. Итак, в обеих ситуациях, если для процесса CS1 условие окончания задержки истинно, оно действительно истинно. (Аргументы для процесса CS2 симметричны.)

Листинг 3.5. Алгоритм разрыва узла для двух процессов: крупномодульное решение

```
bool in1 = false, in2 = false;
int last = 1;

process CS1 {
  while (true) {
    last = 1; in1 = true;      /* протокол входа */
    ⟨await (!in2 or last == 2);⟩
    критическая секция;
    in1 = false;               /* протокол выхода */
    некритическая секция;
  }
}

process CS2 {
  while (true) {
    last = 2; in2 = true;      /* протокол входа */
    ⟨await (!in1 or last == 1);⟩
    критическая секция;
    in2 = false;               /* протокол выхода */
    некритическая секция;
  }
}
```

Поскольку условие окончания задержки не обязательно вычислять неделимым образом, каждый оператор await можно заменить циклом while, который повторяется, пока условие окончания задержки ложно. Таким образом, получаем мелкомодульный алгоритм разрыва узла (листинг 3.6).

В этой программе решается проблема критических секций для двух процессов. Такую же основную идею можно использовать для решения задачи при любом числе процессов. В частности, для каждого из n процессов протокол входа должен состоять из цикла, который проходит n-1 этапов. На каждом этапе используются экземпляры алгоритма разрыва узла для двух процессов, чтобы определить, какие процессы проходят на следующий этап. Если гарантируется, что все n-1 этапов может пройти не более, чем один процесс, то в критической секции одновременно будет находиться не больше одного процесса.

Пусть in[1:n] и last[1:n] — целочисленные массивы. Значение элемента in[i] показывает, какой этап выполняет процесс CS[i]. Значение last[j] показывает, какой процесс последним начал выполнять этап j. Эти переменные используются, как показано в листинге 3.7. Внешний цикл for выполняется n-1 раз. Внутренний цикл for процесса CS[i] проверяет все остальные процессы. Процесс CS[i] ждет, если некоторый другой процесс находится на этапе с равным или большим номером этапа, а процесс CS[i] был последним процессом, достигшим этапа j. Как только этапа j достигнет еще один процесс, или все процессы "перед" процессом CS[i] выйдут из своих критических секций, процесс CS[i] получит возможность выполняться на следующем этапе. Таким образом, не более n-1 процессов могут пройти первый этап, n-2 — второй и так далее. Это гарантирует, что пройти все n этапов и выполнять свою критическую секцию процессы могут только по одному.

Листинг 3.6. Алгоритм разрыва узла для двух процессов: мелкомодульное решение

```
bool in1 = false, in2 = false;
int last = 1;

process CS1 {
  while (true) {
    last = 1; in1 = true;      /* протокол входа */
    while (in2 and last == 1) skip;
    критическая секция;
    in1 = false;               /* протокол выхода */
    некритическая секция;
  }
}

process CS2 {
  while (true) {
    last = 2; in2 = true;      /* протокол входа */
    while (in1 and last == 2) skip;
    критическая секция;
    in2 = false;               /* протокол выхода */
    некритическая секция;
  }
}
```

Листинг 3.7. Алгоритм разрыва узла для n процессов

```
int in[1:n] = ([n] 0), last[1:n] = ([n] 0);

process CS[i = 1 to n] {
  while (true) {
    for [j = 1 to n] {      /* протокол входа */
      /* запомним, что процесс i находится на этапе j
         и там является последним */
      last[j] = i; in[i] = j;
      for [k = 1 to n st i != k] {
        /* ждать, если процесс k находится на этапе с большим номером
           и процесс i был последним из прошедших на этап j */
        while (in[k] >= in[i] and last[j] == i) skip;
      }
    }
    критическая секция;
    in[i] = 0;              /* протокол выхода */
    некритическая секция;
  }
}
```

Решение для n процессов свободно от состояний активного тупика, избегает ненужных задержек и гарантирует возможность входа. Эти свойства следуют из того, что данный процесс задерживается, только если некоторый другой процесс находится в протоколе входа впереди данного, и из предположения, что каждый процесс в конце концов выходит из своей критической секции.

3.3.2. Алгоритм билета

Алгоритм разрыва узла для n процессов весьма сложен и неясен, и отчасти потому, что не очевидно обобщение алгоритма для двух процессов на случай n процессов. Построим более прозрачное решение задачи критической секции для n процессов, иллюстрирующее, как для упорядочения процессов используются целочисленные счетчики. Это решение называется *алгоритмом билета*, поскольку основано на вытягивании билетов (номеров) и последующего ожидания очереди.

В некоторых магазинах используется следующий метод обслуживания покупателей (посетителей) в порядке их прибытия: входя в магазин, посетитель получает номер, который больше номера любого из ранее вошедших. Затем посетитель ждет, пока обслужат всех людей, получивших меньшие номера. Этот алгоритм реализован с помощью автомата для выдачи номеров и счетчика, отображающего номер обслуживаемого посетителя. Если за счетчиком следит один работник, посетители обслуживаются по одному в порядке прибытия. Описанную идею можно использовать и для реализации справедливого протокола критической секции.

Пусть `number` и `next` — целые переменные с начальными значениями 1, а `turn[1:n]` — массив целых с начальными значениями 0. Чтобы войти в критическую секцию, процесс `CS[i]` сначала присваивает элементу `turn[i]` текущее значение `number` и увеличивает значение `number` на 1. Чтобы процессы (посетители) получали уникальные номера, эти действия должны выполняться неделимым образом. После этого процесс `CS[i]` ожидает, пока значение `next` не станет равным полученному им номеру. При завершении критической секции процесс `CS[i]` увеличивает на 1 значение `next`, снова в неделимом действии.

Описанный протокол реализован в алгоритме, приведенном в листинге 3.8. Поскольку значения `number` и `next` считываются и увеличиваются в неделимых действиях, следующий предикат будет глобальным инвариантом.

```
TICKET: next > 0 ∧ (∀ i: 1 <= i <= n:
  (CS[i] в своей критической секции) ⇒ (turn[i] == next) ∧
  (turn[i] > 0) ⇒ (∀ j: 1 <= j <= n, j != i:
                                turn[i] != turn[j]) )
```

Последняя строка гласит, что ненулевые значения элементов массива `turn` уникальны. Следовательно, только один `turn[i]` может быть равен `next`, т.е. только один процесс может находиться в критической секции. Отсюда же следует отсутствие взаимоблокировок и ненужных задержек. Этот алгоритм гарантирует возможность входа при стратегии планирования, справедливой в слабом смысле, поскольку условие окончания задержки, ставшее истинным, таким и остается.

Листинг 3.8. Алгоритм билета: крупномодульное решение

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
## глобальный инвариант — предикат TICKET (см. текст)

process CS[i = 1 to n] {
  while (true) {
    ⟨turn[i] = number; number = number + 1;⟩
    ⟨await (turn[i] == next);⟩
    критическая секция;
    ⟨next = next + 1;⟩
    некритическая секция;
  }
}
```

В отличие от алгоритма разрыва узла, алгоритм билета имеет потенциальный недостаток, общий для алгоритмов, использующих увеличение счетчиков: значения `number` и `next` не ограниче-

ны. Если алгоритм билета выполнять достаточно долго, увеличение счетчиков приведет к арифметическому переполнению. Однако на практике это крайне маловероятно и не является проблемой.

Алгоритм в листинге 3.8 содержит три крупномодульных действия. Оператор `await` легко реализуется циклом с активным ожиданием, поскольку в булевом выражении использована только одна разделяемая переменная. Последнее неделимое действие, увеличение `next`, можно реализовать с помощью обычных инструкций загрузки и сохранения, поскольку в любой момент времени только один процесс выполняет протокол выхода. К сожалению, первое неделимое действие (чтение значения `number` и его увеличение) реализовать непросто.

У некоторых машин есть инструкции, которые возвращают старое значение переменной и увеличивают или уменьшают ее в одном неделимом действии. Эти инструкции выполняют именно то, что нужно для реализации алгоритма билета. В качестве примера приведем инструкцию "извлечь и сложить" (Fetch-and-Add — FA), которая работает следующим образом.

```
FA(var, incr):
    ⟨int tmp = var; var = var + incr; return(tmp);⟩
```

В листинге 3.9 показан алгоритм билета, реализованный с помощью инструкции `FA`.

Листинг 3.9. Алгоритм билета: мелкомодульное решение

```
int number = 1, next = 1, turn[1:n] = ([n] 0);

process CS[i = 1 to n] {
  while (true) {
    turn[i] = FA(number,1);      /* протокол входа */
    while (turn[i] != next) skip;
    критическая секция;
    next = next + 1;             /* протокол выхода */
    некритическая секция;
  }
}
```

Для машин, не имеющих инструкции типа "извлечь и сложить", необходим другой метод. Главное требование алгоритма билета — каждый процесс должен получить уникальный номер. Если у машины есть инструкция неделимого увеличения, первый шаг протокола входа можно реализовать так:

```
turn[i] = number; ⟨number = number + 1;⟩
```

Переменная `number` гарантированно увеличивается правильно, но процессы могут не получить уникальных номеров. Например, каждый из процессов может выполнить первое присваивание примерно в одно и то же время и получить один и тот же номер! Поэтому важно, чтобы оба присваивания выполнялись в одном неделимом действии.

Нам уже известны два других способа решения задачи критической секции: циклические блокировки и алгоритм разрыва узла. Чтобы обеспечить неделимость получения номеров, можно воспользоваться любым из них. Например, пусть *CSenter* — протокол входа критической секции, а *CSexit* — соответствующий протокол выхода. Тогда в программе 3.9 инструкцию "извлечь и сложить" можно заменить следующей последовательностью.

(3.10) *CSenter*; `turn[i] = number; number = number+1;` *CSexit*;

Такой подход выглядит необычным, но на практике он работает хорошо, особенно если для реализации протоколов входа и выхода доступна инструкция типа "проверить-установить". При использовании инструкции "проверить-установить" процессы получают номера не обязательно в соответствии с порядком, в котором они пытаются это сделать, и теоретически процесс может зациклиться навсегда. Но с очень высокой вероятностью каждый процесс получит номер, и большинство номеров будут выбраны по порядку. Причина в том, что крити-

ческая секция в (3.10) очень коротка, и процесс не должен задерживаться в протоколе входа *CSenter*. Основной источник задержек в алгоритме билета — это ожидание, пока значение переменной `next` не станет равно значению `turn[i]`.

3.3.3. Алгоритм поликлиники

Алгоритм билета можно непосредственно реализовать на машинах, имеющих операцию типа "извлечь и сложить". Если такая инструкция недоступна, можно промоделировать часть алгоритма билета, в которой происходит получение номера с использованием (3.10). Но для этого нужен еще один протокол критической секции, и решение не обязательно будет обладать свойством справедливости. Здесь представлен *алгоритм поликлиники*, подобный алгоритму билета. Он обеспечивает справедливость планирования и не требует специальных машинных инструкций. Естественно, он сложнее, чем алгоритм билета (см. листинг 3.9).

По алгоритму билета каждый посетитель получает уникальный номер и ожидает, пока значение `next` не станет равным этому номеру. Алгоритм поликлиники использует другой подход. Входя, посетитель смотрит на всех остальных и выбирает номер, который больше любого другого. Все посетители должны ждать, пока назовут их номер. Как и в алгоритме билета, следующим обслуживается посетитель с наименьшим номером. Отличие состоит в том, что для определения очередности обслуживания посетители сверяются друг с другом, а не с общим счетчиком.

Как и в алгоритме билета, пусть `turn[1:n]` — массив целых с начальными значениями `0`. Чтобы войти в критическую секцию, процесс `CS[i]` сначала присваивает переменной `turn[i]` значение, которое на 1 больше, чем максимальное среди текущих значений элементов массива `turn`. Затем `CS[i]` ожидает, пока значение `turn[i]` не станет наименьшим среди ненулевых элементов массива `turn`. Таким образом, инвариант алгоритма поликлиники выражается следующим предикатом.

CLINIC: (∀ i: 1 <= i <= n:
(CS[i] *в своей критической секции*) ⇒ (turn[i] > 0) ∧
(turn[i] > 0) ⇒ (∀ j: 1<= j <= n, j != i:
turn[j] == 0 ∨ turn[i] < turn[j]))

Выходя из критической секции, процесс `CS[i]` присваивает `turn[i]` значение `0`.

В листинге 3.10 показан крупномодульный вариант алгоритма поликлиники, соблюдающий поставленные условия. Первое неделимое действие обеспечивает уникальность всех ненулевых значений элементов массива `turn`. Оператор `for` гарантирует, что следствие предиката *CLINIC* истинно, когда процесс `P[i]` выполняет свою критическую секцию. Этот алгоритм удовлетворяет условию взаимного исключения, поскольку одновременно не могут быть истинными условия `turn[i] != 0` для всех `i` и *CLINIC*. Ненулевые значения элементов массива `turn` уникальны и, как обычно, предполагается, что каждый процесс в конце концов выходит из своей критической секции, поэтому взаимных блокировок нет. Отсутствуют также излишние задержки процессов, поскольку сразу после выхода процесса `CS[i]` из критической секции `turn[i]` получает значение `0`. Наконец, алгоритм поликлиники гарантирует возможность входа в критическую секцию, если планирование справедливо в слабом смысле, поскольку ставшее истинным условие окончания задержки остается таковым. (Значения элементов массива `turn` в алгоритме поликлиники могут быть как угодно велики, но значения элементов массива `turn` продолжают возрастать, только если *всегда* есть хотя бы один процесс, пытающийся войти в критическую секцию. Однако на практике это маловероятно.)

Алгоритм поликлиники в листинге 3.10 нельзя непосредственно реализовать на современных машинах. Чтобы присвоить переменной `turn[i]`, необходимо найти максимальное из `n` значений, а оператор `await` дважды обращается к разделяемой переменной `turn[j]`. Эти операции можно было бы реализовать неделимым образом, используя еще один протокол критической секции, например, алгоритм разрыва узла, но это слишком неэффективно. К счастью, есть более простой выход.

Листинг 3.10. Алгоритм поликлиники: крупномодульное решение

```
int turn[1:n] = ([n] 0);
## глобальный инвариант — предикат CLINIC (см. текст)

process CS[i = 1 to n] {
  while (true) {
    ⟨turn[i] = max(turn[1:n]) + 1;⟩
    for [j = 1 to n st j != i]
      ⟨await (turn[j] == 0 or turn[i] < turn[j]);⟩
    критическая секция;
    turn[i] = 0;
    некритическая секция;
  }
}
```

Если необходимо синхронизировать n процессов, полезно сначала разработать решение для двух процессов, а затем обобщить его (так мы поступили с алгоритмом разрыва узла). Итак, рассмотрим следующий протокол входа для процесса CS1.

```
turn1 = turn2 + 1;
while (turn2 != 0 and turn1 > turn2) skip;
```

Аналогичен и следующий протокол входа для процесса CS2.

```
turn2 = turn1 + 1;
while (turn1 != 0 and turn2 > turn1) skip;
```

Каждый процесс присваивает значение своей переменной turn в соответствии с оптимизированным вариантом (3.10), а операторы await реализованы в виде цикла с активным ожиданием.

Проблема этого "решения" в том, что ни операторы присваивания, ни циклы while не выполняются неделимым образом. Следовательно, процессы могут начать выполнение своих протоколов входа приблизительно одновременно, и оба присвоят переменным turn1 и turn2 значение 1. Если это случится, оба процесса окажутся в своих критических секциях в одно и то же время.

Частично решить эту проблему можно по аналогии с алгоритмом 3.6: если обе переменные turn1 и turn2 имеют значения 1, то один из процессов должен выполняться, а другой — приостанавливаться. Например, пусть выполняется процесс с меньшим номером; тогда в условии цикла задержки процесса CS2 изменим второй конъюнкт: turn2 >= turn1.

К сожалению, оба процесса все еще могут одновременно оказаться в критической секции. Допустим, что процесс CS1 считывает значение turn2 и получает 0. Процесс CS2 начинает выполнять свой протокол входа, определяет, что переменная turn1 все еще имеет значение 0, присваивает turn2 значение 1 и входит в критическую секцию. В этот момент CS1 может продолжить выполнение своего протокола входа, присвоить turn1 значение 1 и затем войти в критическую секцию, поскольку переменные turn1 и turn2 имеют значение 1, и процесс CS1 в этом случае получает преимущество. Такая ситуация называется *состоянием гонок*, поскольку процесс CS1 "обгоняет" CS2 и не учитывает, что процесс CS2 изменил переменную turn2.

Чтобы избежать состояния гонок, необходимо, чтобы каждый процесс присваивал своей переменной turn значение 1 (или любое отличное от нуля) в самом начале протокола входа. После этого процесс должен проверить значение переменной turn других процессов и переприсвоить значение своей переменной, т.е. протокол входа процесса CS1 выглядит следующим образом.

```
turn1 = 1; turn1 = turn2 + 1;
while (turn2 != 0 and turn1 > turn2) skip;
```

Протокол входа процесса CS2 аналогичен.

```
turn2 = 1; turn2 = turn1 + 1;
while (turn1 != 0 and turn2 >= turn1) skip;
```

Теперь один процесс не может выйти из цикла while, пока другой не закончит начатое ранее присваивание turn. В этом решении процессу CS1 отдается преимущество перед CS2, когда у обоих процессов ненулевые значения переменной turn.

Протоколы входа процессов несимметричны, поскольку условие задержки второго цикла слегка отличается. Однако их можно записать и в симметричном виде. Пусть (a,b) и (c,d) — пары целых чисел. Определим отношение сравнения для них таким образом:

```
(a,b) > (c,d) == true,  если a > c или a == c и b > d
              == false, иначе
```

Теперь можно переписать условие turn1 > turn2 процесса CS1 в виде (turn1,1) > (turn2,2), а условие turn2 >= turn1 в процессе CS2 — (turn2,2) > (turn1,1).

Достоинство симметричной записи в том, что теперь алгоритм поликлиники для двух процессов легко обобщить на случай n процессов (листинг 3.11). Каждый из процессов сначала показывает, что он собирается войти в критическую секцию, присваивая своей переменной turn значение 1. Затем он находит максимальное значение из всех turn[i] и прибавляет к нему 1. Наконец, процесс запускает цикл for и, как в крупномодульном решении, ожидает своей очереди. Отметим, что максимальное значение массива определяется считыванием всех его элементов и выбором наибольшего. Эти действия *не являются* неделимыми, поэтому точный результат не гарантируется. Однако, если несколько процессов получают одно и то же значение, они упорядочиваются в соответствии с правилом, описанным выше.

Листинг 3.11. Алгоритм поликлиники: мелкомодульное решение

```
int turn[1:n] = ([n] 0);

process CS[i = 1 to n] {
  while (true) {
    turn[i] = 1; turn[i] = max(turn[1:n])+1;
    for [j = 1 to n st j != i]
      while (turn[j] != 0 and
             (turn[i],i) > (turn[j],j)) skip;
    критическая секция;
    turn[i] = 0;
    некритическая секция;
  }
}
```

3.4. Барьерная синхронизация

Многие задачи можно решить с помощью итерационных алгоритмов, которые последовательно вычисляют приближения к решению и завершаются, когда получен окончательный ответ или достигнута заданная точность вычислений (как во многих численных методах). Обычно такие алгоритмы работают с массивами чисел, и на каждой итерации одни и те же действия выполняются над всеми элементами массива. Следовательно, для синхронного параллельного вычисления отдельных частей решения можно использовать параллельные процессы. Мы уже видели несколько примеров такого рода; гораздо больше их представлено в следующем разделе и дальнейших главах.

Основным свойством большинства параллельных итерационных алгоритмов является зависимость результатов каждой итерации от результатов предыдущей. Один из способов построить такой алгоритм — реализовать тело каждой итерации, используя операторы co. Если не учитывать завершимость и считать, что на каждой итерации выполняется n задач, получим такой общий вид алгоритма.

```
while (true) {
  co [i = 1 to n]
```

```
        код решения задачи i;
      oc
    }
```

К сожалению, этот подход весьма неэффективен, поскольку оператор co порождает n процессов на каждой итерации. Создать и уничтожить процессы намного дороже, чем реализовать их синхронизацию. Поэтому альтернативная структура алгоритма делает его намного эффективнее — процессы создаются один раз в начале вычислений, а потом синхронизируются в конце каждой итерации.

```
process Worker[i = 1 to n] {
  while (true) {
    код решения задачи i;
    ожидание завершения всех n задач;
  }
}
```

Точка задержки в конце каждой итерации представляет собой барьер, которого для продолжения работы должны достигнуть все процессы, поэтому этот механизм называется *барьерной синхронизацией*. Барьеры могут понадобиться в конце циклов, как в данном примере, или на промежуточных стадиях, как будет показано ниже.

Далее разработано несколько реализаций барьерной синхронизации, использующих различные способы взаимодействия процессов, и описано, при каких условиях лучше всего использовать каждый тип барьера.

3.4.1. Разделяемый счетчик

Простейший способ описать требования к барьеру — использовать разделяемый целочисленный счетчик, скажем, count с нулевым начальным значением. Предположим, что есть n рабочих процессов, которые должны собраться у барьера. Когда процесс доходит до барьера, он увеличивает значение переменной count. Когда значение счетчика count станет равным n, все процессы смогут продолжить работу. Получаем следующий код.

(3.11)
```
int count = 0;

process Worker[i = 1 to n] {
  while (true) {
    код реализации задачи i;
    ⟨count = count + 1;⟩
    ⟨await (count == n);⟩
  }
}
```

Оператор await можно реализовать циклом с активным ожиданием. При наличии неделимой инструкции типа FA ("извлечь и сложить"), определенной в разделе 3.3, этот барьер можно реализовать следующим образом.

```
FA(count,1);
while (count != n) skip;
```

Однако данный код не вполне соответствует поставленной задаче. Сложность состоит в том, что значением count должен быть 0 в начале каждой итерации, т.е. count нужно обнулять каждый раз, когда все процессы пройдут барьер. Более того, она должна иметь значение 0 перед тем, как любой из процессов вновь попытается ее увеличить.

Эту проблему можно решить с помощью двух счетчиков, один из которых увеличивается до n, а другой уменьшается до 0. Их роли меняются местами после каждой стадии. Однако использование разделяемых счетчиков приводит к чисто практическим трудностям. Во-первых, увели-

чивать и уменьшать их значения нужно неделимым образом. Во-вторых, когда в программе (3.11) процесс приостанавливается, он непрерывно проверяет значение переменной count. В худшем случае n-1 процессов будут ожидать, пока последний процесс достигнет барьера. В результате возникнет серьезный конфликт обращения к памяти, если только программа не выполняется на мультипроцессорной машине с согласованной кэш-памятью. Но даже в этом случае значение счетчика count непрерывно изменяется, и необходимо постоянно обновлять каждый кэш. Таким образом, реализация барьера с использованием счетчиков возможна, только если на целевой машине есть неделимые инструкции увеличения и согласованная кэш-память с эффективным механизмом обновления. Кроме того, число n должно быть относительно мало.

3.4.2. Флаги и управляющие процессы

Один из способов избежать конфликтов обращения к памяти — реализовать счетчик count с помощью n переменных, значения которых прибавляются к одному и тому же значению. Пусть, например, есть массив целых arrive[1:n] с нулевыми начальными значениями. Заменим операцию увеличения счетчика count в программе (3.11) присваиванием arrive[i] = 1. Тогда глобальным инвариантом станет такой предикат.

```
count == (arrive[1] + ... + arrive[n])
```

Если элементы массива arrive хранятся в разных строках кэш-памяти (для их бесконфликтной записи процессами), то конфликтов обращения к памяти не будет.

В программе (3.11) осталось реализовать оператор await и обнулить элементы массива arrive в конце каждой итерации. Оператор await можно было бы записать в таком виде.

```
⟨await ((arrive[1] + ... + arrive[n]) == n);⟩
```

Но в таком случае снова возникают конфликты обращения к памяти, причем это решение также неэффективно, поскольку сумму элементов arrive[i] теперь постоянно вычисляет каждый ожидающий процесс Worker.

Обе проблемы — конфликтов обращения к памяти и обнуления массива — можно решить, используя дополнительный набор разделяемых значений и еще один процесс, Coordinator. Пусть каждый процесс Worker вместо того, чтобы суммировать элементы массива arrive, ждет, пока не станет истинным единственное логическое значение. Пусть continue[1:n] — дополнительный массив целых с нулевыми начальными значениями. После того как Worker[i] присвоит 1 элементу arrive[i], он должен ждать, пока значением переменной continue[i] не станет 1.

(3.12)
```
arrive[i] = 1;
⟨await (continue[i] == 1);⟩
```

Процесс Coordinator ожидает, пока все элементы массива arrive не станут равны 1, затем присваивает значение 1 всем элементам массива continue.

(3.13)
```
for [i = 1 to n] ⟨await (arrive[i] == 1);⟩
for [i = 1 to n] continue[i] = 1;
```

Операторы await в (3.12) и (3.13) можно реализовать в виде циклов while, поскольку каждый из них ссылается только на одну разделяемую переменную. В процессе Coordinator для ожидания установки всех элементов arrive можно использовать оператор for. Поскольку для продолжения процессов Worker должны быть установлены все элементы arrive, процесс Coordinator может проверять их в любом порядке. Конфликтов обращения к памяти теперь не будет, поскольку процессы ожидают изменения различных переменных, каждая из которых может храниться в своей строке кэш-памяти.

Переменные arrive и continue в программах (3.12) и (3.13) являются примерами так называемого *флага (флажка)*. Его устанавливает (поднимает) один процесс, чтобы сообщить другому о выполнении условия синхронизации. Дополним программы (3.12) и (3.13) кодом, который сбрасывает флаги (присваивая им значение 0) для подготовки к следующей итерации. При этом используются два основных правила.

(3.14) **Правила синхронизации с помощью флагов**: а) флаг синхронизации сбрасывается только процессом, ожидающим его установки; б) флаг нельзя устанавливать до тех пор, пока не известно точно, что он сброшен.

Первое правило гарантирует, что флаг не будет сброшен, пока процесс не определит, что он установлен. В соответствии с этим правилом в (3.12) флаг continue[i] должен сбрасываться процессом Worker[i], а обнулять все элементы массива arrive в (3.13) должен Coordinator. В соответствии со вторым правилом один процесс не устанавливает флаг, пока он не сброшен другим. В противном случае, если другой синхронизируемый процесс в дальнейшем ожидает повторной установки флага, возможна взаимная блокировка. В (3.13) это означает, что Coordinator должен сбросить arrive[i] перед установкой continue[i]. Coordinator может сделать это, выполнив еще один оператор for после первого for в (3.13). Coordinator может также сбросить arrive[i] сразу после того, как дождался его установки. Добавив код сброса флагов, получим барьер с управляющим (листинг 3.12).

Листинг 3.12. Барьерная синхронизация с управляющим процессом

```
int arrive[1:n] = ([n] 0), continue[1:n] = ([n] 0);

process Worker[i = 1 to n] {
  while (true) {
    код решения задачи i;
    arrive[i] = 1;
    ⟨await (continue[i] == 1);⟩
    continue[i] = 0
  }
}

process Coordinator {
  while (true) {
    for [i = 1 to n] {
      ⟨await (arrive[i] == 1);⟩
      arrive[i] = 0;
    }
    for [i = 1 to n] continue[i] = 1;
  }
}
```

Хотя в программе 3.12 барьерная синхронизация реализована так, что конфликты обращения к памяти исключаются, у данного решения есть два нежелательных свойства. Во-первых, нужен дополнительный процесс. Синхронизация с активным ожиданием эффективна, если только каждый процесс выполняется на отдельном процессоре, так что процессу Coordinator нужен свой собственный процессор. Но, возможно, было бы лучше использовать этот процессор для другого рабочего процесса.

Второй недостаток использования управляющего процесса состоит в том, что время выполнения каждой итерации процесса Coordinator, и, следовательно, каждого экземпляра барьерной синхронизации пропорционально числу процессов Worker. В итерационных алгоритмах часто все рабочие процессы имеют идентичный код. Это значит, что если каждый

рабочий процесс выполняется на отдельном процессоре, то все они подойдут к барьеру примерно в одно время. Таким образом, все флаги `arrive` будут установлены практически одновременно. Однако процесс `Coordinator` проверяет флаги в цикле, по очереди ожидая, когда каждый из них будет установлен.

Обе проблемы можно преодолеть, объединив действия управляющего и рабочих процессов так, чтобы каждый рабочий процесс был одновременно и управляющим. Организуем рабочие процессы в дерево (рис. 3.1). Сигнал о том, что процесс подошел к барьеру (флаг `arrive[i]`), отсылается вверх по дереву, а сигнал о разрешении продолжения выполнения (флаг `continue[i]`) — вниз. Узел рабочего процесса ждет, когда к барьеру подойдут его сыновья, после чего сообщает родительскому узлу о том, что он тоже подошел к барьеру. Когда все сыновья корневого узла подошли к барьеру, это значит, что все остальные рабочие узлы тоже подошли к барьеру. Тогда корневой узел может сообщить сыновьям, что они могут продолжить выполнение. Те, в свою очередь, разрешают продолжить выполнение своим сыновьям, и так далее. Специфические действия, которые должен выполнить узел каждого вида, описаны в листинге 3.13. Операторы `await` в данном случае можно реализовать в виде циклов активного ожидания.

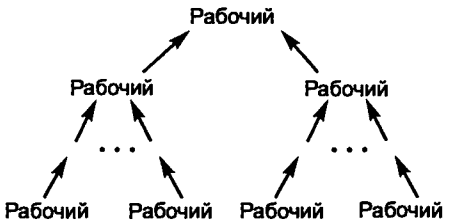


Рис. 3.1. Барьер с древовидной структурой

Листинг 3.13. Барьерная синхронизация с помощью объединяющего дерева

```
узел-лист L: arrive[L] = 1;
             ⟨await (continue[L] == 1);⟩
             continue[L] = 0;
промежуточный узел I: ⟨await (arrive[left] == 1);⟩
                      arrive[left] = 0;
                      ⟨await (arrive[right] == 1);⟩
                      arrive[right] = 0;
                      arrive[I] = 1;
                      ⟨await (continue[I] == 1);⟩
                      continue[I] = 0;
                      continue[left] = 1; continue[right] = 1;
корневой узел R: ⟨await (arrive[left] == 1);⟩
                 arrive[left] = 0;
                 ⟨await (arrive[right] == 1);⟩
                 arrive[right] = 0;
                 continue[left] = 1; continue[right] = 1;
```

Реализация, приведенная в листинге 3.13, называется *барьером с объединяющим деревом*, поскольку каждый процесс объединяет результаты работы своих сыновних процессов и отправляет родительскому. Этот барьер использует столько же переменных, сколько и "централизованная" версия с управляющим процессом, но он намного эффективнее при больших n, поскольку высота дерева пропорциональна $\log_2 n$.

Объединяющее дерево можно сделать еще эффективнее, если корневой узел будет отправлять единственное сообщение, которое разрешает продолжать работу всем остальным узлам. Например, узлы могут ждать, пока корневой узел не установит флаг `continue`. Сбрасывать флаг `continue` можно двумя способами. Первый способ — применить двойную буферизацию, т.е. использовать два флага продолжения и переключаться между ними. Второй способ — изменять смысл флага продолжения, т.е. на четных циклах ждать, когда его значением станет 1, а на нечетных — 0.

3.4.3. Симметричные барьеры

В барьере с объединяющим деревом процессы играют разные роли. Промежуточные узлы дерева выполняют больше действий, чем листья или корень. Кроме того, корневой узел должен ждать, пока сигналы о прибытии пройдут через все дерево. Если все процессы работают по одному алгоритму и выполняются на отдельных процессорах, то к барьеру они подойдут примерно одновременно. Таким образом, если все процессы по пути к барьеру выполняют одну и ту же последовательность действий, то они могут пройти барьер с одинаковой скоростью. В этом разделе представлены *симметричные барьеры*, особенно удобные на многопроцессорных машинах с разделенной памятью и неоднородным временем доступа к памяти.

Симметричный барьер для n процессов строится из пар простых двухпроцессных барьеров. Чтобы создать двухпроцессный барьер, можно использовать метод "управляющий-рабочий", но тогда действия двух процессов будут различаться. Вместо этого можно создать полностью симметричный барьер. Пусть каждый процесс при достижении им барьера устанавливает собственный флаг. Если `W[i]` и `W[j]` — два процесса, то симметричный барьер для них реализуется следующим образом.

(3.15)
```
/* код барьера для рабочего процесса W[i]*/
⟨await (arrive[i] == 0);⟩ /* ключевая строка — см. текст */
arrive[i] = 1;
⟨await (arrive[j] == 1);⟩
arrive[j] = 0;

/* код барьера для рабочего процесса W[j]*/
⟨await (arrive[j] == 0);⟩ /* ключевая строка — см. текст */
arrive[j] = 1;
⟨await (arrive[i] == 1);⟩
arrive[i] = 0;
```

Последние три строки каждого процесса исполняют роли, о которых было сказано выше. Первая же строка на первый взгляд может показаться лишней, поскольку в ней задано просто ожидание сброса собственного флага процесса. Однако она необходима, чтобы не допустить ситуацию, когда процесс успеет вернуться к барьеру и установить собственный флаг до того, как другой процесс сбросит флаг *на предыдущем использовании барьера*. Итак, чтобы программа соответствовала правилам синхронизации флагами (3.14), необходимы все четыре строки программы.

Теперь необходимо решить вопрос, как объединить экземпляры двухпроцессных барьеров, чтобы получить барьер для n процессов. В частности, нужно построить схему связей так, чтобы каждый процесс в конце концов знал о том, что остальные процессы достигли барьера. Лучше всего для этого подойдет некоторая бинарная схема соединений, размер которой пропорционален числу $\log_2 n$.

Пусть `Worker[1:n]` — массив процессов. Если n является степенью 2, то процессы можно объединить по схеме, показанной на рис. 3.2. Этот тип барьеров называется *барьером-бабочкой* (butterfly barrier) из-за схемы связей, аналогичной схеме соединений в преобразовании Фурье, которая похожа на бабочку. Как видно из рисунка, барьер-бабочка состоит из $\log_2 n$ уровней. На разных уровнях процесс синхронизируется с разными процессами. На уровне s процесс синхронизируется с процессом на расстоянии 2^{s-1}. Когда каждый процесс прошел через все уровни, до барьера дошли все процессы и могут быть продолжены. Дело в том, что каждый процесс прямо или косвенно синхронизируется со всеми остальными. (Если число n не является степенью 2, то барьер-бабочку можно построить, используя наименьшую степень 2, которая больше n. Отсутствующие процессы заменяются существующими, хотя это и не очень эффективно.)

Другая схема соединений показана на рис. 3.3. Она лучше, поскольку может быть использована при любых n (не только степенях 2). Здесь также несколько уровней, и на уровне s рабочий процесс синхронизируется с процессом на расстоянии 2^{s-1}. На каждом двухпроцессном барьере

процесс устанавливает флаг прибытия процесса справа от него (по модулю n) и ожидает установки собственного флага прибытия, а затем сбрасывает его. Эта структура называется *барьером с распространением*, поскольку основана на распространении информации среди n процессов за $\lceil \log_2 n \rceil$ циклов. Каждый процесс распространяет сообщение о своем прибытии к барьеру.

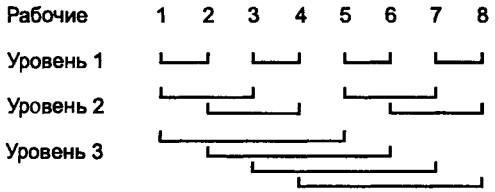


Рис. 3.2. Барьер-бабочка для восьми процессов

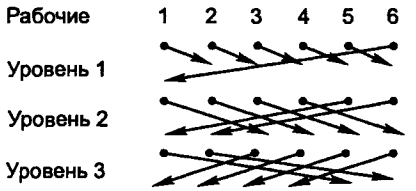


Рис. 3.3. Барьер с распространением для шести процессов

В реализации барьера для n процессов, независимо от используемой схемы соединений, важно избежать состояния гонок, которое может возникнуть при использовании нескольких экземпляров базового двухпроцессного барьера. Рассмотрим барьер-бабочку (см. рис. 3.2). Предположим, что есть только один массив переменных-флагов и они используются, как указано в (3.15). Пусть процесс 1 приходит к первому уровню и устанавливает флаг `arrive[1]`. Пусть процесс 2 — медленный, и еще не достиг первого уровня, а процессы 3 и 4 дошли до первого уровня барьера, установили флаги, сбросили их друг у друга и прошли на второй уровень. На втором уровне процесс 3 должен синхронизироваться с процессом 1, поэтому он ожидает установки флага `arrive[1]`. Флаг *уже* установлен, поэтому процесс 3 сбрасывает флаг `arrive[1]` и переходит на уровень 3, хотя этот флаг был установлен для процесса 2. Таким образом, в результате работы сети некоторые процессы пройдут барьер раньше, чем должны, а другие будут вечно ожидать перехода на следующий уровень. Та же проблема может возникнуть и при использовании барьера с распространением (см. рис. 3.3).

Описанные ошибки синхронизации являются следствием того, что используется только один набор флагов для каждого процесса. Для решения этой проблемы можно воспользоваться своим собственным набором флагов для каждого уровня барьера для n процессов, но лучше присваивать флагам больше значений.

Если флаги целочисленные, их можно использовать как возрастающие счетчики, которые запоминают количество уровней барьера, пройденных каждым процессом. Начальное значение каждого флага — 0. Каждый раз, когда рабочий процесс `i` приходит на новый уровень барьера, он увеличивает значение счетчика `arrive[i]`. Затем рабочий процесс `i` определяет номер процесса-партнера `j` на текущем уровне и ожидает, пока значение `arrive[j]` не станет, *как минимум*, таким же, как значение `arrive[i]`. Это описывается следующим кодом.

```
# код барьера для рабочего процесса i
for [s = 1 to num_stages] {
  arrive[i] = arrive[i] + 1;
  определить соседа j на уровне s
  while (arrive[j] < arrive[i]) skip;
}
```

Таким способом рабочий процесс `i` убеждается, что процесс `j` зашел, как минимум, так же далеко.

Описанный подход к использованию возрастающих счетчиков и уровней позволяет избежать состояния гонок, избавляет от необходимости для процесса ожидать переустановку собственного флага (первая строка кода (3.15)) и переустанавливать флаг другого процесса (последняя строка кода (3.15)). Таким образом, каждый уровень каждого барьера — это всего три строки кода. Единственный недостаток этого способа — счетчики все время возрастают, поэтому теоретически возможно их переполнение. Однако на практике это крайне маловероятно.

Подведем итог темы барьеров. Наиболее прост и удобен для небольшого числа процессов барьер-счетчик, но при условии, что существует неделимая инструкция "извлечь и сложить". Симметричный барьер наиболее эффективен для машин с разделяемой памятью, поскольку все процессы выполняют один и тот же код, а в идеальном случае — с одинаковой скоростью. (На машинах с распределенной памятью часто более эффективным является барьер с древовидной структурой, сокращающий взаимодействие между процессами.) При любой структуре барьера основная задача — избежать состояния гонок. Это достигается с помощью либо множества флагов (по одному на каждый уровень), либо возрастающих счетчиков.

3.5. Алгоритмы, параллельные по данным

В *алгоритмах, параллельных по данным*, несколько процессов выполняют один и тот же код и работают с разными частями разделяемых данных. Для синхронизации выполнения отдельных фаз процессов используются барьеры. Этот тип алгоритмов теснее всего связан с синхронными мультипроцессорами, или SIMD-машинами, т.е. машинами с одним потоком инструкций и многими потоками данных (single instruction, multiple data — SIMD). В SIMD-машинах аппаратно поддерживаются мелкомодульные вычисления и барьерная синхронизация. Однако алгоритмы, параллельные по данным, полезны и в асинхронных мультипроцессорных машинах при условии, что затраты на барьерную синхронизацию с лихвой компенсируются высокой степенью параллелизма процессов.

В данном разделе разработаны решения, параллельные по данным, для трех задач: частичное суммирование массива, поиск конца связанного списка и итерационный метод Якоби для решения дифференциальных уравнений в частных производных. Они иллюстрируют основные методы, используемые в алгоритмах, параллельных по данным, и барьерную синхронизацию. В конце раздела описаны многопроцессорные SIMD-машины и показано, как они помогают избежать взаимного влияния процессов и, следовательно, избавиться от необходимости программирования барьеров. Эффективной реализации алгоритмов, параллельных по данным, на машинах с разделяемой и распределенной памятью посвящена глава 11.

3.5.1. Параллельные префиксные вычисления

Часто бывает нужно применить некоторую операцию ко всем элементам массива. Например, чтобы вычислить среднее значение числового массива `a[n]`, нужно сначала сложить все элементы массива, а затем разделить сумму на `n`. Иногда нужно получить средние значения для всех префиксов `a[0:i]` массива. Для этого нужно вычислить суммы всех префиксов. Такой тип вычислений очень часто встречается, поэтому, например, в языке APL есть даже специальные операторы редукции ("сворачивания") *reduce* и просмотра *scan*. SIMD-машины с массовым параллелизмом вроде Connection Machine обеспечивают аппаратную реализацию операторов редукции для упаковки значений в сообщения.

В данном разделе показано, как параллельно вычисляются суммы всех префиксов массива. Эта операция называется *параллельным префиксным вычислением*. Базовый алгоритм может быть использован с любым ассоциативным бинарным оператором (сложение, умножение, логические операторы, вычисление максимума и другие). Поэтому параллельные префиксные вычисления используются во многих приложениях, включая обработку изображений, матричные вычисления и анализ регулярных языков (см. упражнения в конце главы).

Пусть дан массив `a[n]` и нужно вычислить `sum[n]`, где `sum[i]` означает сумму первых `i` элементов массива `a`. Очевидный способ последовательного решения этой задачи — пройти по элементам двух массивов.

```
sum[0] = a[0];
for [i = 1 to n-1]
  sum[i] = sum[i-1] + a[i];
```

На каждой итерации значение a[i] прибавляется к уже вычисленной сумме предыдущих i-1 элементов.

Теперь посмотрим, как этот алгоритм можно распараллелить. Если нужно просто найти сумму всех элементов, можно выполнить следующее. Сначала параллельно сложить пары элементов массива, например, складывать a[0] и a[1] синхронно с другими парами. После этого (тоже параллельно) объединить результаты первого шага, например, сложить сумму a[0] и a[1] с суммой a[2] и a[3] параллельно с вычислением других частичных сумм. Если этот процесс продолжить, то на каждом шаге количество просуммированных элементов будет удваиваться. Сумма всех элементов массива будет вычислена за $\lceil \log_2 n \rceil$ шагов. Это лучшее, что можно сделать, если элементы обрабатываются парами.

Для параллельного вычисления сумм всех префиксов можно адаптировать описанный метод удвоения числа обработанных элементов. Сначала присвоим всем элементам sum[i] значения a[i]. Затем параллельно сложим значения sum[i-1] и sum[i] для всех i >= 1, т.е. сложим все элементы, которые находятся на расстоянии 1. Теперь удвоим расстояние и сложим элементы sum[i-2] с sum[i], на этот раз для всех i >= 2. Если продолжать удваивать расстояние, то после $\lceil \log_2 n \rceil$ шагов будут вычислены все частичные суммы. Следующая таблица иллюстрирует шаги алгоритма для массива из шести элементов.

Начальные значения элементов a	1	2	3	4	5	6
Значения sum на расстоянии 1	1	3	5	7	9	11
Значения sum на расстоянии 2	1	3	6	10	14	18
Значения sum на расстоянии 4	1	3	6	10	15	21

В листинге 3.14 представлена реализация этого алгоритма. Каждый процесс сначала инициализирует один элемент массива sum, а затем циклически вычисляет частичные суммы. Процедура barrier(i), вызываемая в программе, реализует точку барьерной синхронизации, аргумент i — идентификатор вызывающего процедуру процесса. Выход из процедуры происходит, когда все n процессов выполнят команду barrier. В теле процедуры может быть использован один из алгоритмов, описанных в предыдущем разделе. (Для этой задачи барьеры можно оптимизировать, поскольку на каждом шаге синхронизируются только два процесса.)

Листинг 3.14. Вычисление частичных сумм элементов массива

```
int a[n], sum[n], old[n];

process Sum[i = 0 to n-1] {
  int d = 1;
  sum[i] = a[i];  /* инициализация элементов sum */
  barrier(i);
  ## SUM: sum[i] = (a[i-d+1] + ... + a[i])
  while (d < n) {
    old[i] = sum[i]; /* сохранить старое значение */
    barrier(i);
    if ((i-d) >= 0)
      sum[i] = old[i-d] + sum[i];
    barrier(i);
    d = d+d;  /* удвоить расстояние */
  }
}
```

Точки барьеров в программе 3.14 нужны для устранения взаимного влияния процессов. Например, все элементы массива sum должны быть проинициализированы до того, как какой-нибудь процесс обратится к ним. Также каждый процесс перед обновлением элемента

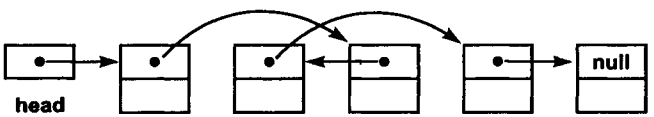
sum[i] должен сохранить копию его старого значения. Инвариант цикла *SUM* определяет, какая часть префикса массива a просуммирована на каждой итерации.

Как уже было отмечено, этот алгоритм можно изменить для использования с любым ассоциативным бинарным оператором. Для этого достаточно поменять оператор, преобразующий элементы массива sum. Выражение для комбинирования результатов записано в виде old[i-d] + sum[i], поэтому бинарный оператор не обязан быть коммутативным. Программу 3.14 можно адаптировать и для числа процессов меньше n; тогда каждый процесс будет отвечать за объединение частичных сумм полосы массива.

3.5.2. Операции со связанными списками

При работе со связанными структурами данных типа деревьев для поиска и вставки элементов за время, пропорциональное логарифму, часто используются сбалансированные бинарные деревья. Однако при использовании алгоритмов, параллельных по данным, многие операции даже с линейными списками можно реализовать за логарифмическое время. Покажем, как найти конец последовательно связанного списка. Этот же алгоритм можно использовать и для других операций над последовательно связанными списками, например, вычисления всех частичных сумм значений, вставки элемента в список приоритетов или поэлементного сравнения двух списков.

Предположим, что есть связанный список, содержащий не более n элементов. Связи хранятся в массиве link[n], а данные — в массиве data[n]. На начало списка указывает еще одна переменная, head. Если элемент i является частью списка, то или head == i, или link[j] == i для некоторого j от 0 до n-1. Поле link последнего элемента списка является указателем "в никуда" (пустым), что обозначается null. Предположим, что поля link элементов вне списка также пусты, а список уже инициализирован. Ниже приводится пример такого списка.



Задача состоит в том, чтобы найти конец списка. Стандартный последовательный алгоритм начинает работу с начала списка head и следует по ссылкам, пока не найдет пустой указатель. Последний из просмотренных элементов и есть конец списка. Время работы этого алгоритма пропорционально длине списка. Однако поиск конца списка можно выполнить за время, пропорциональное логарифму его длины, если использовать алгоритм, параллельный по данным, и метод удвоения из предыдущего раздела.

Каждому элементу списка назначается процесс Find. Пусть end[n] — разделяемый массив целых чисел. Если элемент i является частью списка, то задача процесса Find[i] — присвоить переменной end[i] значение, равное индексу последнего элемента списка, в противном случае процесс Find[i] должен присвоить end[i] значение null. Чтобы не рассматривать частные случаи, допустим, что список содержит хотя бы два элемента.

В начале работы каждый процесс присваивает элементу end[i] значение link[i], т.е. индекс следующего элемента списка (если он есть). Таким образом, массив end в начале работы воспроизводит схему связей списка. Затем процессы выполняют ряд этапов. На каждом этапе процесс рассматривает элемент с индексом end[end[i]]. Если элементы end[end[i]] и end[i] — не пустые указатели, то процесс присваивает элементу end[i] значение end[end[i]]. Таким образом, после первого цикла переменная end[i] будет указывать на элемент списка, находящийся на расстоянии в две связи от начального (если такой есть). После двух циклов значение end[i] будет указывать на элемент списка, удаленный на четыре связи (опять-таки, если он существует). После $\lceil \log_2 n \rceil$ циклов каждый процесс найдет конец списка.

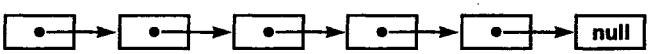
Листинг 3.15. Поиск конца последовательно связанного списка

```
int link[n], end[n];

process Find[i = 0 to n-1] {
  int new, d = 1;
  end[i] = link[i];  /* инициализация элементов end */
  barrier(i);
  ## FIND: end[i] == индекс конца списка на расстоянии
  ##       не более, чем 2^(d-1) связей от элемента i
  while (d < n) {
    new = null;  /* проверить, нужно ли обновить end[i] */
    if (end[i] != null and end[end[i]] != null)
      new = end[end[i]];
    barrier(i);
    if (new != null) /* обновить end[i] */
      end[i] = new;
    barrier(i);
    d = d+d;  /* удвоить расстояние */
  }
}
```

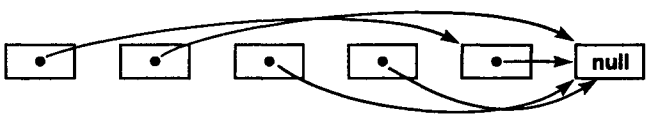
В листинге 3.15 представлена реализация этого алгоритма. Поскольку метод программирования тот же, что и для параллельных префиксных вычислений, структура алгоритма такая же, как в листинге 3.14. `barrier(i)` — это вызов процедуры, реализующей барьерную синхронизацию процесса `i`. Инвариант цикла *FIND* определяет, на что указывает элемент массива `end[i]` до и после каждой итерации. Если конец списка находится от элемента `i` на расстоянии не более 2^{d-1} связей, то в дальнейших итерациях значение `end[i]` не изменится.

Для иллюстрации работы алгоритма рассмотрим следующий список из шести элементов.

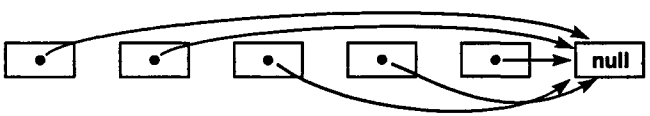


В начале цикла `while` процесса `Find` элементы массива `end` содержат такие же связи. После первой итерации в массиве `end` содержатся следующие связи.

Отметим, что ссылки двух последних элементов массива `end` далее не изменяются, поскольку они уже правильны. После второго цикла ссылки в элементах массива таковы.



После третьего (и последнего) цикла все элементы массива `end` достигли заключительного значения.



Как и при параллельных префиксных вычислениях, этот алгоритм можно адаптировать, чтобы использовать меньше, чем `n` процессов. Тогда каждый из них будет отвечать за вычисление некоторого подмножества ссылок `end`.

3.5.3. Сеточные вычисления: итерация Якоби

Многие задачи обработки изображений или научного моделирования решаются с помощью так называемых *сеточных вычислений*. Они основаны на использовании матрицы точек (сетки), наложенной на область пространства. В задаче обработки изображений матрица инициализируется значениями пикселей (точек), а целью является поиск чего-то вроде множества соседних пикселей с одинаковой интенсивностью. В научном моделировании часто используются приближенные решения дифференциальных уравнений в частных производных; в этом случае граничные элементы матрицы инициализируются краевыми условиями, а цель состоит в приближенном вычислении значений в каждой внутренней точке. (Это соответствует поиску устойчивого решения уравнения.) В любом случае, сеточные вычисления имеют следующую общую схему.

```
инициализировать матрицу;
while (еще не завершено) {
  для каждой точки вычислить новое значение;
  проверить условие завершения;
}
```

Обычно на каждой итерации новые значения точек вычисляются параллельно.

В качестве конкретного примера приведем простое решение уравнения Лапласа для двухмерного случая: $\Delta^2 = 0$. (Это дифференциальное уравнение в частных производных; подробности — в разделе 11.1.) Пусть `grid[n+1,n+1]` — матрица точек. Границы массива `grid` (левый и правый столбцы, верхняя и нижняя строки) представляют края двухмерной области. Сетке, наложенной на область, соответствуют `n×n` внутренних элементов массива `grid`. Задача — вычислить устойчивые значения внутренних точек. Для уравнения Лапласа можно использовать метод конечных разностей типа итераций Якоби. На каждой итерации новое значение каждой внутренней точки вычисляется как среднее значение четырех ее ближайших соседей.

В листинге 3.16 представлены сеточные вычисления для решения уравнения Лапласа с помощью итераций Якоби. Для синхронизации шагов вычислений вновь применяются барьеры. Каждая итерация состоит из двух основных шагов: обновление значений `newgrid` с проверкой на сходимость и перемещение содержимого массива `newgrid` в массив `grid`. Для того чтобы новые сеточные значения зависели только от старых, используются две матрицы. Вычисления можно закончить либо после фиксированного числа итераций, либо при достижении заданной точности, когда новые значения `newgrid` будут отличаться от значений `grid` не более, чем на `EPSILON`. Разности можно вычислять параллельно, но с последующим объединением результатов. Это можно сделать с помощью параллельных префиксных вычислений; решение оставляется читателю (см. упражнения в конце главы).

Листинг 3.16. Сеточные вычисления для решения уравнения Лапласа

```
real grid[n+1,n+1], newgrid[n+1,n+1];
bool converged = false;

process Grid[i = 1 to n, j = 1 to n] {
  while (не сошлось) {
    newgrid[i,j] = (grid[i-1,j] + grid[i+1,j]+
                    grid[i,j-1] + grid[i,j+1]) / 4;
    проверить, сошлось ли решение;
    barrier(i);
    grid[i,j] = newgrid[i,j];
    barrier(i);
  }
}
```

Алгоритм в листинге 3.16 правилен, но в некоторых отношениях слишком упрощен. Во-первых, массив `newgrid` копируется в массив `grid` на каждой итерации. Было бы намного эффективнее "развернуть" цикл, чтобы на каждой итерации сначала обновлялись значения, переходящие из `grid` в `newgrid`, а затем — из `newgrid` в `grid`. Во-вторых, лучше использовать алгоритм последовательной сверхрелаксации, сходящийся быстрее. В-третьих, программа в листинге 3.16 является слишком мелкомодульной для асинхронных мультипроцессоров. Поэтому гораздо лучше разделить сетку на блоки и каждому блоку назначить один процесс (и процессор). Все эти нюансы подробно рассмотрены в главе 11, где показано, как эффективно выполнять сеточные вычисления и на мультипроцессорах с разделяемой памятью, и на машинах с распределенной памятью.

3.5.4. Синхронные мультипроцессоры

В асинхронном мультипроцессоре все процессоры выполняют разные процессы с потенциально разными скоростями. Такие мультипроцессоры называются MIMD-машинами (multiple instruction — multiple data, много команд — много данных), поскольку имеют несколько потоков команд и данных, т.е. состоят из нескольких независимых процессоров. Обычно предполагается именно такая модель выполнения.

MIMD-машины являются наиболее гибкими мультипроцессорами, поэтому используются чаще других. Однако в последнее время стали доступными и синхронные мультипроцессоры (SIMD-машины), например, Connection Machine (начало 1990-х) или машины Maspar (середина–конец 1990-х). В SIMD-машине несколько потоков данных, но только один поток инструкций. Все процессоры *синхронно* выполняют одну и ту же последовательность команд. Это делает SIMD-машины особенно подходящими для алгоритмов, параллельных по данным. Например, алгоритм 3.14 вычисления всех частичных сумм массива для SIMD-машины упрощается следующим образом.

```
int a[n], sum[n];

process Sum[i = 0 to n-1] {
  sum[i] = a[i];  /* инициализация элементов массива SUM */
  while (d < n) {
    if ((i-d) >= 0)  /* обновить sum */
      sum[i] = sum[i-d] + sum[i];
    d = d + d;  /* удвоить расстояние */
  }
}
```

Программные барьеры не нужны, поскольку все процессы одновременно выполняют одни и те же команды, следовательно, каждая команда и обращение к памяти сопровождаются неявным барьером. Кроме того, для хранения старых значений не нужны дополнительные переменные. При присваивании значения элементу `sum[i]` *каждый* процесс(ор) извлекает из массива `sum` старые значения элементов перед тем, как присваивать новые. По этой причине параллельные инструкции присваивания на SIMD-машине становятся неделимыми, в результате чего исключаются некоторые источники взаимного влияния процессов.

Создать SIMD-машину с большим числом процессоров технологически намного проще, чем построить MIMD-машину с массовым параллелизмом. Это делает SIMD-машины привлекательными для решения больших задач, в которых можно использовать алгоритмы, параллельные по данным. С другой стороны, SIMD-машины являются специализированными, т.е. в любой момент времени вся машина выполняет одну программу. (Это основная причина, по которой интерес к SIMD-машинам невелик.) Кроме того, программисту нелегко все время загружать каждый процессор полезной работой. В приведенном выше алгоритме, например, все меньше и меньше процессоров на каждой итерации обновляют элементы `sum[i]`, но все они

должны вычислять значение условия в операторе `if`. Если условие не выполняется, то процесс приостанавливается, пока все остальные не обновят значения элементов массива `sum`. Таким образом, время выполнения оператора `if` — это *общее* время выполнения *всех* ветвей, даже если какая-то из них не затрагивается. Например, время выполнения оператора `if/then/else` на каждом процессоре — это сумма времени вычисления условия, выполнения `then`- или `else`-ветви.

3.6. Параллельные вычисления с портфелем задач

Как указано в предыдущем разделе, многие итерационные задачи могут быть решены с помощью программирования, параллельного по данным. Решение рекурсивных задач, основанное на принципе "разделяй и властвуй", можно распараллелить, выполняя рекурсивные вызовы параллельно, а не последовательно, как было показано в разделе 1.5.

В данном разделе представлен еще один способ реализации параллельных вычислений, в котором используется так называемый *портфель задач*. Задача является независимой единицей работы. Задачи помещаются в портфель, разделяемый несколькими рабочими процессами. Каждый рабочий процесс выполняет следующий основной код.

```
while (true) {
  получить задачу из портфеля;
  if (задач больше нет)
    break;     # выход их цикла while
  выполнить задачу, возможно, порождая новые задачи;
}
```

Этот подход можно использовать для реализации рекурсивного параллелизма; тогда задачи будут представлены рекурсивными вызовами. Его также можно использовать для решения итеративных проблем с фиксированным числом независимых задач.

Парадигма портфеля задач имеет несколько полезных свойств. Во-первых, она весьма проста в использовании. Достаточно определить представление задачи, реализовать портфель, запрограммировать выполнение задачи и выяснить, как распознается завершение работы алгоритма. Во-вторых, программы, использующие портфель задач, являются *масштабируемыми* в том смысле, что их можно использовать с любым числом процессоров; для этого достаточно просто изменить количество рабочих процессов. (Однако производительность программы при этом может и не изменяться.) И, наконец, эта парадигма упрощает реализацию *балансировки нагрузки*. Если длительности выполнения задач различны, то, вероятно, некоторые из задач будут выполняться дольше других. Но пока задач больше, чем рабочих процессов (в два–три раза), общие объемы вычислений, осуществляемых рабочими процессорами, будут примерно одинаковыми.

Ниже показано, как с помощью портфеля задач реализуется умножение матриц и адаптивная квадратура. При умножении матриц используется фиксированное число задач. В адаптивной квадратуре задачи создаются динамически. В обоих примерах для защиты доступа к портфелю задач применяются критические секции, а для обнаружения окончания — барьероподобная синхронизация.

3.6.1. Умножение матриц

Вновь рассмотрим задачу умножения матриц `a` и `b` размером `n` × `n`. Это требует вычисления `n2` промежуточных произведений, по одному на каждую комбинацию из строки `a` и столбца `b`. Каждое промежуточное умножение — это независимое вычисление, которое можно выполнить параллельно. Предположим, однако, что программа будет выполняться на машине с числом процессоров `PR`. Тогда желательно использовать `PR` рабочих процессов, по одному на каждый процессор. Чтобы сбалансировать вычислительную загрузку, процессы

должны вычислять примерно поровну промежуточных произведений. В разделе 1.4 каждому рабочему процессу часть вычислений назначалась статически. В данном случае воспользуемся портфелем задач, и каждый рабочий процесс будет захватывать задачу при необходимости.

Если число PR намного меньше, чем n, то подходящий для задания объем работы — одна или несколько строк результирующей матрицы c. (Это ведет к разумной локализации матриц a и c с учетом того, что данные в них хранятся по строкам.) Для простоты используем одиночные строки. В начальном состоянии портфель содержит n задач, по одной на строку. Задачи могут быть расположены в любом порядке, поэтому портфель можно представить простым перечислением строк.

```
int nextRow = 0;
```

Рабочий процесс получает задачу из портфеля, выполняя неделимое действие

```
⟨ row = nextRow; nextRow++; ⟩
```

Здесь row — локальная переменная. Портфель пуст, когда значение row не меньше n. Неделимое действие в указанной строке программы — это еще один пример вытягивания билета. Его можно реализовать с помощью инструкции "извлечь и сложить", если она доступна, или блокировок для защиты критической секции.

В листинге 3.17 представлена схема программы. Предполагается, что матрицы инициализированы. Рабочие процессы вычисляют внутренние произведения обычным способом. Программа завершается, когда все рабочие процессы выйдут из цикла while. Для определения этого момента можно воспользоваться разделяемым счетчиком done с нулевым начальным значением. Перед тем как рабочий процесс выполнит оператор break, он должен увеличить значение счетчика в неделимом действии. Если нужно, чтобы последний рабочий процесс выводил результаты, в конец кода каждого рабочего процесса можно добавить следующие строки.

```
if (done == n)
  напечатать матрицу c;
```

Здесь переменная done используется в качестве барьера-счетчика.

Листинг 3.17. Умножение матриц с помощью портфеля задач

```
int nextRow = 0; # портфель задач
double a[n,n], b[n,n], c[n,n];

process Worker[w = 1 to P] {
  int row;
  double sum; # для промежуточных произведений
  while (true) {
    # получить задачу
    ⟨ row = nextRow; nextRow++; ⟩
    if (row >= n)
      break;
    вычислить внутренние произведения для c[row,*];
  }
}
```

3.6.2. Адаптивная квадратура

Напомним, что задача квадратуры состоит в аппроксимации интеграла функции f(x) на промежутке от a до b. По алгоритму адаптивной квадратуры сначала вычисляется середина отрезка между a и b — точка m. Затем аппроксимируются три площади под графиком функции: от a до m, от m до b и от a до b. Если сумма двух меньших площадей с некоторой задан-

ной точностью равна большей, то приближение считается достаточно хорошим. Если нет, большая задача делится на две подзадачи, и процесс повторяется.

Парадигму портфеля задач можно использовать для реализации адаптивной квадратуры. Задача представляет собой отрезок для проверки; он определяется концами интервала, значениями функции в этих точках и приближением площади для этого интервала. Сначала есть одна задача — для всего отрезка от a до b.

Рабочий процесс циклически получает задачу из портфеля и выполняет ее. В отличие от программы умножения матриц, в данном случае портфель задач может быть (временно) пуст, и рабочий процесс вынужден задерживаться до получения задачи. Кроме того, выполнение одной задачи обычно приводит к возникновению двух меньших задач, которые рабочий процесс должен поместить в портфель. Наконец, здесь гораздо труднее определить, когда работа будет завершена, поскольку просто ждать опустошения портфеля недостаточно. (Действительно, как только будет получена первая задача, портфель *станет* пустым.) Вместо этого можно считать, что работа сделана, если портфель пуст и *каждый* рабочий процесс ждет получения новой задачи.

В листинге 3.18 показана программа для адаптивной квадратуры, использующая портфель задач. Он представлен очередью и счетчиком. Еще один счетчик отслеживает число простаивающих процессов. Вся работа заканчивается, когда значение переменной `size` равно нулю, а счетчика `idle` — n. Заметим, что программа содержит несколько неделимых действий. Они нужны для защиты критических секций, в которых происходит доступ к разделяемым переменным. Все неделимые действия, кроме одного, безусловны, поэтому их можно защитить блокировками. Однако оператор `await` нужно реализовать с помощью более сложного протокола, описанного в разделе 3.2, или более мощного механизма синхронизации типа семафоров или мониторов.

Программа в листинге 3.18 задает чрезмерное использование портфеля задач. В частности, решая создать две задачи, рабочий процесс помещает их в портфель, затем выполняет цикл и получает оттуда новую задачу (возможно, ту же, которую только что поместил туда). Вместо этого можно заставить рабочий процесс помещать одну задачу в портфель, а другую оставлять себе для выполнения. Когда портфель заполнится до состояния, обеспечивающего сбалансированную вычислительную загрузку, следовало бы заставить рабочий процесс выполнять задачу полностью, используя последовательную рекурсию, а не помещать новые задачи в портфель.

Листинг 3.18. Адаптивная квадратура с использованием портфеля задач

```
type task = (double left, right, fleft, fright, lrarea);
queue bag(task);     # портфель задач
int size;            # число задач в портфеле
int idle = 0;        # число простаивающих процессов
double total = 0.0; # общая площадь
вычислить аппроксимацию площади на отрезке от a до b;
добавить в портфель задачу (a, b, f(a), f(b), area);
count = 1;

process Worker[w = 1 to PR] {
  double left, right, fleft, fright, lrarea;
  double mid, fmid, larea, rarea;
  while (true) {
    # проверить завершение
    ⟨ idle++;
      if (idle == n && size == 0) break; ⟩
    # получить задачу из портфеля;
    ⟨ await (size > 0)
      удалить задачу из портфеля;
      size--; idle --; ⟩
    mid = (left+right) / 2;
    fmid = f(mid);
```

```
    larea = (fleft+fmid) * (mid-left) / 2;
    rarea = (fmid+fright) * (right-mid) / 2;
    if (abs((larea+rarea) - lrarea) > EPSILON) {
      ⟨ поместить в портфель (left, mid, fleft, fmid, larea);
        поместить в портфель(mid, right, fmid, fright, rarea);
       size = size + 2; ⟩
    } else
      ⟨ total = total + lrarea; ⟩
  }
  if (w == 1)    # рабочий процесс 1 выводит результат
    printf("общая площадь %f\n", total1);
}
```

Историческая справка

Впервые задача критической секции была описана Дейкстрой [Dijkstra, 1965]. Эту фундаментальную задачу изучали десятки людей, опубликовавших буквально сотни работ по данной теме. В данной главе были представлены четыре наиболее важных решения. (Рейнал [Raynal, 1986] написал целую книгу по алгоритмам взаимного исключения.) Хотя разработка решений с активным ожиданием вначале была чисто академическим упражнением (поскольку активное ожидание неэффективно в однопроцессорной системе), появление мультипроцессоров подстегнуло новую волну интереса к этой теме. В действительности все современные мультипроцессоры имеют команды, поддерживающие как минимум одно решение с активным ожиданием. Большинство этих команд описаны в упражнениях в конце главы.

В работе Дейкстры [1965] было представлено первое программное решение для n процессов. Это было расширение решения для двух процессов, разработанного голландским математиком Т. Деккером (см. упражнения). Однако в исходной формулировке Дейкстры не требовалось свойство возможности входа (3.5). Дональд Кнут [Knuth, 1966] стал первым, кто опубликовал решение, гарантирующее возможность входа.

Алгоритм разрыва узла был изобретен Г. Питерсоном [Peterson, 1981]; теперь его часто называют *алгоритмом Питерсона*. В отличие от ранних решений Дейкстры, Деккера и других авторов, этот алгоритм очень прост для двух процессов. Алгоритм Питерсона также легко обобщается для n-процессного решения (см. листинг 3.7). Это решение требует, чтобы процесс прошел через все n-1 уровней, даже если ни один другой процесс не делает попыток войти в критическую секцию. Блок и Ву [Block and Woo, 1990] представили вариант этого алгоритма, в котором необходимо только m уровней, если m процессов пытаются войти в критическую секцию (см. упражнения).

Алгоритм поликлиники был изобретен Лампортом [Lamport, 1974]. (В листинге 3.11 представлена его улучшенная версия из работы [Lamport, 1979].) Кроме того, что этот алгоритм нагляднее, чем предыдущие решения задачи критической секции, он позволяет процессам входить, по существу, в порядке FIFO (first in — first out, первым вошел — первым вышел). Он также обладает интересным свойством, которое делает его устойчивым против некоторых аппаратных сбоев. Во-первых, если один процесс считывает значение `turn[i]` в то время, как другой процесс присваивает этой переменной новое значение в протоколе входа, результатом операции чтения может стать любое число между 1 и записываемым значением. Во-вторых, процесс `CS[i]` может в любой момент дать сбой при условии, что он немедленно присвоит переменной `turn[i]` значение 0. Однако в решении Лампорта значение `turn[i]` может быть неограниченным, если в критической секции всегда находится хотя бы один процесс.

Алгоритм билета является простейшим из решений задачи критической секции для n процессов. Он позволяет процессам входить в критические секции в порядке получения номеров. Фактически алгоритм билета может рассматриваться как оптимизация алгоритма

поликлиники. Однако для его работы необходима инструкция "извлечь и сложить", которая есть лишь у немногих машин (такие машины описаны в книге [Almasi and Gottlieb, 1994]).

Гарри Джордан [Jordan, 1978] был одним из первых, кто понял важность барьерной синхронизации в параллельных итеративных алгоритмах; он считается первым, кто ввел термин "барьер". Хотя барьеры и не настолько распространены, как критические секции, их тоже изучали многие люди, разработавшие несколько различных реализаций. Барьер на основе объединяющего дерева (см. листинги 3.1 и 3.13) аналогичен разработке Ю, Тценга и Лоури [Yew, Tzeng, Lowrie, 1987]. Барьер-бабочка был построен Бруксом [Brooks, 1986]. Хенсген, Финкель и Манбер [Hensgen, Finkel, Manber, 1988] разработали барьер с распространением; в их работе также описывается турнирный барьер, аналогичный по структуре объединяющему дереву. Гупта [Gupta, 1989] описал "нечеткий барьер" ("fuzzy barrier"), включающий область операторов, которые процесс может выполнять в ожидании перехода через барьер. В работе Гупты также описана аппаратная реализация нечетких барьеров.

Как сказано в нескольких местах этой главы, реализации блокировок и барьеров с активным ожиданием могут вызвать конфликты обращения к памяти и соединительной сети. Поэтому важно, чтобы приостановленные процессы в циклах обращались к локальной памяти, например, кэшированным копиям переменных. В главе 8 книги [Hennessy and Patterson, 1996] по архитектуре обсуждаются вопросы синхронизации мультипроцессоров, представлен интересный исторический обзор и дается большое количество ссылок.

В прекрасной статье [Mellor-Crummy and Scott, 1991] анализируется быстродействие нескольких протоколов блокировок и барьеров, включая большинство описанных в данной книге. В этой работе также представлен новый протокол блокировки, основанный на связанном списке, и масштабируемый барьер на основе дерева, приводящий к активному ожиданию, в котором используется только локальная память. В статье [McKenney, 1996] описаны различные виды схем блокировки в параллельных программах и даны правила выбора схемы для конкретных ситуаций, а в [Savage et al., 1997] — инструмент под названием Eraser (Стиратель). Он может обнаруживать гонки по данным, которые вызваны ошибками синхронизации в многопоточных программах, основанных на блокировках.

В мультипрограммных (с квантованием по времени) системах производительность может существенно снизиться, если процессы прерываются во время удержания блокировки. Один из способов преодолеть эту проблему — использовать *неблокирующие* алгоритмы вместо блокировок со взаимным исключением. Реализация структуры данных является неблокирующей, если процесс завершает операцию за конечное число шагов, независимо от скорости выполнения других процессов. Эта идея и неблокирующая реализация параллельных очередей представлены в работе Херлихи и Уинга (Herlihy and Wing, 1990]. В их реализации операция вставки выполняется *без ожидания*, т.е. завершается за конечное число шагов, также независимо от скоростей выполнения. В статье [Herlihy, 1990] представлена общая методология реализации высоко параллельных структур данных, а в [Herlihy, 1991] — всестороннее обсуждение синхронизации без ожидания. В статье [Michael and Scott, 1998] оценивается быстродействие неблокирующих реализаций нескольких структур данных; там же представлены модификации реализаций, основанных на блокировках, безопасные при прерываниях. Авторы делают вывод, что неблокирующие реализации отлично подходят для очередей, а безопасные при прерываниях блокировки на мультипрограммных системах обгоняют по производительности обычные блокировки.

Алгоритмы, параллельные по данным, наиболее тесно связаны с SIMD-машинами, допускающими синхронную обработку тысяч элементов данных. Многие ранние примеры алгоритмов, параллельных по данным, были разработаны для машины NYU Ultracomputer, описанной в [Schwartz, 1980]; для всех алгоритмов была характерной логарифмическая зависимость времени работы от объема данных. Ultracomputer является MIMD-машиной; его разработчики осознавали всю важность эффективных критических секций и протоколов барьерной синхронизации. Они реализовали операцию "заменить и сложить" в раннем аппа-

ратном прототипе [Gottlieb et al., 1983]; эта инструкция прибавляет число к слову в памяти и возвращает результат. Вместо нее в более поздних версиях машины Ultracomputer появилась инструкция "извлечь и сложить", описанная в [Almasi and Gottlieb, 1994]. Она возвращает число, хранящееся в памяти до прибавления. Это изменение было сделано потому, что инструкцию "извлечь и сложить", в отличие от "заменить и сложить", можно обобщить на любой бинарный оператор — например, "извлечь и вычислить максимум" или "извлечь и логически умножить". Этот обобщенный вид инструкции получил название "извлечь и Ф".

Появление в середине 1980-х машины Connection Machine вызвало новую волну интереса к алгоритмам, параллельным по данным. Эта машина была разработана Хиллисом [Hillis, 1985] как часть его докторской диссертации в МТИ (MIT). (Как мало диссертаций имели такое влияние!) Connection Machine была SIMD-машиной, так что барьерная синхронизация автоматически обеспечивалась после каждой машинной команды. В этой машине были тысячи обрабатывающих элементов; это позволяло алгоритмам, параллельным по данным, иметь очень мелкомодульную структуру, например, отдельный процессор мог быть выделен для каждого элемента массива. В статье [Hillis and Steele, 1986] дан обзор Connection Machine и описаны интересные алгоритмы, параллельные по данным.

Парадигма портфеля задач для параллельных вычислений была предложена в статье [Carriero, Gelernter, Leichter, 1986]. В этой работе показано, как реализовать портфель задач в программной нотации Linda, которая будет описана в главе 7. Эту модель программирования они назвали *моделью тиражируемых рабочих* (replicated workers), поскольку портфель задач могли разделять сколько угодно рабочих процессов. Сейчас некоторые называют ее *моделью работ по найму* (work farm), поскольку для выполнения задач из портфеля "нанимаются" рабочие процессы. В данной книге предпочтение отдается названию "портфель задач", поскольку оно характеризует суть подхода — разделяемый портфель задач.

Литература

Almasi, G. S., and A. Gottlieb. 1994. *Highly Parallel Computing*, 2nd ed. Menlo Park. CA: Benjamin/Cummings.

Block, K., and T.-K. Woo. 1990. A more efficient generalization of Peterson's mutual exclusion algorithm. *Information Processing Letters* 35 (August): pp. 219–222.

Brooks, E. D., III. 1986. The butterfly barrier. *Int. Journal of Parallel Prog.* 15, 4 (August): pp. 295–307.

Carriero, N., D. Gelernter, and J. Leichter. 1986. Distributed data structures in Linda. *Thirteenth ACM Symp. on Principles of Prog. Langs.*, January, pp. 236–242.

Dijkstra, E. W. 1965. Solution of a problem in concurrent programming control. *Comm. ACM* 8, 9 (September): p. 569.

Gottlieb, A., B. D. Lubachevsky, and L. Rudolph. 1983. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. on Prog. Languages and Systems* 5, 2 (April): pp. 164–189.

Gupta, R. 1989. The fuzzy barrier: a mechanism for high speed synchronization of processors. *Third Int. Conf. on Architectural Support for Prog. Languages and Operating Systems*, April, pp. 54–63.

Hennessy, J. L., and D. A. Patterson. 1996. *Computer Architecture: A Quantitative Approach*. 2nd ed. San Francisco: Morgan Kaufmann.

Hensgen, D., R. Finkel, and U. Manber. 1988. Two algorithms for barrier synchronization. *Int. Journal of Parallel Prog.* 17. 1 (January): pp. 1–17.

Herlihy, M. P. 1990. A methodology for implementing highly concurrent data structures. *Proc. Second ACM Symp. on Principles & Practice of Parallel Prog.*, March, pp. 197–206.

Herlihy, M. P. 1991. Wait-free synchronization. *ACM Trans. on Prog. Languages and Systems* 11, 1 (January): pp. 124–149.

Herlihy, M. P., and J. M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Prog. Languages and Systems* 12, 3 (July): pp. 463–492.

Hillis, W. D. 1985. *The Connection Machine.* Cambridge, MA: MIT Press.

Hillis, W. D., and G. L. Steele, Jr. 1986. Data parallel algorithms. *Comm. ACM* 29, 12 (December): pp. 1170–1183.

Jordan, H. F. 1978. A special purpose architecture for finite element analysis. *Proc. 1978 Int. Conf. on Parallel Processing*, pp. 263–266.

Knuth, D. E. 1966. Additional comments on a problem in concurrent programming control. *Comm. ACM* 9, 5 (May): pp. 321, 322.

Lamport, L. 1974. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM* 17, 8 (August): pp. 453–455.

Lamport, L. 1979. A new approach to proving the correctness of multiprocess programs. *ACM Trans. on Prog. Languages and Systems* 1, 1 (July): pp. 84–97.

Lamport, L. 1987. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems* 5, 1 (February): pp. 1–11.

McKenney, P. E. 1996. Selecting locking primitives for parallel programming. *Comm. ACM* 39, 10 (October): pp. 75–82.

Mellor-Crummey, J. M., and M. L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems* 9, 1 (February): pp. 21–65.

Michael, M. M., and M. L. Scott. 1998. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computer* 51, pp. 1–26.

Peterson, G. L. 1981. Myths about the mutual exclusion problem. *Information Processing Letters* 12, 3 (June): pp. 115–116.

Raynal. M. 1986. *Algorithms for Mutual Exclusion.* Cambridge. MA: MIT Press.

Savage, S., M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM. Trans. on Computer Systems* 15, 4 (November): pp. 391–411.

Schwartz, J. T. 1980. Ultracomputers. *ACM Trans. on Prog. Languages and Systems* 2,4 (October): pp. 484–521.

Yew, P.-C., N.-F. Tzeng, and D. H. Lawrie. 1987. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. on Computers* C-36, 4 (April): pp. 388–395.

Упражнения

3.1. Ниже представлен алгоритм Деккера — первое решение задачи критической секции для двух процессов.

```
bool enter1 = false, enter2 = false;
int turn = 1;

process P1 {
  while (true) {
    enter1 = true;
    while (enter2)
      if (turn == 2) {
        enter1 = false;
        while (turn == 2) skip;
        enter1 = true;
      }
    критическая секция;
    enter1 = false; turn = 2;
    некритическая секция;
```

```
    }
  }

  process P2 {
    while (true) {
      enter2 = true;
      while (enter1)
        if (turn == 1) {
          enter2 = false;
          while (turn == 1) skip;
          enter2 = true;
        }
      критическая секция;
      enter2 = false; turn = 1;
      некритическая секция;
    }
  }
```

Объясните, как эта программа обеспечивает взаимное исключение и возможность входа, избегает взаимных блокировок и излишних задержек. В связи со свойством возможности входа укажите, сколько раз один процесс, пытающийся войти в критическую секцию, может пропускать другой процесс.

3.2. Предположим, компьютер имеет неделимые инструкции декремента DEC и инкремента INC, которые возвращают также знаковый бит результата. В частности, инструкция декремента задает такие действия.

```
DEC(var, sign):
  ⟨ var = var - 1;
    if (var >= 0) sign = 0; else sign = 1; ⟩
```

Инструкция INC аналогична, но в ней к переменной var прибавляется 1.

Используя инструкции DEC и/или DEC, постройте решение задачи критической секции для n процессов. Не беспокойтесь о свойстве возможности входа. Четко и ясно опишите, как ваше решение работает и почему является правильным.

3.3. Допустим, что компьютер имеет неделимую инструкцию swap, определяемую таким образом.

```
Swap(var1, var2):
  ⟨ tmp = var1; var1 = var2; var2 = tmp; ⟩
```

Переменная tmp является внутренним регистром:

а) разработайте решение задачи критической секции для n процессов, используя оператор Swap. Не беспокойтесь о свойстве возможности входа. Четко и ясно опишите, как ваше решение работает и почему является правильным;

б) измените свой ответ к пункту *а* так, чтобы программа хорошо работала на мультипроцессорной системе с кэш-памятью. Объясните, какие изменения сделаны (если они есть) и почему;

в) используя оператор Swap, разработайте справедливое решение задачи критической секции, обеспечивающее свойство возможности входа каждому ждущему процессу. Ключ к решению задачи — упорядочить процессы; наиболее очевидный порядок — первый пришедший обслуживается первым. Отметим, что просто использовать ответ к пункту *а* для реализации неделимого действия в алгоритме билета нельзя, поскольку нельзя получить справедливое решение из несправедливых компонентов. Можно предположить, что каждый процесс имеет уникальный идентификатор, ска-

жем, целое число от 1 до n. Объясните свое решение, аргументируйте его правильность и справедливость.

3.4. Предположим, что компьютер имеет неделимую инструкцию "сравнить и обменять", которая задает следующие действия.

```
CSW(a, b, c);
  ⟨ if (a == c)
      {c = b; return (0); }
  else
      {a = c; return (1); } ⟩
```

Параметры a, b и c являются простыми переменными типа целых чисел. Разработайте решение задачи критической секции для n процессов, используя инструкцию CSW. Не беспокойтесь о свойстве возможности входа. Четко и ясно опишите, как ваше решение работает и почему является правильным.

3.5. В некоторых RISC-машинах (reduced instruction set computer — компьютер с сокращенным набором команд) есть такие инструкции.

```
LL(register, variable)   # блокированная загрузка
  ⟨ register = variable; location = &variable; ⟩
SC(variable, value)      # условное сохранение
  ⟨ if (location == &variable)
        { variable = value; return (1); }
    else
        return (0); ⟩
```

Инструкция LL (load locked — блокированная загрузка) неделимым образом загружает значение variable в регистр register и сохраняет адрес variable в специальном регистре location. Он разделяется всеми процессорами и изменяется *только* в результате выполнения инструкций LL. Инструкция SC (store conditional — условное сохранение) неделимым образом проверяет, равен ли адрес переменной variable адресу в регистре location. Если равен, инструкция SC сохраняет значение value в variable и возвращает 1; иначе SC возвращает 0:

а) используя эти инструкции, разработайте решение задачи критической секции для n процессов. Не беспокойтесь о свойстве возможности входа;

б) достаточно ли мощны данные инструкции для разработки справедливого решения задачи критической секции? Если да, приведите решение. Если нет, объясните, почему.

3.6. Рассмотрим инструкции LL и SC, определенные в предыдущем упражнении. Их можно использовать для реализации неделимой инструкции "извлечь и сложить" (FA). Покажите, как это сделать. (*Указание.* Вам понадобится цикл активного ожидания.)

3.7. Рассмотрим следующий протокол критической секции [Lamport, 1987].

```
int lock = 0;

process CS[i = 1 to n] {
  while (true) {
    ⟨await (lock == 0)⟩; lock = i; Задержка;
    while (lock != i) {
      ⟨await (lock == 0)⟩; lock = i; Задержка;
    }
    критическая секция;
    lock = 0;
    некритическая секция;
  }
}
```

а) допустим, что код *Задержка* удален. Обеспечивает ли теперь протокол взаимное исключение? Исключается ли взаимная блокировка? Нет ли излишней задержки? Гарантирует ли такой протокол возможность входа? Тщательно обоснуйте каждый ответ;

б) пусть процессы выполняются действительно параллельно на мультипроцессоре. Предположим, что код *Задержка* зацикливается на время, достаточное для того, чтобы каждый процесс `i`, ожидающий снятия блокировки (`lock == 0`), успел выполнить оператор присваивания `lock = i`. Обеспечивает ли теперь протокол взаимное исключение, возможность входа, отсутствие взаимной блокировки и излишних задержек? Тщательно обоснуйте каждый ответ.

3.8. Допустим, что ваша машина имеет следующую неделимую инструкцию.

```
flip(lock)
  ⟨ lock = (lock + 1) % 2; # изменить признак блокировки
    return (lock); ⟩       # возвратить новое значение
```

Некто предложил такое решение задачи критической секции для *двух* процессов:

```
int lock = 0;  # разделяемая переменная

process CS[i = 1 to 2] {
  while (true) {
    while (flip(lock) != 1)
      while (lock != 0) skip;
    критическая секция;
    lock = 0;
    некритическая секция;
  }
}
```

а) объясните, почему это решение *не будет* работать, т.е. укажите порядок выполнения, в результате которого оба процесса одновременно окажутся в своих критических секциях;

б) допустим, что первую строку тела оператора `flip` изменили — теперь она выполняет сложение по модулю 3, а не по модулю 2. Будет ли теперь это решение работать для двух процессов? Объясните свой ответ.

3.9. Рассмотрим следующий вариант алгоритма разрыва узла для n процессов [Block and Woo, 1990]:

```
int in = 0, last[1:n];   # разделяемые переменные

process CS[i = 1 to n] {
  int stage;
  while (true) {
    ⟨in = in + 1;⟩ stage = 1; last[stage] = i;
    ⟨await (last[stage] != i or in <= stage);⟩
    while (last[stage] != i) { #переход на следующую стадию
      stage = stage + 1; last[stage] = i;
      ⟨await (last[stage] != i or in <= stage);⟩
    }
    критическая секция;
    ⟨in = in - 1;⟩
    некритическая секция;
  }
}
```

а) объясните, как эта программа гарантирует взаимное исключение, избегает взаимоблокировок и гарантирует возможность входа;

б) сравните производительность этого алгоритма и алгоритма разрыва узла (см. листинг 3.7). Какой из них быстрее, если в критическую секцию пытается войти только один процесс? Насколько быстрее? Какой из них быстрее, когда все n процессов пытаются войти в критическую секцию? Насколько быстрее?

в) преобразуйте крупномодульное решение, приведенное выше, в мелкомодульное, в котором неделимыми действиями являются только чтение и запись переменных. Операции инкремента и декремента неделимыми не считаются. (*Указание.* Заменить переменную `in` массивом.)

3.10. а) Измените крупномодульный алгоритм билета (см. листинг 3.8) так, чтобы переполнения переменных `next` и `number` не было; (*Указание.* Выберите константу MAX, которая больше n, и используйте модульную арифметику.)

б) пользуясь ответом к пункту *а*, измените мелкомодульный алгоритм билета (см. листинг 3.9) так, чтобы переполнения переменных `next` и `number` не было. Предполагается, что доступна инструкция "извлечь и сложить".

3.11. В алгоритме поликлиники (см. листинг 3.11) значения переменной `turn` не ограничены, если в критической секции постоянно находится хотя бы один процесс. Допустим, что чтение и запись неделимы. Можно ли изменить алгоритм так, чтобы значения `turn` всегда были ограниченными? Если да, предложите измененный алгоритм. Если нет, объясните, почему.

3.12. В протоколах критической секции, приведенных в тексте, все процессы выполняют один и тот же алгоритм. Задачу можно также решить с помощью управляющего процесса: когда обычный процесс `CS[i]` хочет войти в свою критическую секцию, он сообщает об этом управляющему и ждет допуска:

а) разработайте протоколы обычных и управляющего процессов. Не беспокойтесь о свойстве возможности входа; (*Указание.* За идеями обращайтесь к барьеру с управляющим процессом в листинге 3.12.)

б) измените ответ к пункту *а* так, чтобы обеспечить свойство возможности входа.

3.13. В программе (3.11) показано, как использовать разделяемый счетчик для барьерной синхронизации, но это решение не снимает проблему переустановки счетчика в 0. Разработайте полное решение, используя два счетчика. Сначала постройте крупномодульное решение с помощью операторов `await`. Затем разработайте мелкомодульное решение, предполагая, что доступна инструкция "извлечь и сложить". Будьте внимательны к тому, что процесс может возвратиться к барьеру до того, как его покинут все остальные процессы. (*Указание.* "Инкремент" в инструкции "извлечь и сложить" может быть отрицательным.)

3.14. *Турнирный барьер* имеет дерево с такой же структурой, как на рис. 3.1, но рабочие процессы взаимодействуют иначе, чем в барьере с объединяющим деревом (см. листинг 3.13). В частности, каждый рабочий процесс является листом дерева. Пары смежных рабочих процессов ждут прибытия друг друга. Один "побеждает" и проходит на следующий уровень дерева, а другой ожидает. Победитель "турнира" на вершине дерева объявляет, что все рабочие процессы достигли барьера, т.е. сообщает им, что можно продолжить выполнение:

а) напишите программы рабочих процессов со всеми деталями синхронизации. Допустим, что количество рабочих процессов n является степенью числа 2. Либо используйте два набора переменных, либо переустановите значения так, чтобы рабочие процессы смогли использовать турнирный барьер на следующем цикле итерации;

б) сравните свой ответ к пункту *а* с барьером на основе объединяющего дерева (см. листинг 3.13). Сколько переменных нужно для каждого типа барьера? Если каждое присваивание и оператор await занимают единицу времени, то сколько всего времени требует выполнение барьерной синхронизации в каждом алгоритме? Каким будет общее время для барьера с объединяющим деревом, если его изменить так, чтобы корень дерева рассылал одиночное сообщение continue (продолжать), как описано в тексте?

3.15. а) Опишите все детали барьера-бабочки для восьми процессов. Укажите все необходимые переменные, напишите код каждого выполняемого процесса. Барьер должен быть повторно используемым;

б) повторите пункт *а* для 8-процессного барьера с распространением;

в) сравните свои ответы к пунктам *а* и *б*. Сколько переменных нужно для каждого типа барьера? Если каждое присваивание и оператор await занимают единицу времени, то сколько всего времени требует выполнение барьерной синхронизации в каждом алгоритме?

г) повторите пункты *а*, *б* и *в* для 6-процессного барьера;

д) повторите пункты *а*, *б* и *в* для 14-процессного барьера.

3.16. Рассмотрим следующую реализацию n-процессного барьера:

```
int arrive[1:n] = ([n] 0);  # разделяемый массив
код, выполняемый процессом Worker[1]:
  arrive[1] = 1;
  ⟨await (arrive[n] == 1);⟩
код, выполняемый процессами Worker[i = 2 to n]:
  ⟨await (arrive[i-1] == 1);⟩
  arrive[i] = 1;
  ⟨await (arrive[n] == 1);⟩
```

а) объясните, как работает этот барьер;

б) какова временная сложность этого барьера?

в) расширьте код барьера, чтобы барьер был повторно используемым.

3.17. Предположим, что есть n рабочих процессов, пронумерованных от 1 до n, а машина обеспечивает выполнение неделимой инструкции инкремента. Рассмотрим следующий код n-процессного барьера, который должен быть повторно используемым.

```
int count = 0; go = 0;  # разделяемые переменные
код, выполняемый процессом Worker[1]:
  ⟨await (count == n-1);⟩
  count = 0;
  go = 1;
код, выполняемый процессами Worker[2:n]:
  ⟨count++;⟩
  ⟨await (go == 1);⟩
```

а) объясните, что ошибочно в этом коде;

б) исправьте код, чтобы он работал. Дополнительные разделяемые переменные использовать нельзя, но можно ввести локальные переменные;

в) допустим, что данный код правилен. Предположим, что все процессы прибыли к барьеру одновременно. Сколько времени пройдет, пока каждый процесс сможет покинуть барьер? Можно считать, что каждое присваивание и оператор await (после того, как условие становится истинным) занимают единицу времени.

3.18. В параллельном префиксном алгоритме (см. листинг 3.14) есть три точки барьерной синхронизации. Некоторые из них можно оптимизировать, поскольку не всегда для продолжения работы все процессы должны достигать барьера. Определите, какие барьеры можно оптимизировать, объясните детали оптимизации. Используйте как можно меньше двухпроцессных барьеров.

3.19. Измените каждый из следующих алгоритмов, чтобы использовать k процессов вместо n процессов (предполагается, что n кратно k):

а) параллельные префиксные вычисления (см. листинг 3.14);

б) вычисления со связанным списком (см. листинг 3.15);

в) сеточные вычисления (см. листинг 3.16).

3.20. Один из возможных способов сортировки n целых чисел состоит в использовании обменной сортировки "четный-нечетный" (или сортировки перестановкой четных и нечетных). Предположим, что есть n процессов `P[1:n]`, и n четно. При этом способе сортировки каждый процесс выполняет серию циклов. На нечетных циклах процессы `P[odd]` с нечетными номерами меняются значениями с `P[odd+1]`, если значения не упорядочены. На четных циклах процессы `P[even]` с четными номерами меняются значениями с `P[even+1]`, если значения не упорядочены (на четных циклах процессы `P[1]` и `P[n]` ничего не делают):

а) определите, сколько циклов нужно выполнить в худшем случае для сортировки n чисел. Напишите алгоритм, параллельный по данным, для сортировки целочисленного массива `a[1:n]` в возрастающем порядке;

б) измените ответ к пункту *а* так, чтобы программа завершалась, как только массив окажется отсортированным (это возможно уже в начальном состоянии);

в) измените ответ к пункту *а* так, чтобы использовались k процессов (n кратно k).

3.21. Предположим, что есть n процессов `P[1:n]`, и `P[1]` имеет некоторое локальное значение v, которое нужно передать остальным, сохранив в каждом элементе массива `a[1:n]`. Очевидный последовательный алгоритм требует линейного времени. Напишите алгоритм, параллельный по данным, которому достаточно логарифмического времени.

3.22. Допустим, что `P[1:n]` — массив процессов, а `b[1:n]` — разделяемый булев массив:

а) напишите алгоритм, параллельный по данным, для подсчета числа элементов `b[i]`, имеющих значение "истина";

б) предположим, что в пункте *а* получен ответ `count` между 0 и n. Напишите алгоритм, параллельный по данным, присваивающий уникальное целое значение от 1 до `count` каждому процессу `P[i]`, для которого `b[i]` истинно.

3.23. Пусть даны два последовательно связанных списка. Напишите алгоритм, параллельный по данным, который ставит в соответствие элементы с одинаковыми номерами в последовательностях. По завершении алгоритма эти элементы списков должны указывать друг на друга. Если один список длиннее другого, то лишние элементы более длинного списка должны содержать пустые указатели. Определите необходимые структуры. Не изменяйте исходные списки; вместо этого сохраните ответы в дополнительных массивах.

3.24. Пусть дан последовательно связанный список, элементы которого связаны в порядке возрастания их полей данных. Стандартный последовательный алгоритм вставки новых элементов в необходимое место имеет линейную оценку времени работы (в среднем должна быть просмотрена половина списка). Напишите алгоритм, параллельный по данным, для вставки нового элемента в список за логарифмическое время.

3.25. Рассмотрим простой язык для вычисления выражений со следующим синтаксисом:

выражение ::= операнд | выражение оператор операнд
операнд ::= идентификатор | число
*оператор ::= + | **

Идентификатор — это последовательность букв или цифр, начинающаяся с буквы, число — последовательность цифр, а оператор — знак + или *.

Дан массив символов `ch[1:n]`. Каждый символ — это буква, цифра, пробел, + или *. Последовательность символов от `ch[1]` до `ch[n]` представляет предложение на описанном выше языке.

Напишите алгоритм, параллельный по данным, который для каждого символа определяет токен (нетерминал), которому принадлежит символ. Можно предположить, что есть n процессов, по одному на символ. Результатом для каждого символа должен быть ответ: `ID`, `NUMBER`, `PLUS`, `TIMES` или `BLANK`. (*Указания.* Регулярный язык можно анализировать с помощью конечного автомата, представленного матрицей переходов. Строки матрицы проиндексированы состояниями, столбцы — символами. Значением элемента матрицы является новое состояние автомата, в которое он должен перейти, имея текущие состояние и входной символ. Композиция функций переходов ассоциативна, поэтому можно использовать параллельные префиксные вычисления.)

3.26. В обработке изображений возникает следующая задача выделения области. Дан массив целых чисел `image[1:n,1:n]`. Значение каждого элемента — интенсивность пикселя. Соседями пикселя являются четыре пикселя (слева, справа, снизу и сверху от него). Два соседних пикселя принадлежат одной области, если их значения равны. Таким образом, область — это максимальное множество пикселей, которые связаны (в смысле транзитивно-рефлексивного замыкания отношения соседства) и имеют одинаковые значения. Задача состоит в том, чтобы найти все области и присвоить всем пикселям каждой области уникальную (для области) метку. Точнее, пусть `label[1:n,1:n]` — еще одна матрица, и начальным значением элемента `label[i,j]` является `n*i + j`. Заключительное значение элемента `label[i,j]` должно быть максимальным из начальных меток области, которой принадлежит пиксель `[i,j]`. Опишите сеточные вычисления, параллельные по данным, для определения заключительных значений элементов `label`. Вычисления должны заканчиваться, если ни один из элементов `label` не изменяется.

3.27. Использование сеточных вычислений при решении задачи разметки областей из предыдущего упражнения в худшем случае требует времени выполнения $O(n^2)$. Это может произойти, например, если область "извивается" по всему изображению. Но даже для простых изображений сеточные вычисления требуют времени $O(n)$.

3.28. Задачу разметки областей можно решить за время $O(\log n)$. Во-первых, для каждого пикселя определим, находится ли он на границе области и, если да, то какие из его соседей тоже принадлежат границе. Во-вторых, для каждого граничного пикселя создадим указатели на его граничных соседей. Получим двунаправленные списки, соединяющие все точки границы области. В-третьих, используя списки, распространим максимальное из значений граничных пикселей на все остальные пиксели границы. (Пиксель с наибольшим значением для каждой области будет на ее границе.) И, наконец, для распространения метки каждой области на ее внутренние пиксели используем префиксные вычисления.

3.29. Напишите программу, параллельную по данным, реализующую этот алгоритм. Время ее выполнения должно быть $O(\log n)$.

3.30. Рассмотрим задачу порождения всех простых чисел до некоторого предела `L` с помощью модели "портфель задач". Один из способов решения состоит в имитации решета Эратосфена, в котором из массива целых чисел последовательно вычеркиваются числа, кратные простым: 2, 3, 5 и т.д. В этом случае портфель будет хранить следующее про-

стое число, которое нужно использовать. Этот метод легко программируется, но требует больших затрат памяти для хранения всего массива из L целых чисел.

3.31. Другой способ решения этой задачи состоит в проверке всех нечетных чисел одного за другим. В этом случае портфель будет содержать все нечетные числа. (Их хранить не нужно, поскольку из текущего числа легко получить следующее.) Для каждого кандидата проверяем, простое ли это число. Если да, добавим его в растущий список известных простых чисел. Этот список используется для проверки следующих кандидатов. Данный метод требует значительно меньше памяти, но возникает сложная проблема синхронизации.

3.32. Напишите параллельные программы реализации обоих алгоритмов. Используйте W рабочих процессов. В конце каждой программы выведите 10 последних найденных простых чисел и время работы вычислительной части программы. Сравните затраты времени и памяти в этих двух программах для разных значений L и W.

3.33. Рассмотрим задачу определения количества слов в словаре, буквы в которых не повторяются (уникальны). Прописные и строчные буквы не различаются. (В большинстве систем Unix есть один или несколько словарей, например в файле `/usr/dict/words`.) Напишите параллельную программу для решения этой задачи. Используйте модель "портфель задач" и W рабочих процессов. В конце программы выведите количество слов, состоящих из уникальных букв, и наиболее длинные из них. Перед началом параллельных вычислений словарь можно прочитать в разделяемые переменные.

3.34. Параллельная очередь — это очередь, в которой операции удаления и вставки могут выполняться параллельно. Пусть очередь хранится в массиве `queue[1:n]`. Две переменные, `front` и `rear`, указывают соответственно на первый заполненный элемент и на следующую пустую ячейку. Операция удаления задерживается до тех пор, пока в ячейке `queue[front]` не появится элемент, затем удаляет его и увеличивает значение `front` (по модулю n). Операция вставки приостанавливается, пока нет пустой ячейки, затем новый элемент помещается в ячейку `queue[rear]`, и значение `rear` увеличивается.

3.35. Разработайте алгоритмы для вставки в очередь и удаления из нее с максимальным параллелизмом. В частности, за исключением критических точек, где происходит обращение к разделяемым переменным, вставки и удаления элементов должны иметь возможность выполняться параллельно как по отношению друг к другу, так и с внутренним параллелизмом. Вам понадобятся дополнительные переменные. Можно предположить, что доступна инструкция "извлечь и сложить".

Глава 4

Семафоры

Как видно из предыдущей главы, большинство протоколов с активным ожиданием достаточно сложны. Кроме того, нет четкой грани между переменными для синхронизации и переменными для вычислений. Это усложняет разработку и использование протоколов с активным ожиданием.

Еще один недостаток активного ожидания — его неэффективность в большинстве многопоточных программ. Обычно количество процессов больше числа процессоров, за исключением синхронных параллельных программ, где на каждый процесс приходится по одному процессору, поэтому активное ожидание процесса становится эффективнее, если использовать процессор для выполнения другого процесса.

Синхронизация является основой параллельных программ, поэтому для разработки правильных протоколов синхронизации желательно иметь специальные средства, которые можно использовать для блокирования приостанавливаемых процессов. Первым таким средством синхронизации, не потерявшим актуальности и сегодня, стали *семафоры*. Они облегчают защиту критических секций и могут использоваться систематически для реализации планирования и сигнализации. По этой причине они включены во все известные автору библиотеки многопоточного и синхронного параллельного программирования. Кроме того, семафоры допускают различные способы реализации, как с помощью активного ожидания, описанного в предыдущей главе, так и с помощью ядра (см. главу 6).

Идея семафора в соответствии с названием взята из метода синхронизации движения поездов, принятого на железной дороге. Железнодорожный семафор — это "сигнальный флажок", показывающий, свободен путь впереди или занят другим поездом. По мере движения поезда семафоры устанавливаются и сбрасываются. Семафор остается установленным на время, достаточное, чтобы при необходимости остановить другой поезд. Таким образом, железнодорожные семафоры можно рассматривать как устройства, которые сигнализируют об условиях, чтобы обеспечить взаимоисключающее прохождение поездов по критическим участкам пути. Семафоры в параллельных программах аналогичны — они предоставляют базовый механизм сигнализации и используются для реализации взаимного исключения и условной синхронизации.

В этой главе определяется синтаксис и семантика семафоров, а также демонстрируется их применение для решения задач синхронизации. Для сравнения пересматриваются некоторые задачи, решенные в предыдущих главах, в том числе задачи критической секции, производителей и потребителей, а также барьера. Кроме того, представлены новые интересные задачи: об ограниченном (кольцевом) буфере, обедающих философах, читателях и писателях, распределении ресурсов по принципу "кратчайшая задача". По ходу изложения вводятся три полезных приема программирования: изменение переменных, использование разделенных двоичных семафоров и передача эстафеты (общий метод управления порядком выполнения процессов).

4.1. Синтаксис и семантика

Семафор — это особый тип разделяемой переменной, которая обрабатывается только двумя *неделимыми* операциями P и V. Семафор можно считать экземпляром класса семафор, операции P и V — методами этого класса с дополнительным атрибутом, определяющим их неделимость.

Значение семафора является *неотрицательным целым числом*. Операция V используется для сигнализации о том, что событие произошло, поэтому она увеличивает значение семафора. Операция P приостанавливает процесс до момента, когда произойдет некоторое событие, поэтому она, дождавшись, когда значение семафора станет положительным, уменьшает его.[9] Сила семафоров обусловлена тем, что выполнение операции P может быть приостановлено.

Семафор объявляется так:

```
sem s;
```

По умолчанию начальным значением является 0, но семафор можно инициализировать любым положительным значением, например:

```
sem lock = 1;
```

Массивы семафоров можно объявлять и при необходимости инициализировать обычным образом:

```
sem forks[5] = ([5] 1);
```

Если бы в этой декларации не было инициализации, то начальным значением каждого семафора в массиве `forks` был 0.

После объявления и инициализации семафор можно обрабатывать *только* с помощью операций P и V. Каждая из них является неделимым действием с одним аргументом. Пусть s — семафор. Тогда операции `P(s)` и `V(s)` определяются следующим образом.

```
P(s): ⟨await (s > 0) s = s - 1;⟩
V(s): ⟨s = s + 1;⟩
```

Операция V увеличивает значение s на единицу неделимым образом. Операция P уменьшает значение s, но, чтобы после вычитания значение s не стало отрицательным, она сначала ожидает, пока s не станет положительным.

Приостановка выполнения и вычитание в операции P являются *единым* неделимым действием. Предположим, что s — семафор с текущим значением 1. Если два процесса пытаются одновременно выполнить операцию `P(s)`, то это удастся сделать только одному из них. Но если один процесс пытается выполнить операцию `P(s)`, а другой — `V(s)`, то обе операции будут успешно выполнены в непредсказуемом порядке, а конечным значением семафора s станет 1.

Обычный семафор может принимать любые неотрицательные значения, *двоичный семафор* — только значения 1 или 0. Это значит, что операция V для двоичного семафора может быть выполнена, только когда его значение 0. (Операцию V для двоичного семафора можно определить как ожидание, пока значение семафора станет меньше 1, и затем его увеличение на 1.)

Поскольку операции с семафором определяются в терминах операторов `await`, их формальная семантика следует непосредственно из применения правила оператора `await` (см. раздел 2.6). Правила вывода для операций P и V получаются непосредственно из правила оператора `await`, примененного к конкретным операторам в определении P и V.

Свойства справедливости для операций с семафорами тоже следуют из их определения с помощью оператора `await`. Используем терминологию раздела 2.8. Если условие `s > 0` становится и далее остается истинным, выполнение операции `P(s)` завершится при справедливой в слабом смысле стратегии планирования. Если условие `s > 0` становится истинным бесконечно часто, то выполнение операции `P(s)` завершится при справедливой в сильном смысле стратегии планирования. Операция V для обычного семафора является безусловным неделимым действием, поэтому она завершится, если стратегия планирования безусловно справедлива.

Как будет показано в главе 6, при реализации семафоров обычно обеспечивается, что, если процессы приостанавливаются, выполняя операции P, они возобновляются в том же по-

[9] Буквы P и V взяты от голландских слов (это описано в исторической справке в конце главы). Можно считать, что буква P связана со словом "пропустить" (pass), а форма буквы V, расширяющаяся к верху, обозначает увеличение значения. Некоторые авторы вместо P и V используют названия `wait` и `signal`, но здесь эти команды оставлены для главы о мониторах.

рядке, в котором были приостановлены. Следовательно, процесс, ожидающий выполнения операции P, сможет в конце концов продолжить работу, если другие процессы выполнят соответствующее число операций V.

4.2. Основные задачи и методы

Семафоры непосредственно поддерживают реализацию взаимного исключения, необходимого, например, в задаче критической секции. Кроме того, они обеспечивают поддержку простых форм условной синхронизации, где они используются для сигнализации о событиях. Для решения более сложных задач эти два способа применения семафоров можно комбинировать.

В данном разделе иллюстрируется применение семафоров для взаимного исключения и условной синхронизации на примере решения четырех задач: критических секций, барьеров, производителей и потребителей, ограниченных (кольцевых) буферов. В решении последних двух задач применяется метод *разделенных двоичных семафоров*. В дальнейших разделах показано, как использовать представленные здесь методы для решения более сложных задач синхронизации.

4.2.1. Критические секции: взаимное исключение

Напомним, что в задаче критической секции каждый из n процессов многократно выполняет критическую секцию кода, в которой требуется исключительный доступ к некоторому разделяемому ресурсу, а затем некритическую секцию кода, в которой он работает только с локальными объектами. Каждому процессу, находящемуся в своей критической секции, нужен взаимоисключающий доступ к разделяемому ресурсу.

Семафоры были придуманы отчасти, чтобы облегчить решение задачи критической секции. В листинге 3.2 представлено решение, использующее переменные для блокировки, в котором переменная lock имеет значение "истина", когда ни один процесс не находится в своей критической секции, и значение "ложь" в противном случае. Пусть значение "истина" представлено как 1, а "ложь" — как 0. Тогда процесс перед входом в критическую секцию должен подождать, пока значение переменной lock не станет равным 1, и присвоить ей значение 0. Выходя из критической секции, процесс должен присвоить переменной lock значение 1.

Именно эти операции и поддерживаются семафорами. Пусть mutex — семафор с начальным значением 1. Выполнение операции P(mutex) — это то же, что и ожидание, пока значение переменной lock не станет равным 1, и последующее присвоение ей значения 0. Аналогично выполнение операции V(mutex) — это то же, что присвоение lock значения 1 (при условии, что это можно сделать, только когда она имеет значение 0). Эти рассуждения приводят к решению задачи критической секции, показанному в листинге 4.1.

Листинг 4.1. Решение задачи критической секции с использованием семафоров

```
sem mutex = 1;

process CS[i = 1 to n] {
  while (true) {
    P(mutex);
    критическая секция;
    V(mutex);
    некритическая секция;
  }
}
```

4.2.2. Барьеры: сигнализирующие события

В разделе 3.4 барьерная синхронизация была представлена в качестве средства синхронизации алгоритмов, параллельных по данным (раздел 3.5). В реализациях барьеров с активным ожиданием использованы переменные-флаги, которые устанавливаются приходящими к барьеру процессами и сбрасываются покидающими его. Как при решении задачи критической секции, семафоры облегчают реализацию барьерной синхронизации. Основная идея — использовать семафор в качестве флага синхронизации. Выполняя операцию V, процесс устанавливает флаг, а при операции P — ждет установки флага и сбрасывает его. (Если каждому процессу параллельной программы выделен собственный процессор, то задержки на барьерах должны быть реализованы с помощью циклов активного ожидания, а не блокировки процессов. Таким образом, желательна реализация семафоров с активным ожиданием.)

Вначале рассмотрим задачу реализации барьера для двух процессов. Напомним, что необходимо выполнить два требования. Во-первых, ни один процесс не должен перейти барьер, пока к нему не подошли оба процесса. Во-вторых, барьер должен допускать многократное использование, поскольку обычно одни и те же процессы синхронизируются после каждого этапа вычислений. Для решения задачи критической секции достаточно лишь одного семафора для блокировки, поскольку нужно просто определить, находится ли процесс в критической секции. Но при барьерной синхронизации необходимы два семафора в качестве сигналов, чтобы знать, приходит процесс к барьеру или уходит от него.

Сигнализирующий семафор s — это семафор с нулевым (как правило) начальным значением. Процесс сигнализирует о событии, выполняя операцию `V(s)`; другие процессы ожидают события, выполняя `P(s)`. Для двухпроцессного барьера два существенных события состоят в том, что процессы прибывают к барьеру. Следовательно, поставленную задачу можно решить с помощью двух семафоров `arrive1` и `arrive2`. Каждый процесс сообщает о своем прибытии к барьеру, выполняя операцию V для своего семафора, и затем ожидает прибытия другого процесса, выполняя для его семафора операцию P. Это решение приведено в листинге 4.2. Поскольку барьерная синхронизация симметрична, процессы действуют одинаково — каждый из них просто сигнализирует о прибытии и ожидает на других семафорах. Используемые таким образом семафоры похожи на флаговые переменные, поэтому их применение должно следовать принципам синхронизации флагами (3.14).

Листинг 4.2. Барьерная синхронизация с помощью семафоров

```
sem arrive1 = 0, arrive2 = 0;

process Worker1 {
  ...
  V(arrive1);      /* сигнал о прибытии */
  P(arrive2);      /* ожидание другого процесса */
  ...
}

process Worker2 {
  ...
  V(arrive2);      /* сигнал о прибытии */
  P(arrive1);      /* ожидание другого процесса */
  ...
}
```

С помощью приведенного в листинге 4.2 барьера можно реализовать барьер-бабочку для n процессов (см. рис. 3.2), или барьер с распространением (см. рис. 3.3). В обоих случаях понадобится массив семафоров `arrive`. На каждом этапе процесс i сначала сообщает о своем

прибытии, выполняя операцию V(arrive[i]), а затем ожидает прибытия остальных процессов, выполняя P для их элементов массива arrive. В отличие от ситуации с переменными-флагами здесь нужен только один массив семафоров arrive, поскольку действие операции V "запоминается", тогда как значение флаговой переменной может быть перезаписано.

Семафоры можно использовать и в качестве сигнальных флагов в реализации барьерной синхронизации для n процессов с управляющим процессом (см. листинг 3.12) или комбинирующим деревом (см. листинг 3.13). Операции V запоминаются, поэтому используется меньше семафоров, чем флаговых переменных. В управляющем процессе Coordinator (см. листинг 3.12), например, нужен всего один семафор.

4.2.3. Производители и потребители: разделенные двоичные семафоры

В данном разделе вновь рассматривается задача о производителях и потребителях, поставленная в разделе 1.6 и пересмотренная в разделе 2.5. Там предполагалось, что есть только один производитель и один потребитель. Здесь рассматривается общий случай, когда есть несколько производителей и несколько потребителей. Приводимое ниже решение демонстрирует еще одно применение семафоров в качестве сигнальных флагов и знакомит с важным понятием разделенного двоичного семафора, обеспечивающего еще один способ защиты критических секций кода.

В задаче о производителях и потребителях производители посылают сообщения, получаемые потребителями. Процессы общаются с помощью разделяемого буфера, управляемого двумя операциями: deposit (поместить) и fetch (извлечь). Выполняя операцию deposit, производители помещают сообщения в буфер; потребители получают сообщения с помощью операции fetch. Чтобы сообщения не перезаписывались и каждое из них могло быть получено только один раз, выполнение операций deposit и fetch должно чередоваться, причем первой должна быть deposit.

Запрограммировать необходимое чередование операций можно с помощью семафоров. Такие семафоры используются либо для сообщения о том, что процессы достигают критических точек выполнения, либо для отображения состояния разделяемых переменных. Здесь критические точки выполнения — это начало и окончание операций deposit и fetch. Соответствующие изменения разделяемого буфера состоят в том, что он заполняется или опустошается. Поскольку производителей и потребителей может быть много, проще связать семафор с каждым из двух возможных состояний буфера, а не с точками выполнения процессов.

Пусть empty (пустой) и full (полный) — два семафора, отображающие состояние буфера. В начальном состоянии буфер пуст, поэтому семафор empty имеет значение 1 (т.е. произошло событие "опустошить буфер"), а full — 0. Перед выполнением операции deposit производитель сначала ожидает опустошения буфера. Когда производитель помещает в буфер сообщение, буфер становится заполненным. И, наоборот, перед выполнением операции fetch потребитель сначала ожидает заполнения буфера, а затем опустошает его. Процесс ожидает события, выполняя операцию P для соответствующего семафора, и сообщает о событии, выполняя V. Полученное таким образом решение показано в листинге 4.3.

Переменные empty и full в листинге 4.3 являются двоичными семафорами. Вместе они образуют так называемый *разделенный двоичный семафор*, поскольку в любой момент времени только один из них может иметь значение 1. Термин "разделенный двоичный семафор" объясняется тем, что переменные empty и full могут рассматриваться как единый двоичный семафор, разделенный на две части. В общем случае разделенный двоичный семафор может быть образован любым числом двоичных семафоров.

Роль разделенных двоичных семафоров особенно важна в реализации взаимного исключения. Предположим, что один из двоичных семафоров инициализирован значением 1

(соответственно, остальные имеют значение 0). Допустим, что в процессах, использующих семафоры, *каждая* выполняемая ветвь начинается операцией P для одного из семафоров и заканчивается операцией V (для одного из семафоров). Тогда все операторы между P и ближайшей V выполняются со взаимным исключением, т.е. если процесс находится между операциями P и V, то все семафоры равны 0, и, следовательно, ни один процесс не сможет завершить операцию P, пока первый процесс не выполнит V.

Листинг 4.3. Производители и потребители, использующие семафоры

```
typeT buf;      /* буфер некоторого типа T */
sem empty = 1, full = 0;

process Producer[i = 1 to M] {
  while (true) {
    ...
    /* произвести данные и поместить их в буфер */
    P(empty);
    buf = data;
    V(full);
  }
}

process Consumer[j = 1 to N] {
  while (true) {
    /* извлечь результат и потребить его */
    P(full);
    result = buf;
    V(empty);
    ...
  }
}
```

Решение задачи производителей и потребителей (см. листинг 4.3) иллюстрирует именно такое применение семафоров. Каждый процесс-производитель `Producer` попеременно выполняет операции `P(empty)` и `V(full)`, а каждый процесс-потребитель `Consumer` — `P(full)` и `V(empty)`. В разделе 4.4 это свойство разделенного двоичного семафора будет использовано для создания общего метода реализации операторов `await`.

4.2.4. Кольцевые буферы: учет ресурсов

Из последнего примера видно, как синхронизировать доступ к одному буферу обмена. Если данные производятся и потребляются примерно с одинаковой частотой, то процессу не приходится долго ждать доступа к буферу. Однако обычно потребитель и производитель работают неравномерно. Например, производитель может быстро создать сразу несколько элементов, а затем долго вычислять до следующей серии элементов. В таких случаях увеличение емкости буфера может существенно повысить производительность программы, уменьшая число блокировок процессов. (Это пример классического противоречия между временем вычислений и объемом памяти.)

Здесь решается так называемая *задача о кольцевом буфере*, который используется в качестве многоэлементного коммуникационного буфера. Решение основано на решении задачи из предыдущего раздела. Оно также демонстрирует применение обычных семафоров в качестве счетчиков ресурсов.

Предположим пока, что есть только один производитель и только один потребитель. Производитель помещает сообщения в разделяемый буфер, потребитель извлекает их оттуда. Буфер содержит очередь уже помещенных, но еще не извлеченных сообщений. Эта очередь мо-

жет быть представлена связанным списком или массивом. Здесь используется массив, более простой для программирования. Итак, представим буфер массивом `buf[n]`, где `n > 1`. Пусть переменная `front` является индексом первого сообщения очереди, а `rear` — индексом первой пустой ячейки после сообщения в конце очереди. Вначале переменные `front` и `rear` имеют одинаковые значения, скажем, `0`.

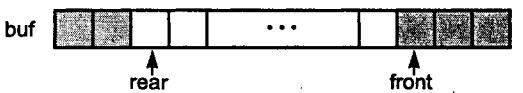
При таком представлении буфера производитель помещает в него сообщение со значением `data`, выполнив следующие действия:

```
buf[rear] = data; rear = (rear+1) % n;
```

Аналогично потребитель извлекает сообщение в свою локальную переменную `result`, выполняя действия:

```
result = buf[front]; front = (front+1) % n;
```

Оператор взятия остатка (`%`) используется для того, чтобы значения переменных `front` и `rear` всегда были в пределах от `0` до `n-1`. Очередь буферизованных сообщений хранится в ячейках от `buf[front]` до `buf[rear]` (не включительно). Переменная `buf` интерпретируется как кольцевой массив, в котором за `buf[n-1]` следует `buf[0]`. Вот пример конфигурации массива `buf`.



Затемненные ячейки заполнены, белые — пусты.

Если используется только один буфер (как в задаче "производитель–потребитель"), то выполнение операций `deposit` и `fetch` должно чередоваться. При наличии нескольких буферов операцию `deposit` можно выполнить, если есть пустая ячейка, а `fetch` — если сохранено хотя бы одно сообщение. Фактически, если есть пустая ячейка и сохраненное сообщение, операции `deposit` и `fetch` могут выполняться одновременно, поскольку обращаются к разным ячейкам и, следовательно, не влияют друг на друга. Однако требования синхронизации для одноэлементного и кольцевого буфера *одинаковы*. В частности, операции `P` и `V` применяются одним и тем же образом. Единственное отличие состоит в том, что семафор `empty` инициализируется значением `n`, а не `1`, поскольку в начальном состоянии есть `n` пустых ячеек. Решение показано в листинге 4.4.

Листинг 4.4. Кольцевой буфер с использованием семафоров

```
typeT buf[n]; /* массив некоторого типа T */
int front = 0, rear = 0;
sem empty = n, full = 0;  /*  n-2 <= empty+full <= n */

process Producer {
  while (true) {
    ...
    создать сообщение data;
    /* поместить data в буфер */
    P(empty);
    buf[rear] = data; rear = (rear+1) % n;
    V(full);
  }
}

process Consumer {
  while (true) {
    /* извлечь и потребить сообщение result */
    P(full);
    result = buf[front]; front = (front+1) % n;
```

```
    V(empty);
    ...
  }
}
```

В листинге 4.4 семафоры играют роль *счетчиков ресурсов*: каждый учитывает количество элементов ресурса: `empty` — число пустых ячеек буфера, а `full` — заполненных. Когда ни один процесс не выполняет `deposit` или `fetch`, сумма значений обоих семафоров равна общему числу ячеек `n`. Семафоры, учитывающие ресурсы, полезны в случаях, когда процессы конкурируют за доступ к таким многоэлементным ресурсам, как ячейки буфера или блоки памяти.

В программе (см. листинг 4.4) предполагалось, что есть только один производитель и один потребитель. Это гарантировало неделимое выполнение операций `deposit` и `fetch`. Теперь предположим, что есть несколько процессов-производителей. При наличии хотя бы двух свободных ячеек два из них могли бы выполнить операцию `deposit` одновременно. Но тогда оба попытались бы поместить свои сообщения в одну и ту же ячейку! (Если бы они присваивали новое значение ячейке `buf[rear]` до увеличения значения переменной `rear`.) Аналогично, если есть несколько потребителей, два из них одновременно могут выполнить `fetch` и получить одно и то же сообщение. Таким образом, `deposit` и `fetch` становятся критическими секциями. Одинаковые операции должны выполняться со взаимным исключением, но разные могут выполняться одновременно, поскольку при работе семафоров `empty` и `full` производители и потребители обращаются к разным ячейкам буфера. Необходимое исключение можно реализовать, используя решение задачи критической секции (см. листинг 4.1) с отдельными семафорами для защиты каждой критической секции. Законченное решение приведено в листинге 4.5.

Листинг 4.5. Несколько производителей и потребителей, использующих семафоры

```
typeT buf[n];      /* массив некоторого типа T */
int front = 0, rear = 0;
sem empty = n, full = 0;      /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1;   /* для взаимного исключения */

process Producer[i = 1 to M] {
  while (true) {
    ...
    создать сообщение data;
    /* поместить data в буфер */
    P(empty);
    P(mutexD);
    buf[rear] = data; rear = (rear+1) % n;
    V(mutexD);
    V(full);
  }
}

process Consumer[j = 1 to N] {
  while (true) {
    /* извлечь и потребить сообщение result */
    P(full);
    P(mutexF);
    result = buf[front]; front = (front+1) % n;
    V(mutexF);
    V(empty);
    ...
  }
}
```

Итак, отдельно решены две задачи синхронизации — сначала между одним производителем и одним потребителем, затем между несколькими производителями и несколькими потребителями. В результате оказалось легко объединить решения двух подзадач для получения полного решения задачи. Такая же идея будет использована в решении задачи о читателях и писателях в разделе 4.3. Таким образом, везде, где присутствуют несколько типов синхронизации, полезно реализовать их отдельно и затем объединить решения.

4.3. Задача об обедающих философах

В предыдущем разделе было показано, как использовать семафоры для решения задачи критической секции. В этом и следующем разделах на основе этого решения строится реализация выборочного взаимного исключения для двух классических задач: об обедающих философах и о читателях и писателях. Решение задачи об обедающих философах иллюстрирует реализацию взаимного исключения между процессами, конкурирующими за доступ к пересекающимся множествам разделяемых переменных, а читателей и писателей — реализацию комбинации параллельного и исключительного доступа к разделяемым переменным. В упражнениях есть дополнительные задачи выборочного взаимного исключения.

Хотя задача об обедающих философах скорее занимательная, чем практическая, она аналогична реальным проблемам, в которых процессу необходим одновременный доступ более, чем к одному ресурсу. Поэтому она часто используется для иллюстрации и сравнения различных механизмов синхронизации.

'4.1) **Задача об обедающих философах**. Пять философов сидят возле круглого стола. Они проводят жизнь, чередуя приемы пищи и размышления. В центре стола находится большое блюдо спагетти. Спагетти длинные и запутанные, философам тяжело управляться с ними, поэтому каждый из них, чтобы съесть порцию, должен пользоваться двумя вилками. К несчастью, философам дали всего пять вилок. Между каждой парой философов лежит одна вилка, поэтому они договорились, что каждый будет пользоваться только теми вилками, которые лежат рядом с ним (слева и справа). Задача — написать программу, моделирующую поведение философов. Программа должна избегать неудачной (и в итоге роковой) ситуации, в которой все философы голодны, но ни один из них не может взять обе вилки — например, когда каждый из них держит по одной вилке и не хочет отдавать ее.

Задача проиллюстрирована на рис. 4.1. Ясно, что два сидящих рядом философа не могут есть одновременно. Кроме того, раз вилок всего пять, одновременно могут есть не более, чем двое философов.

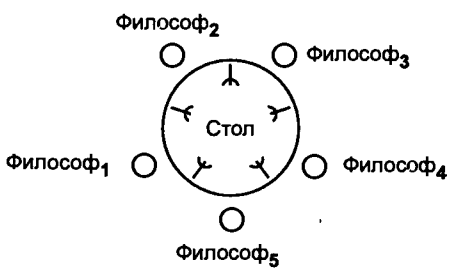


Рис. 4.1. Обедающие философы

Предположим, что периоды раздумий и приемов пищи различны — для их имитации в программе можно использовать генератор случайных чисел. Проимитируем поведение философов следующим образом.

```
process Philosopher[i = 0 to 4] {
  while (true) {
    поразмыслить;
    взять вилки;
    поесть;
    отдать вилки;
  }
}
```

Для решения задачи нужно запрограммировать операции "взять вилки" и "отдать (освободить) вилки". Вилки являются разделяемым ресурсом, поэтому сосредоточимся на их взятии и освобождении. (Можно решать эту задачу, отслеживая, едят ли философы; см. упражнения в конце главы.)

Каждая вилка похожа на блокировку критической секции: в любой момент времени владеть ею может только один философ. Следовательно, вилки можно представить массивом семафоров, инициализированных значением 1. Взятие вилки имитируется операцией P для соответствующего семафора, а освобождение — операцией V.

Процессы, по существу, идентичны, поэтому естественно предполагать, что они выполняют одинаковые действия. Например, каждый процесс может сначала взять левую вилку, потому правую. Однако это может привести ко взаимной блокировке процессов. Например, если все философы возьмут свои левые вилки, то они навсегда останутся в ожидании возможности взять правую вилку.

Необходимое условие взаимной блокировки — возможность кругового ожидания, т.е. когда один процесс ждет ресурс, занятый вторым процессом, который ждет ресурс, занятый третьим, и так далее до некоторого процесса, ожидающего ресурс, занятый первым процессом. Таким образом, чтобы избежать взаимной блокировки, достаточно обеспечить невозможность возникновения кругового ожидания. Для этого можно заставить один из процессов, скажем, Philosopher[4], сначала взять правую вилку. Это решение показано в листинге 4.6. Возможен также вариант решения, в котором философы с четным номером берут вилки в одном порядке, а с нечетным — в другом.

Листинг 4.6. Решение задачи об обедающих философах с использованием семафоров

```
sem fork[5] = {1, 1, 1, 1, 1};

process Philosopher[i = 0 to 3] {
  while (true) {
    P(fork[i]); P(fork[i+1]); #взять левую вилку, потом правую
    поесть;
    V(fork[i]); V(fork[i+1]);
    поразмыслить;
  }
}

process Philosopher[4] {
while (true) {
    P(fork[0]); P(fork[4]); #взять правую вилку, потом левую
    поесть;
    V(fork[0]); V(fork[4]);
    поразмыслить;
  }
}
```

4.4. Задача о читателях и писателях

Задача о читателях и писателях — это еще одна классическая задача синхронизации. Как и задачу об обедающих философах, ее часто используют для сравнения механизмов синхронизации. Она также весьма важна для практического применения.

(4.2) **Задача о читателях и писателях**. Базу данных разделяют два типа процессов — читатели и писатели. Читатели выполняют транзакции, которые просматривают записи базы данных, а транзакции писателей и просматривают, и изменяют записи. Предполагается, что вначале база данных находится в непротиворечивом состоянии (т.е. отношения между данными имеют смысл). Каждая отдельная транзакция переводит базу данных из одного непротиворечивого состояния в другое. Для предотвращения взаимного влияния транзакций процесс-писатель должен иметь исключительный доступ к базе данных. Если к базе данных не обращается ни один из процессов-писателей, то выполнять транзакции могут одновременно сколько угодно читателей.

Приведенное выше определение касается разделяемой базы данных, но ею может быть файл, связанный список, таблица и т.д.

Задача о читателях и писателях — это еще один пример выборочного взаимного исключения. В задаче об обедающих философах пары процессов конкурировали за доступ к вилкам. Здесь за доступ к базе данных соревнуются классы процессов. Процессы-читатели конкурируют с писателями, а отдельные процессы-писатели — между собой. Задача о читателях и писателях — это также пример задачи общей условной синхронизации: процессы-читатели должны ждать, пока к базе данных имеет доступ хотя бы один процесс-писатель; процессы-писатели должны ждать, пока к базе данных имеют доступ процессы-читатели или другой процесс-писатель.

В этом разделе представлены два различных решения задачи о читателях и писателях. В первом она рассматривается как задача взаимного исключения. Это решение является коротким, и его легко реализовать. Однако в нем читатели получают преимущество перед писателями (почему — сказано ниже), и его трудно модифицировать, например, для достижения справедливости. Во втором решении задача рассматривается как задача условной синхронизации. Это решение длиннее, оно кажется более сложным, но на самом деле его тоже легко реализовать. Более того, оно без труда изменяется для того, чтобы реализовать для читателей и писателей различные стратегии планирования. Важнее то, что во втором решении представлен мощный метод программирования, который называется *передачей эстафеты* и применим для решения *любой* задачи условной синхронизации.

4.4.1. Задача о читателях и писателях как задача исключения

Процессам-писателям нужен взаимоисключающий доступ к базе данных. Доступ процессов-читателей как группы также должен быть взаимоисключающим по отношению к любому процессу-писателю. Полезный для любой задачи избирательного взаимного исключения подход — вначале ввести дополнительные ограничения, реализовав больше исключений, чем требуется, а затем ослабить ограничения. Представим задачу как частный случай задачи критической секции. Очевидное дополнительное ограничение — обеспечить исключительный доступ к базе данных каждому читателю и писателю. Пусть переменная `rw` — это семафор взаимного исключения с начальным значением `1`. В результате получим решение с дополнительным ограничением (листинг 4.7).

Рассмотрим, как ослабить ограничения в программе листинга 4.7, чтобы процессы-читатели могли работать параллельно. Читатели как группа должны блокировать работу писателей, но только *первый* читатель должен захватить блокировку взаимного исключения путем, выполнив операцию `P(rw)`. Остальные читатели могут сразу обращаться к базе данных. Аналогично читатель, заканчивая работу, должен снимать блокировку, только если является последним активным процессом-читателем. Эти замечания приводят к решению, представленному в листинге 4.8.

Листинг 4.7. Решение задачи о читателях и писателях с дополнительным ограничением

```
sem rw = 1;

process Reader[i = 1 to M] {
  while (true) {
    ...
    P(rw);     # захватить блокировку исключительного доступа
    читать базу данных;
    V(rw);     # освободить блокировку
  }
}

process Writer[i = 1 to N] {
  while (true) {
    ...
    P(rw);     # захватить блокировку исключительного доступа
    записать в базу данных;
    V(rw);     # освободить блокировку
  }
}
```

Листинг 4.8. Схема решения задачи о читателях и писателях

```
int nr = 0;    # число активных читателей
sem rw = 1;    # блокировка для исключения читателей и писателей

process Reader[i = 1 to M] {
  while (true) {
    ...
    ⟨ nr = nr+1;
      if (nr == 1) P(rw);  # получить блокировку, если первый
    ⟩
    читать базу данных;
    ⟨ nr = nr-1;
      if (nr == 0) V(rw);  # снять блокировку, если последний
    ⟩
  }
}

process Writer[i = 1 to N] {
  while (true) {
    ...
    P(rw);
    записать в базу данных;
    V(rw);
  }
}
```

В листинге 4.8 переменная nr служит для подсчета числа активных читателей. В протоколе входа для процессов-читателей сначала значение переменной nr увеличивается на 1, затем проверяется, равно ли оно 1. Чтобы избежать взаимного влияния процессов-читателей, сложение и проверка должны выполняться как критическая секция, поэтому для обеспечения неделимого выполнения протокола входа читателей использованы угловые скобки. Аналогично

вычитание и проверка значения переменной nr в протоколе выхода должны выполняться неделимым образом, поэтому протокол выхода тоже заключен в угловые скобки.

Для уточнения схемы решения в листинге 4.8 до полного решения, использующего семафоры, нужно просто реализовать неделимые действия с помощью семафоров. Каждое действие является критической секцией, а реализация критических секций представлена в листинге 4.1. Пусть mutexR — семафор, обеспечивающий взаимное исключение процессов-читателей в законченном решении задачи о читателях и писателях (листинг 4.9). Отметим, что mutexR инициализируется значением 1, начало каждого неделимого действия реализовано операцией P(mutexR), а конец — операцией V(mutexR).

Листинг 4.9. Решение задачи о читателях и писателях, использующее семафоры

```
int nr = 0;      # число активных читателей
sem rw = 1;      # блокировка доступа к базе данных
sem mutexR = 1; # блокировка доступа читателей к nr

process Reader[i = 1 to m] {
  while (true) {
    ...
    P(mutexR);
      nr = nr+1;
      if (nr == 1) P(rw);  # получить блокировку, если первый
    V(mutexR);
    читать базу данных;
    P(mutexR);
      nr = nr-1;
      if (nr == 0) V(rw);  #снять блокировку, если последний
    V(mutexR);
  }
}

process Writer[i = 1 to n] {
  while (true) {
    ...
    P(rw);
    записать в базу данных;
    V(rw);
  }
}
```

Алгоритм в листинге 4.9 реализует решение задачи с *преимуществом читателей*. Если некоторый процесс-читатель обращается к базе данных, а другой читатель и писатель достигают протоколов входа, то новый читатель получает преимущество перед писателем. Следовательно, это решение не является справедливым, поскольку бесконечный поток процессов-читателей может постоянно блокировать доступ писателей к базе данных. Чтобы получить справедливое решение, программу в листинге 4.9 изменить весьма сложно (см. историческую справку), но далее будет разработано другое решение, которое легко преобразуется к справедливому.

4.4.2. Решение задачи о читателях и писателях с использованием условной синхронизации

Задача о читателях и писателях рассматривалась с точки зрения взаимного исключения. Целью было гарантировать, что писатели исключают друг друга, а читатели как класс — писателей. Решение (см. листинг 4.9) получено было в результате объединения решений для двух

задач критической секции: одно — для доступа к базе данных читателей и писателей, другое — для доступа читателей к переменной nr.

Построим другое решение поставленной задачи, начав с более простого определения необходимой синхронизации. В этом решении будет представлен общий метод программирования, который называется *передачей эстафеты* и использует разделенные двоичные семафоры как для исключения, так и для сигнализации приостановленным процессам. Метод передачи эстафеты можно применить для реализации любых операторов типа await и, таким образом, для реализации произвольной условной синхронизации. Этот метод можно также использовать для точного управления порядком, в котором возобновляются приостановленные процессы.

В соответствии с определением (4.2) процессы-читатели просматривают базу данных, а процессы-писатели и читают, и изменяют ее. Для сохранения непротиворечивости (целостности) базы данных писателям необходим исключительный доступ, но процессы-читатели могут работать параллельно в любом количестве. Простой способ описания такой синхронизации состоит в подсчете процессов каждого типа, которые обращаются к базе данных, и ограничении значений счетчиков. Например, пусть nr и nw — переменные с неотрицательными целыми значениями, хранящие соответственно число процессов-читателей и процессов-писателей, получивших доступ к базе данных. Нужно избегать плохих состояний, в которых значения обеих переменных положительны или значение nw больше 1:

BAD: (nr > 0 ∧ nw > 0) ∨ nw > 1

Дополняющее множество хороших состояний описывается отрицанием приведенного выше предиката, упрощенным до такого выражения:

RW: (nr == 0 ∨ nw == 0) ∧ nw <= 1

Первая часть формулы определяет, что читатели и писатели не могут получать доступ к базе данных одновременно. Вторая часть говорит, что не может быть больше одного активного писателя. В соответствии с этим описанием задачи схема основной части процесса-читателя выглядит так.

⟨nr = nr+1;⟩
читать базу данных;
⟨nr = nr-1;⟩

Соответствующая схема процесса-писателя такова.

⟨nw = nw+1;⟩
записать в базу данных;
⟨nw = nw-1;⟩

Чтобы уточнить эти схемы до крупномодульного решения, нужно защитить операции присваивания разделяемым переменным, гарантируя тем самым, что предикат *RW* является глобальным инвариантом. В процессах-читателях для этого необходимо защитить увеличение nr условием (nw == 0), поскольку при увеличении переменной nr значением nw должен быть 0. В процессах-писателях необходимо соблюдение условия (nr == 0 and nw == 0), поскольку при увеличении nw желательно нулевое значение как nr, так и nw. Однако в защите операций вычитания нет необходимости, поскольку никогда не нужно задерживать процесс, освобождающий ресурс. После вставки необходимых для защиты условий получим крупномодульное решение, показанное в листинге 4.10.

Листинг 4.10. Крупномодульное решение задачи о читателях и писателях

```
int nr = 0, nw = 0;
##RW: (nr == 0 ∨ nw == 0) ∧ nw <= 1

process Reader[i = 1 to M] {
  while (true) {
    ...
    ⟨await (nw == 0) nr = nr+1;⟩
```

```
    читать базу данных;
    ⟨nr = nr-1;⟩
  }
}

process Writer[i = 1 to N] {
  while (true) {
    ...
    ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
    записать в базу данных;
    ⟨nw = nw-1;⟩
  }
}
```

4.4.3. Метод передачи эстафеты

Иногда операторы `await` можно реализовать путем прямого использования семафоров или других элементарных операций, но в общем случае это невозможно. Рассмотрим два условия защиты операторов `await` в листинге 4.10. Эти условия перекрываются: условие защиты в протоколе входа писателя требует, чтобы и `nw`, и `nr` равнялись `0`, а в протоколе входа читателя — чтобы `nw` была равна `0`. Ни один семафор не может различить эти условия, поэтому для реализации таких операторов `await`, как указанный здесь, нужен общий метод. Представленный здесь метод называется *передачей эстафеты* (появление названия объяснено ниже). Этот метод достаточно мощен, чтобы реализовать любой оператор `await`.

В листинге 4.10 присутствуют четыре неделимых оператора. Первые два (в процессах читателя и писателя) имеют вид:

```
⟨await (B) S;⟩
```

Как обычно, `B` обозначает логическое выражение, а `S` — список операторов. Последние два неделимых оператора в обоих процессах имеют вид:

```
⟨S;⟩
```

Как было сказано в разделе 2.4, в первой форме может быть представлена *любая* условная синхронизация, а вторая форма является просто ее сокращением для частного случая, когда значение условия `B` неизменно и истинно. Следовательно, если знать, как с помощью семафоров реализуются операторы `await`, можно решить *любую* задачу условной синхронизации.

Для реализации операторов `await` из листинга 4.10 можно использовать разделенные двоичные семафоры. Пусть `e` — двоичный семафор с начальным значением `1`. Он будет применяться для управления входом (entry) в любое неделимое действие.

С каждым условием защиты `B` свяжем один семафор и один счетчик с нулевыми начальными значениями. Семафор будем использовать для приостановки процесса до момента, когда условие защиты станет истинным. В счетчике будет храниться число приостановленных процессов. В программе (см. листинг 4.10) есть два разных условия защиты, по одному в протоколах входа писателей и читателей, поэтому нужны два семафора и два счетчика. Пусть `r` — семафор, связанный с условием защиты в процессе-читателе, а `dr` — соответствующий ему счетчик приостановленных процессов-читателей. Аналогично пусть с условием защиты в процессе-писателе связаны семафор `w` и счетчик приостановленных процессов-писателей `dw`. Вначале нет ожидающих читателей и писателей, поэтому значения `r`, `dr`, `w` и `dw` равны `0`.

Использование трех семафоров (`e`, `r` и `w`) и двух счетчиков (`dr` и `dw`) описано в листинге 4.11. Комментарии поясняют, как реализованы крупномодульные неделимые действия из листинга 4.10. Для выхода из неделимых действий использован следующий код, помеченный *SIGNAL*.

```
    if (nw == 0 and dr > 0) {
      dr = dr-1; V(r);    # возобновить процесс-читатель, или
      }
    elseif (nr == 0 and nw == 0 and dw > 0) {
      dw = dw-1; V(w);    # возобновить процесс-писатель, или
      }
    else
      V(e);               # освободить блокировку входа
```

Роль кода *SIGNAL* — сигнализировать только *одному* из трех семафоров. Это значит, что если нет активных писателей, но есть приостановленный читатель, то он может быть продолжен с помощью операции `V(r)`. Если нет активных читателей или писателей, но есть приостановленный писатель, то он может быть продолжен благодаря операции `V(w)`. Иначе, если нет отложенных процессов, которые можно безопасно продолжить, то входной семафор получит сигнал с помощью операции `V(e)`.

Листинг 4.11. Схема читателей и писателей с передачей эстафеты

```
int nr = 0,  ## RW: (nr == 0 or nw == 0) and nw <= 1
    nw = 0;
sem e = 1,   # управляет входом в критические секции
    r = 0,   # используется для приостановки читателей
    w = 0;   # используется для приостановки писателей
             # всегда 0 <= (e+r+w) <= 1
int dr = 0;  # число приостановленных читателей
    dw = 0;  # число приостановленных писателей

process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0) { dr = dr+1; V(e); P(r); }
      nr = nr+1;
      SIGNAL;    # см. текст
    читать базу данных;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      SIGNAL;
  }
}

process Writer[i = 1 to N] {
  while (true) {
    # ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      SIGNAL;
    записать в базу данных;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      SIGNAL;
  }
}
```

Три семафора в листинге 4.11 образуют разделенный двоичный семафор, поскольку в любой момент времени только один из них может иметь значение 1, а все выполняемые ветви начинаются операциями P и заканчиваются операциями V. Следовательно, операторы между каждой парой P и V выполняются со взаимным исключением. Инвариант синхронизации *RW* является истинным в начале работы программы и перед каждой операцией V, так что он истинен, если один из семафоров имеет значение 1. Кроме того, при выполнении защищенного оператора истинно его условие защиты B, поскольку его проверил либо сам процесс и обнаружил, что оно истинно, либо семафор, который сигнализировал о продолжении приостановленного процесса, если только B истинно. Наконец, рассматриваемое преобразование кода не приводит ко взаимной блокировке, поскольку семафор задержки получает сигнал, только если некоторый процесс находится в состоянии ожидания или должен в него перейти. (Процесс может увеличить счетчик ожидающих процессов и выполнить операцию `V(e)`, но не может выполнить операцию P для семафора задержки.)

Описанный метод программирования называется *передачей эстафеты* из-за способа выработки сигналов семафорами. Когда процесс выполняется внутри критической секции, считается, что он получил эстафету, которая подтверждает его право на выполнение. Передача эстафеты происходит, когда процесс доходит до фрагмента программы *SIGNAL*. Если некоторый процесс ожидает условия, которое теперь стало истинным, эстафета передается одному из таких процессов, который в свою очередь выполняет критическую секцию и передает эстафету следующему процессу. Если ни один из процессов не ожидает условия, которое стало истинным, эстафета передается следующему процессу, который впервые пытается войти в критическую секцию, т.е. следующему процессу, выполняющему `P(e)`.

В листинге 4.11, как и в общем случае, многие экземпляры кода *SIGNAL* можно упростить или опустить. В процессе-читателе, например, перед выполнением первого экземпляра кода *SIGNAL*, т.е. в конце протокола входа процесса-читателя, nr больше нуля и nw равна нулю. Это значит, что фрагмент программы *SIGNAL* можно упростить:

```
if (dr > 0) { dr = dr-1; V(r); }
else V(e);
```

Перед вторым экземпляром кода *SIGNAL* в процессах-читателях обе переменные nw и dr равны нулю. В процессах-писателях nr равна нулю и nw больше нуля перед кодом *SIGNAL* в конце протокола входа писателя. Наконец, обе переменные nw и nr равны нулю перед последним экземпляром кода *SIGNAL* процесса-писателя. С помощью этих закономерностей упростим сигнальные протоколы и получим окончательное решение, использующее передачу эстафеты (листинг 4.12).

В этом варианте программы, если в момент завершения работы писателя несколько процессов-читателей отложены и один продолжает работу, то остальные возобновятся последовательно. Первый читатель увеличит nr и возобновит работу второго приостановленного процесса-читателя, который тоже увеличит nr и запустит третий процесс, и так далее. Эстафета передается от одного приостановленного процесса другому до тех пор, пока все они не возобновятся, т.е. значение переменной nr не станет равным 0. Последний оператор `if` процесса-писателя в листинге 4.12 сначала проверяет, есть ли приостановленные читатели, затем — есть ли приостановленные писатели. Порядок этих проверок можно свободно изменять, поскольку, если есть приостановленные процессы обоих типов, то после завершения протокола выхода писателя сигнал может получить любой из них.

4.4.4. Другие стратегии планирования

Решение задачи о читателях и писателях в листинге 4.12, конечно, длиннее, чем решение 4.8. Однако оно основано на многократном применении простого принципа — всегда передавать эстафету взаимного исключения только одному процессу. Оба решения дают преимущество чита-

Листинг 4.12. Решение задачи о читателях и писателях методом передачи эстафеты

```
int nr = 0,  ## RW: (nr == 0 or nw == 0) and nw <= 1
    nw = 0;
sem e = 1,   # управляет входом в критические секции
    r = 0,   # используется для приостановки читателей
    w = 0;   # используется для приостановки писателей
             # всегда 0 <= (e+r+w) <= 1
int dr = 0;  # число приостановленных читателей
    dw = 0;  # число приостановленных писателей

process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0) { dr = dr+1; V(e); P(r); }
      nr = nr+1;
      if (dr > 0) { dr = dr-1; V(r); }
      else V(e);
    читать базу данных;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      if (nr == 0 and dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}

process Writer[i = 1 to N] {
  while (true) {
    # ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      V(e);
    записать в базу данных;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      if (dr > 0) {dr = dr-1; V(r); }
      elseif (dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}
```

телям перед писателями. Но передачей эстафеты можно управлять, поэтому легко переделать решение 4.12 так, чтобы в нем использовались другие способы планирования. Например, чтобы предпочтение отдавалось писателям, необходимо обеспечить следующие условия:

- если ожидает писатель, то новые читатели должны приостанавливаться;
- приостановленный читатель продолжает работу, только если нет приостановленных писателей.

Первое условие можно выполнить, ужесточив условия приостановки в первом операторе `if` процессов-читателей:

```
if (nw > 0 or dw > 0) { dr = dr+1; V(e); P(r); }
```

Для выполнения второго условия изменим порядок первых двух ветвей оператора `if` процессов-писателей:

```
if (dw > 0) { dw = dw-1; V(w); }
elseif (dr > 0) {dr = dr-1; V(r); }
else V(e);
```

Теперь читатель может продолжить работу, только если нет ожидающих писателей; этот читатель, в свою очередь, может возобновить работу следующего читателя, и так далее. (Может появиться новый писатель, но, пока он не пройдет семафор входа, об этом не узнает ни один процесс.)

Ни одно из описанных выше преобразований не изменяет структуру программы. В этом заключается преимущество метода передачи эстафеты: для управления порядком запуска процессов можно изменять условия защиты, не влияя при этом на правильность решения.

При условии, что справедливы сами операции с семафорами, можно обеспечить справедливый доступ к базе данных, изменив программу 4.12. Например, когда в состоянии ожидания находятся и читатели и писатели, можно запускать их по очереди. Для этого нужно:

- если ожидает писатель, приостанавливать работу нового читателя;
- если ожидает читатель, приостанавливать работу нового писателя;
- когда заканчивает работу читатель, запускать один ожидающий процесс-писатель (если он есть);
- когда заканчивает работу писатель, запускать все ожидающие процессы-читатели (если они есть); иначе запускать один ожидающий процесс-писатель (если он есть).

Можно приостанавливать работу новых читателей и писателей, как показано выше. (Программа в листинге 4.12 удовлетворяет двум последним требованиям.) И здесь структура решения не изменяется.

Метод передачи эстафеты можно применить, чтобы управление порядком, в котором процессы используют ресурсы, сделать более мелкомодульным. Это демонстрируется в следующем разделе. Единственное, чем мы не можем управлять — это порядок запуска процессов, остановленных на входном семафоре, но это зависит от реализации семафоров.

4.5. Распределение ресурсов и планирование

Распределение ресурсов — это задача решения, когда процесс может получить доступ к ресурсу. В параллельных программах ресурсом является все то, при попытке получения чего работа процесса может быть приостановлена. Сюда включается вход в критическую секцию, доступ к базе данных, ячейка кольцевого буфера, область памяти, использование принтера и тому подобное. Несколько частных случаев задачи о распределении ресурсов были рассмотрены выше. В большинстве использовалась простейшая стратегия распределения ресурсов: если некоторый процесс ждет и ресурс свободен, то ресурс распределяется. Например, в решении задачи критической секции (раздел 4.2) разрешение на вход дается *какому-нибудь* из ожидающих процессов; попытка определить, *какой именно* процесс получит разрешение на вход, не делается. Так же и в задаче о кольцевом буфере (раздел 4.2) не было попытки контролировать порядок получения доступа производителей и потребителей к буферу. Лишь в задаче о читателях и писателях рассматривалась более сложная стратегия планирования. Но там целью было дать преимущество классу процессов, а не отдельным процессам.

В данном разделе показано, как реализовать общие стратегии распределения ресурсов и, в частности, как явно управлять выбором процесса, получающего ресурс, когда этого ожидают несколько процессов. Сначала описывается общая структура решения, затем реализация одной из стратегий распределения ресурсов — "кратчайшее задание". В решении использован метод передачи эстафеты и представлена идея частных семафоров, на основе которых решаются другие задачи о распределении ресурсов.

4.5.1. Постановка задачи и общая схема решения

В любой задаче распределения ресурсов процессы конкурируют за использование элементов разделяемого ресурса. Процесс запрашивает один или несколько элементов, выполняя операцию `request`, часто реализованную в виде процедуры. Параметры операции `request` указывают необходимое количество элементов ресурса, определяют особые характеристики (например, размер блока памяти) и идентифицируют запрашивающий процесс. Каждый элемент разделяемого ресурса либо свободен, либо занят (используется). Запрос может быть удовлетворен, только когда все необходимые элементы свободны. Таким образом, процедура `request` ожидает освобождения достаточного количества элементов ресурса, а затем возвращает запрошенное число элементов. После использования элементов ресурса процесс освобождает их операцией `release`. Параметры операции `release` задают идентификаторы возвращаемых элементов. Процесс может освобождать элементы в порядке и количествах, отличных от тех, в которых они запрашивались.

Если опустить представление элементов ресурса, то общая схема операций `request` и `release` такова.

```
request(параметры):
  ⟨await (запрос может быть удовлетворен) получить элементы;⟩
release(параметры):
  ⟨возвратить элементы;⟩
```

Операции должны быть неделимыми, поскольку каждой из них необходим доступ к представлению элементов ресурса. Поскольку в этом представлении используются переменные, отличающиеся от других переменных программы, операции будут неделимыми по отношению к другим действиям и, следовательно, могут выполняться параллельно с ними.

Эту общую схему решения можно реализовать с помощью метода передачи эстафеты (раздел 4.4). Операция `request` имеет вид обычного оператора `await`, поэтому реализуется следующим фрагментом программы.

```
request(параметры):
  P(e);
  if (запрос не может быть удовлетворен) DELAY;
  получить элементы;
  SIGNAL;
```

Операция `release` тоже имеет вид простого неделимого действия и реализуется таким фрагментом программы.

```
release(параметры):
  P(e);
  возвратить элементы;
  SIGNAL;
```

Как и раньше, семафор `e` управляет входом в критическую секцию, а фрагмент кода *SIGNAL* запускает на выполнение приостановленные процессы (если ожидающий запрос может быть удовлетворен) или снимает блокировку семафора входа с помощью операции `V(e)`. Код *DELAY* в операции `request` аналогичен фрагментам кода в начале протоколов входа процессов читателей и писателей (см. листинги 4.11 и 4.12). Он запоминает, что появился новый запрос, который должен быть приостановлен, снимает блокировку с семафора входа с помощью операции `V(e)` и блокирует запрашивающий процесс на семафоре задержки.

Детали реализации кода *SIGNAL* в каждой конкретной задаче распределения ресурсов зависят от условий задержки и их представления. В любом случае код *DELAY* должен сохранять параметры, характеризующие приостановленный запрос для дальнейшего их использования кодом *SIGNAL*. Кроме того, для каждого условия задержки нужен один семафор условия.

В следующем разделе разработано решение частной задачи распределения ресурсов, которое может служить основой для решения любой такой задачи. В упражнениях приводятся некоторые дополнительные задачи распределения ресурсов.

4.5.2. Распределение ресурсов по схеме "кратчайшее задание"

"Кратчайшее задание" (КЗ, Shortest-Job-Next — SJN) — это стратегия распределения ресурсов, которая встречается во многих разновидностях и используется для разных типов ресурсов. Предположим, что разделяемый ресурс состоит из одного элемента (общий случай нескольких элементов рассмотрен в конце данного раздела). Тогда стратегия КЗ определяется следующим образом.

(4.3) **Распределение ресурсов по схеме "кратчайшее задание".** Несколько процессов соперничают в использовании одного разделяемого ресурса. Процесс запрашивает ресурс, выполняя операцию `request(time,id)`, где целочисленный параметр `time` определяет длительность использования ресурса этим процессом, а целое число `id` идентифицирует запрашивающий процесс. Если в момент выполнения операции `request` ресурс свободен, он выделяется для процесса немедленно, иначе процесс приостанавливается. После использования ресурса процесс освобождает его, выполняя операцию `release`. Освобожденный ресурс распределяется приостановленному процессу (если такой есть) с наименьшим значением параметра `time`. Если у нескольких процессов значения `time` равны, то ресурс отдается тому из них, кто дольше всех ждал.

Стратегию КЗ можно использовать, например, для распределения процессоров (параметр `time` будет означать время выполнения), для вывода файлов на принтер (`time` — время печати) или для обслуживания удаленной передачи файлов по протоколу `ftp` (`time` — предполагаемое время передачи файла).

Стратегия КЗ привлекательна, поскольку минимизирует общие затраты времени на выполнение задачи. Вместе с тем, ей присуще несправедливое планирование: процесс может быть приостановлен навсегда, если существует непрерывный поток запросов с меньшим временем использования ресурса. (Такая несправедливость крайне маловероятна на практике, если только ресурс не перегружен.) Если несправедливость нежелательна, то можно слегка изменить стратегию КЗ, чтобы предпочтение отдавалось процессам, ожидающим слишком долго. Этот метод называется *выдержкой*, или *старением* (*aging*).

Если процесс выполняет запрос и ресурс свободен, то этот запрос может быть удовлетворен немедленно, поскольку других ожидающих запросов нет. Таким образом, стратегия КЗ "вступает в игру", только если есть несколько ожидающих запросов. Ресурс один, поэтому для хранения сведений о его доступности достаточно одной переменной. Пусть `free` — логическая переменная, которая истинна, когда ресурс доступен, и ложна, когда он занят. Для реализации стратегии КЗ нужно запоминать и упорядочивать ожидающие запросы. Пусть `pairs` — набор записей вида `(time, id)`, упорядоченных по значениям поля `time`. Если две записи имеют одинаковые значения поля `time`, то они остаются во множестве `pairs` в порядке их появления. В соответствии с этим определением следующий предикат должен быть глобальным инвариантом.

$$SJN\colon\ (\texttt{pairs} = \textit{упорядоченный набор}) \wedge (\texttt{free} \Rightarrow (\texttt{pairs} == \varnothing))$$

Расшифруем: набор `pairs` упорядочен, а если ресурс свободен, то `pairs` — пустое множество. Вначале `free` истинна, а множество `pairs` пусто, так что предикат *SJN* выполняется.

Без учета стратегии КЗ запрос может быть удовлетворен, как только ресурс станет доступным. Отсюда получим следующее крупномодульное решение.

```
bool free = true;    #разделяемая переменная
request(time,id): ⟨await (free) free = false;⟩
release(): ⟨free = true;⟩
```

Однако при использовании стратегии КЗ процесс, выполнивший запрос `request`, должен быть приостановлен до момента, когда будет свободен ресурс *и* запрос процесса станет следующим с точки зрения стратегии КЗ. Из второй части предиката *SJN* следует, что если переменная `free` истинна во время выполнения процессом операции `request`, то множество `pairs` пусто. Следовательно, приведенного выше условия достаточно для определения, может ли запрос удовлетворяться немедленно. Параметр `time` играет роль, только если запрос должен быть отложен — т.е. если переменная `free` ложна. На основе этих соображений можно реализовать операцию `request` таким образом.

```
request(time,id):
  P(e);
  if (!free) DELAY;
  free = false;
  SIGNAL;
```

Соответственно, операция `release` реализуется следующим образом.

```
release():
  P(e);
  free = true;
  SIGNAL;
```

В операции `request` предполагается, что операции `P` над семафором входа `e` обслуживаются в порядке их появления, т.е. по правилу "первым пришел — первым обслужен". Если этого нет, то порядок обработки запросов может не соответствовать стратегии КЗ.

Осталось воплотить стратегию распределения ресурсов КЗ. Для этого используем множество `pairs` и семафоры, реализующие фрагменты кода *SIGNAL* и *DELAY*. Если запрос не может быть удовлетворен, его следует сохранить, чтобы к нему можно было вернуться после освобождения ресурса. Таким образом, во фрагменте кода *DELAY* процесс должен:

- вставить параметры запроса в набор `pairs`;
- освободить управление критической секцией, выполнив операцию `V(e)`;
- остановиться на семафоре до удовлетворения запроса.

Если после освобождения ресурса множество `pairs` не пусто, то в соответствии со стратегией КЗ только один процесс должен получить ресурс. Таким образом, если есть приостановленный процесс, который теперь может продолжить работу, он должен получить сигнал с помощью операции `V` для семафора задержки.

В приведенных выше примерах условий задержки было немного, поэтому нужно было всего несколько семафоров условия. Например, в решении задачи о читателях и писателях в конце предыдущего раздела было только два условия задержки. Но здесь у каждого процесса есть свое условие задержки, которое зависит от его позиции в наборе `pairs`: первый в `pairs` процесс должен быть запущен перед вторым и так далее. Таким образом, каждый процесс должен ожидать на своем семафоре задержки.

Предположим, что ресурс используют `n` процессов. Пусть `b[n]` — массив семафоров, каждый элемент которого имеет начальное значение `0`. Будем считать, что значения идентификаторов процессов `id` уникальны и находятся в пределах от `0` до `n-1`. Тогда процесс `id` приостанавливается на семафоре `b[id]`. Дополнив операции `request` и `release` соответствующей обработкой множества `pairs` и массива `b`, получим решение задачи распределения ресурсов по схеме КЗ (листинг 4.13).

В листинге 4.13 предполагается, что операция вставки помещает пару параметров в такое место последовательности `pairs`, которое сохраняет истинность первой части предиката *SJN*. Следовательно, предикат *SJN* действительно является инвариантом вне операций `request` и `release`, т.е. он является истинным непосредственно после каждой операции `P(e)` и перед каждой `V(e)`. Если есть ожидающие запросы, т.е. множество `pairs` не пусто, оператор `if` кода выработки сигнала в операции `release` запускает только один процесс.

"Эстафетная палочка" переходит к этому процессу, а он присваивает переменной free значение ложь. Этим гарантируется истинность второй части предиката *SJN*, когда множество pairs не пусто. Поскольку ресурс один, остальные запросы не могут быть удовлетворены, так что сигнальный код операции request состоит из одной операции V(e).

Листинг 4.13. Решение задачи распределения ресурсов по схеме "кратчайшее задание" с использованием семафоров

```
bool free = true;
sem e = 1, b[n] = ([n] 0); # для входа и задержки
typedef Pairs = set of (int, int);
Pairs pairs = ∅;
## SJN: (pairs — упорядоченный набор) ∧ (free ⇒ (pairs == ∅))
request(time,id):
  P(e);
  if (!free) {
    вставить(time,id) в pairs;
    V(e);          # снять блокировку входа
    P(b[id]);      # ожидать возобновления
  }
  free = false;
  V(e);       # наилучшее решение, поскольку free == false;
release():
  P(e);
  free = true;
  if (P != ∅) {
    удалить первую пару (time,id) из pairs;
    V(b[id]);      # передать эстафету процессу id
  }
  else V(e);
```

Элементы массива семафоров b в листинге 4.13 являются примером так называемых частных семафоров.

(4.4) **Частный семафор.** Семафор s называется *частным*, если операцию P над ним выполняет только один процесс.

Когда процесс в листинге 4.13 должен быть приостановлен, он выполняет операцию P(b[id]) для блокировки на собственном элементе массива b.

Частные семафоры полезны в ситуациях, когда необходимо иметь возможность сигнализировать отдельному процессу. В некоторых задачах распределения ресурсов, однако, условий задержки может быть меньше, чем процессов, претендующих на использование ресурса. Тогда эффективнее использовать один семафор для каждого условия, а не для каждого процесса. Например, если память выделяется блоками нескольких заданных размеров, и не важен порядок распределения блоков процессам, конкурирующим за блоки одного размера, то достаточно использовать по одному семафору задержки для каждого возможного размера блока.

Решение в листинге 4.13 легко обобщить для использования ресурсов, состоящих из нескольких элементов. В этой ситуации каждый элемент может быть свободен или занят, а операции request и release должны использовать параметр amount, определяющий требуемое количество элементов ресурса. Изменим решение в листинге 4.13 следующим образом:

- булеву переменную free заменим целочисленной переменной avail для хранения количества доступных элементов;
- в операции request проверим условие amount <= avail. Если оно истинно, выделим amount элементов ресурса. Если нет, запомним, сколько элементов ресурса запрошены перед приостановкой процесса;

- в операции release увеличим значение avail на amount, после чего определим, можно ли удовлетворить запрос самого старого из отложенных процессов с минимальным значением параметра time. Если да — запустим его. Если нет, выполним V(e).

После освобождения элементов ресурса возможно удовлетворение нескольких ожидающих запросов. Например, может быть два приостановленных процесса, которым в сумме нужно не больше элементов, чем было освобождено. Тогда первый процесс после получения необходимого ему количества элементов должен выработать сигнал для второго процесса. Таким образом, протокол выработки сигнала в конце операции request должен быть таким же, как и протокол в конце операции release.

4.6. Учебные примеры: библиотека Pthreads

Напомним, что *поток* — это "облегченный" процесс, т.е. процесс с собственным программным счетчиком и стеком выполнения, но без "тяжелого" контекста (типа таблиц страниц), связанного с работой приложения. Некоторые операционные системы уже давно предоставляли механизмы, позволявшие программистам писать многопоточные приложения. Но эти механизмы отличались, поэтому приложения нельзя было переносить между разными операционными системами или даже между разными типами одной операционной системы. Чтобы исправить эту ситуацию, большая группа людей в середине 1990-х годов определила стандартный набор функций языка C для многопоточного программирования. Эта группа работала под покровительством организации POSIX (Portable Operating System Interface — интерфейс переносимой операционной системы), поэтому библиотека называется Pthreads для потоков POSIX. Сейчас эта библиотека широко распространена и доступна на самых разных типах операционных систем UNIX и некоторых других системах.

Библиотека Pthreads содержит много функций для управления потоками и их синхронизации. Здесь описан только базовый набор функций, достаточный для создания и соединения потоков, а также для синхронизации работы потоков с помощью семафоров. (В разделе 5.5 описаны функции для блокировки и переменных условия.) Также представлен простой, но достаточно полный пример приложения типа "производитель-потребитель". Он может служить базовым шаблоном для других приложений, в которых используется библиотека Pthreads.

4.6.1. Создание потока

Для использования библиотеки Pthreads в программе на языке C нужно выполнить четыре действия. Первое — подключить стандартный заголовочный файл библиотеки Pthreads:

```
# include <pthread.h>
```

Второе — объявить переменные для одного дескриптора атрибутов потока и одного или нескольких дескрипторов потока:

```
pthread_attr_t tattr;   /* атрибуты потока */
pthread_t tid;          /* дескриптор потока */
```

Третье — инициализировать атрибуты, выполнив функции:

```
pthread_attr_init(&tattr);
pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
```

И четвертое — создать потоки, как это описано ниже.

Перед созданием потока устанавливаются его начальные атрибуты; многие из них позже можно изменить с помощью функций управления потоками. Атрибуты потока включают размер стека потока, его приоритет и область планирования (локальная или глобальная). Обычно достаточно значений атрибутов, установленных по умолчанию, за исключе-

нием области планирования. Часто программист хочет, чтобы планирование потока происходило глобально, а не локально, т.е. чтобы поток конкурировал за процессор со всеми потоками, а не только с созданными его родительским потоком. В вызове функции pthread_attr_setscope, приведенном выше, это учтено.[10]

Новый поток создается вызовом функции pthread_create:

```
pthread_create(&tid, &tattr, start_func, arg);
```

Первый аргумент — это адрес дескриптора потока, заполняемый при его успешном создании. Второй — адрес дескриптора атрибутов потока, который был инициализирован предыдущим. Новый поток начинает работу вызовом функции start_func с единственным аргументом arg. Если поток создан успешно, функция pthread_create возвращает нуль. Другие значения указывают на ошибку.

Поток завершает свою работу следующим вызовом:

```
pthread_exit(value);
```

Параметр value — это скалярное возвращаемое значение (или NULL). Процедура exit вызывается неявно, если поток возвращается из функции, выполнение которой он начал.

Родительский процесс может ждать завершения работы сыновнего процесса, выполняя функцию

```
pthread_join(tid, value_ptr);
```

Здесь tid является дескриптором сыновнего процесса, а параметр value_ptr — адресом переменной для возвращаемого значения, которая заполняется, когда сыновний процесс вызывает функцию exit.

4.6.2. Семафоры

Потоки взаимодействуют с помощью переменных, объявленных глобальными по отношению к функциям, выполняемым потоками. Потоки могут синхронизироваться с помощью активного ожидания, блокировок, семафоров или условных переменных. Здесь описаны семафоры; блокировки и мониторы представлены в разделе 5.5.

Заголовочный файл semaphore.h содержит определения и прототипы операций для семафоров. Дескрипторы семафоров определены как глобальные относительно потоков, которые будут их использовать, например:

```
sem_t mutex;
```

Дескриптор инициализируется вызовом функции sem_init. Например, следующий вызов присваивает семафору mutex значение 1:

```
sem_init(&mutex, SHARED, 1);
```

Если параметр SHARED не равен нулю, то семафор может быть разделяемым между процессами, иначе семафор могут использовать только потоки одного процесса. Эквивалент операции P в библиотеке Pthreads — функция sem_wait, а операции V — функция sem_post. Итак, один из способов защиты критической секции с помощью семафоров выглядит следующим образом.

```
sem_wait(&mutex);     /* P(mutex) */
критическая секция;
sem_post(&mutex);     /* V(mutex) */
```

Кроме того, в библиотеке Pthreads есть функции для условного ожидания на семафоре, получения текущего значения семафора и его уничтожения.

---

[10] Программы, приведенные в книге и использующие библиотеку Pthreads, тестировались с помощью операционной системы Solaris. На других системах для некоторых атрибутов могут понадобиться другие значения. Например, в системе IRIX видимость для планирования должна быть указана как PTHREAD_SCOPE_PROCESS, и это значение установлено по умолчанию.

4.6.3. Пример: простая программа типа "производитель-потребитель"

В листинге 4.14 приведена простая программа типа производитель-потребитель, аналогичная программе в листинге 4.3. Функции `Producer` и `Consumer` выполняются как независимые потоки. Они разделяют доступ к буферу `data`. Функция `Producer` помещает в буфер последовательность целых чисел от 1 до значения `numIters`. Функция `Consumer` извлекает и складывает эти значения. Для обеспечения попеременного доступа к буферу процесса-производителя и процесса-потребителя использованы два семафора, `empty` и `full`.

Функция `main` инициализирует дескрипторы и семафоры, создает два потока и ожидает завершения их работы. При завершении потоки неявно вызывают функцию `pthread_exit`. В данной программе аргументы потокам не передаются, поэтому в функции `pthread_create` использован адрес `NULL`. Пример, в котором используются аргументы потоков, приведен в разделе 5.5.

Листинг 4.14. Простая программа типа производитель-потребитель, использующая библиотеку Pthreads

```
# include <pthread.h>     /* стандартные строки */
# include <semaphore.h>
# define SHARED 1
# include <stdio.h>
void *Producer(void *);   /* два потока */
void *Consumer(void *);
sem_t empty, full;        /* глобальные семафоры */
int data;                 /* разделяемый буфер */
int numIters;
/* main() - прочитать командную строку и создать потоки */
int main(int argc, char *argv[]) {
  pthread_t pid, cid;      /* поток и атрибуты */
  pthread_attr_t attr;     /* дескрипторы */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  sem_init(&empty, SHARED, 1);    /* sem empty = 1 */
  sem_init(&full, SHARED, 0);    /* sem full = 0 */
  numIters = atoi(argv[1]);
  pthread_create(&pid, &attr, Producer, NULL);
  pthread_create(&cid, &attr, Consumer, NULL);
  pthread_join(pid, NULL);
  pthread_join(cid, NULL);
}
/* поместить в буфер данных числа 1, ..., numIters */
void *Producer(void *arg); {
  int produced;
  for (produced = 1; produced <= numIters; produced++) {
    sem_wait(&empty);
    data = produced;
    sem_post(&full);
  }
}
/* извлечь numIters элементов из буфера и сложить их */
void *Consumer(void *arg) {
  int total = 0, consumed;
  for (consumed = 1; consumed <= numIters; consumed++) {
    sem_wait(&full);
    total = total + data;
    sem_post(&empty);
```

```
  }
  printf("Сумма равна %d\n", total);
}
```

Историческая справка

В середине 1960-х годов Эдсгер Дейкстра (Edsger Dijkstra) и пять его коллег из Технического университета Эйндховена (Нидерланды) разработали одну из первых мультипрограммных операционных систем. (Разработчики назвали ее просто мультипрограммной системой "THE" — по первым буквам названия института на голландском языке.) Эта система имеет элегантную структуру, состоящую из ядра и уровней виртуальных машин, реализованных процессами [Dijkstra, 1968a]. В ней были представлены семафоры, которые Дейкстра изобрел как полезное средство реализации взаимного исключения и выработки сигналов о таких событиях, как прерывания. Дейкстра также ввел термин *частный семафор*.

Поскольку Дейкстра голландец, названия операций P и V происходят от голландских слов. P — это первая буква голландского слова *passeren* (пропустить), а V — первая буква слова *vrijgeven* (освободить). (Отметим аналогию с железнодорожными семафорами.) Дейкстра и его группа позже решили связать букву P со словом *prolagen*, составленного из нидерландских слов *proberen* (попытаться) и *verlagen* (уменьшить), а букву V — со словом *verhogen* (увеличить).

Примерно в это же время Дейкстра написал важную работу [Dijkstra, 1968b] по взаимодействию последовательных процессов. В этой работе было показано, как использовать семафоры для решения различных задач синхронизации, и представлены задачи об обедающих философах и о спящем парикмахере (см. раздел 5.2).

В своей основополагающей работе по мониторам (обсуждаемым в главе 5) Тони Хоар [Hoare, 1974] представил идею разделенного двоичного семафора и показал, как его использовать для реализации мониторов. Однако именно Дейкстра позже дал этому методу название и доказал его практичность. Дейкстра [Dijkstra, 1979] описал использование разделенных двоичных семафоров для решения задачи о читателях и писателях. Он также показал, как реализовать обычные семафоры, используя только разделенные двоичные семафоры [Dijkstra, 1980].

Автор этой книги, вдохновленный работами Дейкстры о разделенных двоичных семафорах, разработал метод передачи эстафеты [Andrews, 1989]. Фактически метод передачи эстафеты является оптимизацией алгоритмов Дейкстры [Dijkstra, 1979, 1980].

Решение задачи об обедающих философах (см. листинг 4.6) является детерминированным, т.е. каждый процесс выполняет предсказуемый набор операций. Леман и Рабин [Lehman and Rabin, 1981] показали, что в любом детерминированном решении, чтобы обеспечить отсутствие взаимоблокировки и зависаний, следует использовать асимметрию или внешний агент. Они также представили интересный абсолютно симметричный вероятностный алгоритм (упражнение 4.19). Основная идея состоит в том, что философы бросают монетку, чтобы определить порядок попыток взять вилки (этим вводится асимметрия), и если два соседа хотят использовать одну и ту же вилку, то уступает тот, который получал вилки последним.

В работе [Courtois, Heymans, and Parnas, 1971] представлена задача о читателях и писателях и два ее решения с помощью семафоров. Первое из них — это решение с приоритетом читателей (раздел 4.4, листинг 4.9). Во втором решении, намного более сложном, предпочтение отдается писателям; в нем используются пять семафоров и два счетчика (см. упражнение 4.20). Кроме того, решение с предпочтением для писателей достаточно сложно понять. В противоположность этому с помощью метода передачи эстафеты, как показано в конце раздела 4.4 и в работе [Andrews, 1989], решение с предпочтением для читателей можно легко изменить так, чтобы преимущество было у писателей, или получить справедливое решение.

Свойства планирования алгоритмов часто зависят от того, являются ли операции с семафорами справедливыми в сильном смысле, т.е. продолжает ли работу процесс, остановленный операцией P, после выполнения достаточного числа операций V. Реализация ядра в главе 6

обеспечивает справедливость в сильном смысле семафоров, поскольку списки блокировок обслуживаются в порядке FIFO (first in — first out, первым пришел — первым ушел). Однако, если заблокированные процессы упорядочены иначе, например, по их приоритету выполнения, то операция P может быть справедливой лишь в слабом смысле. В статье [Morris, 1979] показано, как реализовать решение задачи критической секции без зависаний процессов, используя только справедливые в слабом смысле двоичные семафоры. В работе [Martin and Burch, 1985] представлено более простое решение этой же задачи, а в [Udding, 1986] дано систематическое решение этой задачи, правильность которого легко понять. Во всех трех работах используются разделенные двоичные семафоры и методы, целиком аналогичные передаче эстафеты.

Многие авторы предлагали свои варианты семафоров. Например, в статье [Patil, 1971] предложен примитив PMultiple, который ждет, пока значения всех семафоров некоторого набора не станут неотрицательными, а затем уменьшает их (см. упражнение 4.28). В работе [Reed and Kanodia, 1979] представлены механизмы подсчета событий и сортировки, которые можно использовать для создания семафоров и непосредственно для решения дополнительных задач синхронизации (см. упражнение 4.38). Позже в работе [Faulk and Parnas, 1988] рассмотрены типы синхронизации, возникающие в аппаратных системах реального времени с критическими предельными временами работы. Там доказано, что в системах реального времени операцию P для семафоров необходимо заменить двумя более простыми операциями: pass, ожидающей неотрицательного значения семафора, и down, уменьшающей значение семафора.

Семафоры часто используются для синхронизации в операционных системах. Многие операционные системы предоставляют системные вызовы, доступные для программистов приложений. Как отмечалось в разделе 4.6, библиотека Pthreads была разработана как переносимый стандарт для потоков и низкоуровневых механизмов синхронизации, включая семафоры. (Однако в библиотеке Pthreads нет процедур для барьерной синхронизации.) В нескольких книгах описано программирование потоков в целом и работа с библиотекой Pthreads в частности; один из примеров — [Lewis and Berg, 1998]. Ссылки на информацию о библиотеке Pthreads есть и на Web-странице данной книги (см. предисловие).

Литература

Andrews, G. R. 1989. A method for solving synchronization problems. *Science of Computer Prog.* 13, 4 (December): 1–21.

Courtois, P. J., F. Heymans, and D. L. Parnas. 1971. Concurrent control with "readers" and "writers". *Comm. ACM* 14, 10 (October): 667–668.

Dijkstra, E. W. 1968a. The structure of the "THE" multiprogramming system. *Comm. ACM* 11, 5 (May): 341–346.

Dijkstra, E. W. 1968b. Cooperating sequential processes. In F. Genuys, ed., *Programming Languages.* New York: Academic Press, pp. 43–112. (Э. Дейкстра. Взаимодействующие последовательные процессы. В сб. "Языки программирования" под ред. Ф. Женюи — М.: Мир, 1972.)

Dijkstra. E. W. 1979. A tutorial on the split binary semaphore. EWD 703, Neunen, Netherlands, March.

Dijkstra, E. W. 1980. The superfluity of the general semaphore. EWD 734, Neunen, Netherlands, April.

Faulk, S. R., and D. L. Parnas. 1988. On synchronization in hard-real-time systems. *Comm. ACM* 31, 3 (March): 274–287.

Herman, J. S. 1989. A comparison of synchronization mechanisms for concurrent programming. Master's thesis, CSE-89-26, University of California at Davis.

Hoare, C. A. R. 1974. Monitors: An operating system structuring concept. *Comm. ACM* 17, 10 (October): 549–557.

Lehmann, D., and M. O. Rabin. 1981. A symmetric and fully distributed solution to the dining philosophers problem. *Proc. Eighth ACM Symp. on Principles of Programming Languages*, January, pp. 133–138.

Lewis, B., and D. Berg. 1998. *Multithreaded Programming with Pthreads.* Mountain View. CA: Sun Microsystems Press.

Martin, A. J., and A. J. Burch. 1985. Fair mutual exclusion with unfair P and V operations. *Information Processing Letters* 21, 2 (August): 97–100.

Morris, J. M. 1979. A starvation-free solution to the mutual exclusion problem. *Information Processing Letters* 8, 2 (February): 76–80.

Parnas, D. L. 1975. On a solution to the cigarette smoker's problem (without conditional statements). *Comm. ACM* 18, 3 (March): 181–183.

Patil, S. S. 1971. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes. MIT Project MAC Memo 57, February.

Reed. D. P., and R. K. Kanodia. 1979. Synchronization with eventcounts and sequencers. *Comm. ACM* 22, 2 (February): 115–123.

Udding, J. 1986. Absence of individual starvation using weak semaphores. *Information Processing Letters* 23, 3 (October): 159–162.

Упражнения

4.1. Разработайте модель обычных семафоров, используя только двоичные семафоры. Определите глобальный инвариант, затем разработайте крупномодульное и мелкомодульное решения. (*Указание.* Воспользуйтесь методом передачи эстафеты.)

4.2. В разделе 4.1 операции с семафорами определены с помощью операторов `await`. Это значит, что правила вывода для операций `P` и `V` можно получить, применяя правило оператора `await` (см. раздел 2.6):

а) разработайте правила вывода для операций `P` и `V`;

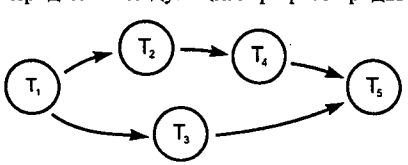
б) используя построенные правила, докажите корректность программы в листинге 4.1. Для этого с помощью метода исключения конфигураций (см. раздел 2.8) докажите невозможность одновременного пребывания двух процессов в критических секциях.

4.3. Напомним, что операция "извлечь и сложить" `FA(var, increment)` является неделимой функцией, которая возвращает старое значение переменной `var` и прибавляет к ней значение `increment`. Используя операцию `FA`, проимитируйте операции `P` и `V` для обычного семафора `s`. Можно предположить, что чтение и запись памяти происходят неделимым образом, но других неделимых операций, кроме `FA`, нет.

4.4. Граф предшествования — это направленный ациклический граф. Его узлы представляют задачи, а дуги показывают порядок их выполнения. Задача может быть выполнена, как только будут завершены все предшествующие ей задачи. Допустим, что задачи являются процессами, и каждый из них имеет такую схему.

```
process T {
    ожидать предшественников, если они есть;
    тело задачи;
    сигнализировать последователям, если они есть;
}
```

а) используя семафоры, покажите, как синхронизировать пять процессов, порядок выполнения которых определен следующим графом предшествования.



Используйте как можно меньше семафоров и не накладывайте ограничений, не указанных графом. Например, после завершения процесса T1 процессы T2 и T3 могут выполняться параллельно;

б) опишите, как синхронизировать процессы, заданные произвольным графом предшествования. Для этого разработайте общий метод распределения семафоров по ребрам графа или процессам и их использования. Не пытайтесь использовать минимально возможное число семафоров, поскольку определение этого числа является NP-трудной задачей для произвольного графа предшествования!

4.5. Предположим, что в машине есть неделимые инструкции инкремента и декремента, `INC(var)` и `DEC(var)`. Инструкция `INC(var)` прибавляет 1 к переменной `var`, `DEC(var)` уменьшает `var` на 1. Считайте, что чтение и запись памяти происходят неделимым образом, но других неделимых операций, кроме `INC` и `DEC`, нет:

а) можно ли промоделировать операции `P` и `V` для обычного семафора `s`? Если можно, то сделайте это и объясните работу модели. Если нельзя, подробно объясните, почему;

б) предположим, что инструкции `INC` и `DEC` также возвращают знаковый бит конечного значения переменной `var`. Если оно отрицательно, инструкция возвращает 1, а иначе — 0. Можно ли теперь промоделировать операции `P` и `V` для обычного семафора `s`? Если можно, то сделайте это и объясните работу модели. Если нельзя, подробно объясните, почему.

4.6. В ядре ОС UNIX есть две операции, аналогичные следующим:

`sleep()`: *заблокировать выполняемый процесс*
`wakeup()`: *запустить все приостановленные процессы*

Эти операции выполняются неделимым образом. Вызов `sleep` всегда блокирует вызвавший ее процесс. Вызов `wakeup` запускает все процессы, заблокированные на момент вызова операции.

4.7. Разработайте реализацию этих операций, используя для синхронизации семафоры. (*Указание.* Примените метод передачи эстафеты.)

Рассмотрим примитивы `sleep` и `wakeup`, определенные в предыдущем упражнении. Процесс `P1` должен выполнить операторы `S1` и `S2`; процесс `P2` — операторы `S3` и `S4`. Оператор `S4` должен быть выполнен после оператора `S1`. Коллега дал вам такую программу:

```
process P1 {                    process P2 {
  S1; wakeup(); S2;               S3; sleep(); S4;
}                               }
```

Корректно ли это решение? Если да, объясните, почему. Если нет, объясните, в чем состоит ошибка, и опишите, как можно изменить операции, чтобы это решение стало правильным.

4.8. Перечислите все возможные заключительные значения переменной `x` в следующей программе. Объясните, как был получен ответ.

```
int x = 0; sem s1 = 1, s2 = 0;
co P(s2); P(s1); x = x*2; V(s1);
// P(s1); x = x*x; V(s1);
// P(s1); x = x+3; V(s2); V(s1);
oc
```

4.9. Рассмотрите барьер с объединяющим деревом в листингах 3.1 и 3.13:

а) используя семафоры для синхронизации, укажите действия в листьях дерева, промежуточных узлах и корне. Обеспечьте возможность повторного использования барьера одной и той же группой процессов;

б) если есть n процессов, то какой функцией от n измеряется общее время работы одного полного цикла барьерной синхронизации? Считайте, что каждая операция с семафором длится одну единицу времени. Проиллюстрируйте свой ответ, показав структуру объединяющего дерева, и укажите время выполнения для нескольких значений параметра n.

4.10. Рассмотрим барьер-бабочку и барьер с распространением в листингах 3.2 и 3.3:

а) используя семафоры для синхронизации, уточните барьер-бабочку для восьми процессов. Приведите код, который должен выполнять каждый из процессов. Барьер должен быть повторно используемым;

б) повторите пункт *а* для барьера с распространением для восьми процессов;

в) сравните ответы к пунктам *а* и *б*. Сколько переменных необходимо для каждого типа барьера? Если каждая операция с семафором длится одну единицу времени, каково общее время, необходимое для барьерной синхронизации по каждому из алгоритмов?

г) повторите пункты *а*, *б* и *в* для барьера на 7 процессов;

д) повторите пункты *а*, *б* и *в* для барьера на 12 процессов.

4.11. Барьер для n процессов можно реализовать с двумя семафорами и одним счетчиком, которые определены следующим образом:

```
int count = 0;
sem arrive = 1, go = 0;
```

Разработайте решение. (*Указание.* Используйте идею передачи эстафеты.)

4.12. Можно построить простой барьер со временем выполнения $O(n)$ для n рабочих процессов, используя массив из n семафоров. Покажите, как это сделать. Рабочие процессы должны синхронизироваться друг с другом. Другие процессы не используйте. Обязательно укажите начальные значения семафоров.

4.13. Рассмотрим следующий вариант реализации оператора await с семафорами.

```
sem e = 1, d = 0; # семафоры входа и задержки
int nd = 0;       # счетчик задержек
# реализация ⟨await(B) S;⟩
P(e);
while (!B)
  { nd = nd+1; V(e); P(d); P(e); }
S;
while (nd > 0)
  { nd = nd-1; V(d); }
V(e);
```

Гарантируется ли этим кодом неделимое выполнение оператора await? Будут ли при этом отсутствовать взаимоблокировки? Обязательно ли будет истинным условие B перед выполнением S? Аргументируйте свои ответы на каждый из этих вопросов.

4.14. Разработайте параллельную реализацию кольцевого буфера для нескольких производителей и нескольких потребителей. Для этого измените решение в листинге 4.4 так, чтобы одновременно могли выполняться несколько операций помещения в буфер и несколько операций извлечения из буфера. Каждая операция помещения данных должна записывать сообщение в отдельную пустую ячейку. Операции извлечения должны считывать данные из разных ячеек.

4.15. Ниже изложен еще один способ решения задачи о кольцевом буфере. Пусть count — целочисленная переменная с возможными значениями от 0 до n. Операции deposit и fetch запрограммированы следующим образом.

```
deposit:
  ⟨ await (count < n)
      buf[rear] = data;
      rear = (rear+1) % n; count = count+1; ⟩
fetch:
  ⟨ await (count > 0)
      result = buf[front];
      front = (front+1) % n; count = count-1; ⟩
```

Реализуйте эти операторы `await` с помощью семафоров. (*Указание.* Используйте вариант метода передачи эстафеты.)

4.16. *Неделимая рассылка.* Предположим, что один процесс-производитель и `n` процессов-потребителей разделяют общий буфер. Производитель помещает в буфер сообщения, потребители извлекают их. Каждое сообщение, помещенное в буфер производителем, должны получить все `n` потребителей, и только после этого производитель сможет поместить в буфер следующее сообщение:

а) разработайте решение этой задачи, используя для синхронизации семафоры;

б) предположим, что буфер состоит из `b` ячеек. Производитель может помещать сообщения только в пустые ячейки, и каждое сообщение должно быть получено всеми `n` потребителями до того, как ячейка будет использована вновь. Кроме того, потребители должны получать сообщения в порядке их помещения в буфер. Однако потребители могут получать сообщения в разное время. Например, разница в количестве полученных сообщений между "быстрым" и "медленным" потребителями может достигать `b`. Дополните ответ к пункту *а*, чтобы получить решение для данной более общей задачи.

4.17. Решите задачу об обедающих философах, уделив основное внимание состояниям философов, а не вилок. Пусть `eating[5]` — массив логических переменных: если философ `Philosopher[i]` ест, то элемент `eating[i]` является истинным, иначе — ложным:

а) определите глобальный инвариант, разработайте крупномодульное решение, а затем мелкомодульное с семафорами для синхронизации. В решении должны отсутствовать взаимоблокировки, но отдельный философ может голодать бесконечно. Используйте метод передачи эстафеты;

б) измените ответ к пункту *а*, чтобы философы не голодали бесконечно, т.е., если философ хочет есть, он в конце концов поест.

4.18. Решите задачу об обедающих философах, используя центральный управляющий процесс. Когда философ хочет есть, он сообщает управляющему и ждет разрешения (т.е. ждет, что ему дадут обе вилки). Для синхронизации используйте семафоры. В решении не должно быть взаимоблокировок и бесконечного голодания философов.

4.19. В задаче об обедающих философах предположим, что для выяснения, какую вилку взять первой, философ бросает жребий:

а) разработайте полностью симметричное решение этой задачи. Все философы должны работать по одному алгоритму. Для синхронизации используйте активное ожидание, семафоры или и то, и другое. Взаимоблокировок быть не должно, но отдельный философ может голодать бесконечно долго, хотя и с малой вероятностью (если вторая вилка недоступна, философ должен положить первую вилку и вновь бросить монетку);

б) расширьте свой ответ к пункту *а*, чтобы исключить возможность бесконечного голодания. Поясните свое решение. (*Указание.* Чтобы один философ не ел второй раз подряд, когда его сосед хочет есть, введите дополнительные переменные.)

4.20. Рассмотрим следующее решение задачи о читателях и писателях с предпочтением для писателей [Courtois et al., 1971].

```
int nr = 0, nw = 0;
sem m1 = 1, m2 = 1, m3 = 1;        #семафоры взаимного исключения
sem read = 1, write = 1;           #семафоры чтения/записи
```

Процессы-читатели:

```
P(m3);
  P(read);
  P(m1);
    nr = nr+1;
    if (nr == 1) P(write);
  V(m1);
  V(read);
V(m3);
читать базу данных;
P(m1);
  nr = nr-1;
  if (nr == 0) V(write);
V(m1);
```

Процессы-писатели:

```
P(m2);
  nw = nw+1;
  if (nw == 1) P(read);
V(m2);
P(write);
записать в базу данных;
V(write);
P(m2);
  nw = nw-1;
  if (nw == 0) V(read);
V(m2);
```

Код, выполняемый процессом-читателем, аналогичен приведенному в листинге 4.9, но код процесса-писателя намного сложнее.

Объясните роль каждого семафора. Постройте утверждения, указывающие, *что именно* истинно в критических точках. Покажите, что решение обеспечивает писателям исключительный доступ к базе данных, и писатель исключает читателей. Представьте убедительные аргументы преимущества писателей перед читателями в этом решении.

4.21. Рассмотрим следующее решение задачи о читателях и писателях. Здесь использованы те же счетчики и семафоры, что и в листинге 4.12, но несколько иначе:

```
int nr = 0, nw = 0;  # число читателей и писателей
sem e = 1;           # семафор взаимного исключения
sem r = 0, w = 0;    # семафоры задержек
int dr = 0, dw = 0;  # счетчики задержек

process Reader[i = 1 to M] {
  while (true) {
    P(e);
      if (nw == 0) { nr = nr+1; V(r); }
      else dr = dr+1;
    V(e);
    P(r);              # ожидать разрешения на чтение
    читать базу данных;
    P(e);
      nr = nr-1;
      if (nr == 0 and dw > 0)
        { dw = dw-1; nw = nw+1; V(w); }
    V(e);
  }
}

process Writer[j = 1 to N] {
while (true) {
    P(e);
      if (nr == 0 and nw == 0) { nw = nw+1; V(w); }
      else dw = dw+1;
    V(e);
    P(w);
    записать в базу данных;
    P(e);
```

```
            nw = nw-1;
            if (dw > 0) { dw = dw-1; nw = nw+1; V(w); }
            else
              while (dr > 0) { dr = dr-1; nr = nr+1; V(r); }
          V(e);
        }
      }
```

а) подробно объясните работу этого решения. Какова роль каждого семафора? Покажите, что это решение обеспечивает писателям исключительный доступ к базе данных и исключение читателей писателем;

б) каким процессам в этом решении отдается предпочтение? Читателям? Писателям? Получают ли читатели и писатели предпочтение по очереди?

в) сравните это решение с программой в листинге 4.12. Сколько операций `P` и `V` процессы выполняют в лучшем случае? В худшем случае? Какую из программ вам легче понять и почему?

4.22. Рассмотрим программу для читателей и писателей в листинге 4.12:

а) измените программу, чтобы у писателей было преимущество перед читателями. Основная идея описана в тексте главы, подробности продумайте сами;

б) сравните свой ответ к пункту *а* с программой из предыдущего упражнения. Сколько операций над семафорами выполняют читатели и писатели в каждом из решений в лучшем случае? В худшем случае?

4.23. Измените решение для задачи о читателях и писателях в листинге 4.12 так, чтобы протокол выхода в процессах `Writer` запускал все приостановленные процессы-читатели, если такие есть. (*Указание.* Вам придется изменить и протокол входа процессов `Reader`.)

4.24. Предположим, что в задаче о читателях и писателях есть `n` процессов. Пусть `rw` — семафор с начальным значением `n`, а протокол читателя имеет вид:

```
P(rw);
читать базу данных;
V(rw);
```

Разработайте протокол для процессов-писателей, который обеспечивает необходимое исключение, не вызывает взаимоблокировок и зависаний (при условии, что операция `P` справедлива в сильном смысле).

4.25. *Задача о молекуле воды.* Допустим, что атомы водорода и кислорода колеблются в пространстве, пытаясь собраться в молекулы воды. Для этого нужна взаимная синхронизация двух атомов водорода и одного атома кислорода. Пусть атомы кислорода (O) и водорода (H) моделируются процессами (потоками). Каждый атом H вызывает процедуру `Hready`, когда он готов объединиться в молекулу. Аналогично готовый к соединению атом кислорода O вызывает процедуру `Oready`.

Напишите эти две процедуры, пользуясь для синхронизации семафорами. Атом H в процедуре `Hready` должен приостановиться, пока другой атом H и один атом O не вызовут соответственно процедуры `Hready` и `Oready`. Тогда один из процессов (скажем, атом O) должен вызвать процедуру `makeWater`. После выхода из процедуры `makeWater` *все три* процесса должны вернуться из своих вызовов `Hready` и `Oready`. Не пользуйтесь в решении активным ожиданием, обеспечьте отсутствие взаимной блокировки и зависаний. Это непростая задача, так что будьте внимательны. (*Указание.* Попробуйте начать с определения глобального инварианта и использовать метод передачи эстафеты.)

4.26. Пусть операции `P` и `V` для семафора заменили следующими:

```
PChunk(s, amt): ⟨await (s >= amt) s = s - amt;⟩
VChunk(s, amt): ⟨s = s + amt;⟩
```

Значением переменной amt является положительное целое число. Эти операции обобщают операции P и V на случаи, когда значение amt не равно 1:

а) с помощью операций PChunk и VChunk постройте простое решение задачи о читателях и писателях с преимуществом читателей. Сначала определите глобальный инвариант, затем разработайте крупномодульное решение, после чего перейдите к мелкомодульному решению задачи;

б) какие еще задачи могут выиграть от использования указанных операций и почему? (*Указание.* Рассмотрите задачи из текста главы и данных упражнений.)

4.27. *Задача о курильщиках* [Patil, 1971], [Parnas, 1975]. Предположим, что есть три процесса-курильщика и один процесс-посредник. Курильщик непрерывно скручивает сигареты и курит их. Чтобы скрутить сигарету, нужны три компонента: табак, бумага и спичка. У одного процесса-курильщика есть табак, у другого — бумага, у третьего — спички, и запасы этих компонентов бесконечны. Посредник кладет на стол по два случайных компонента. Тот процесс-курильщик, у которого есть третий компонент, забирает два компонента со стола, скручивает сигарету и курит ее. Посредник дожидается, пока курильщик закончит. После этого цикл повторяется.

Разработайте решение этой задачи с семафорами для синхронизации. Для этого могут потребоваться и другие переменные.

4.28. Предположим, что операции P и V для семафора заменены следующими.

```
PMultiple(s1, ..., sN):
  ⟨ await (s1 > 0 and ... and sN > 0)
          s1 = s1-1; ...; sN = sN-1; ⟩
VMultiple(s1, ..., sN):
  ⟨ s1 = s1+1; ...; sN = sN+1; ⟩
```

Аргументами могут быть один или несколько семафоров. Эти операции обобщают P и V, позволяя процессу выполнить их для нескольких семафоров одновременно:

а) пользуясь операциями PMultiple и VMultiple, постройте простое решение задачи о курильщиках, поставленной в предыдущем упражнении;

б) рассмотрите другие задачи в главе 4 и в данных упражнениях. Какие из них могут выиграть от использования таких операций и почему?

4.29. Рассмотрим функцию exchange(value), которую процессы вызывают, чтобы обменяться своими аргументами. Первый процесс для вызова функции exchange должен приостановиться. Когда функцию exchange вызывает второй процесс, два значения меняются местами и возвращаются процессам. Тот же сценарий повторяется при третьем и четвертом вызовах процесса, пятом и шестом и так далее.

Разработайте код для реализации тела функции exchange. Для синхронизации используйте семафоры. Обратите внимание на декларации и инициализацию всех необходимых переменных, а также на глобальное по отношению к процедуре положение разделяемых переменных.

4.30. *Общая душевая.* Предположим, что в общежитии есть душевая, которой могут пользоваться и мужчины, и женщины, но не одновременно:

а) определите глобальный инвариант, затем разработайте решение, пользуясь для синхронизации семафорами. Одновременно в душевой может находиться сколько угодно мужчин или женщин. Решение должно обеспечивать необходимое исключение и отсутствие взаимоблокировок, но оно может не быть справедливым;

б) измените свой ответ к пункту *а* так, чтобы в душевой одновременно находилось не больше четырех человек;

в) измените ответ, чтобы обеспечить справедливость планирования. Возможно, понадобится другой подход к решению. (*Указание.* Используйте метод передачи эстафеты.)

4.31. *Узкий мост.* К узкому мосту приезжают машины с севера и с юга. Машины, движущиеся в одном направлении, могут переезжать мост одновременно, а в противоположных — нет:

а) определите глобальный инвариант, затем разработайте решение, пользуясь для синхронизации семафорами. Решение может не быть справедливым;

б) измените ответ к пункту *а* так, чтобы любая машина, подъехавшая к мосту, в конце концов через него переехала. Возможно, для этого понадобится решить задачу иначе. (*Указание.* Используйте метод передачи эстафеты.)

4.32. *Поиск, вставка и удаление.* Процессы трех типов разделяют доступ к однонаправленному списку: процессы поиска, вставки и удаления. Процесс поиска просматривает список, поэтому несколько таких процессов могут работать одновременно. Процесс вставки добавляет новый элемент в конец списка. Операции вставки должны исключать друг друга, предотвращая несколько вставок примерно в одно и то же время, но один процесс вставки может работать одновременно с любым количеством процессов поиска. Процесс удаления исключает элемент на любой позиции в списке. В каждый момент времени только один процесс удаления может иметь доступ к списку. Удаления должны взаимно исключать поиски и вставки:

а) эта задача является примером избирательного взаимного исключения. Разработайте решение с семафорами, по стилю аналогичное решению задачи о читателях и писателях (листинг 4.9);

б) это также пример задачи условной синхронизации. Постройте решение, по стилю аналогичное решению задачи о читателях и писателях (листинг 4.12). Сначала определите свойство синхронизации в виде глобального инварианта. Используйте три счетчика активных процессов: поиска `ns`, вставки `ni`, удаления `nd`. Постройте крупномодульное решение. Разработайте мелкомодульное решение, используя метод передачи эстафеты.

4.33. Рассмотрим следующую задачу выделения памяти. Пусть есть две операции: `request(amount)` и `release(amount)`, где `amount` — положительное целое число. Когда процесс вызывает операцию `request`, его выполнение приостанавливается, пока не станут доступными как минимум `amount` свободных страниц. После этого процесс получает `amount` страниц памяти. Вызывая операцию `release`, процесс освобождает `amount` страниц (страницы могут выделяться и освобождаться в разных количествах):

а) разработайте реализацию операций `request` и `release`, в которой используется стратегия распределения ресурсов "кратчайшая задача" (меньшим запросам отдается предпочтение перед большими). Ваше решение должно быть в стиле программы в листинге 4.13;

б) разработайте реализацию операций `request` и `release`, в которой используется стратегия "первым пришел — первым обслужен". Если объема памяти недостаточно для удовлетворения пришедшего запроса, он должен быть приостановлен.

4.34. Пусть `n` процессов `U[1:n]` разделяют два принтера. Перед использованием принтера процесс `U[i]` вызывает функцию `request(printer)`. Эта операция сначала ждет, пока один из двух принтеров не станет доступным, и возвращает идентификатор свободного принтера. После использования этого принтера процесс `U[i]` освобождает его, вызывая функцию `release(printer)`. Обе функции выполняются неделимым образом:

а) разработайте реализацию операций `request` и `release` с семафорами для синхронизации;

б) предположим, что каждый процесс имеет приоритет, записанный в глобальном массиве `priority[1:n]`. Измените функции `request` и `release` так, чтобы принтер выделялся ожидающему процессу с максимальным приоритетом. Можно считать, что приоритеты процессов уникальны.

4.35. *Голодные птицы.* Есть `n` птенцов и их мать. Птенцы едят из общей миски, в которой сначала находится `F` порций пищи. Каждый птенец съедает порцию еды, спит некоторое время, затем снова ест. Когда кончается еда, птенец, евший последним, зовет мать. Птица наполняет миску `F` порциями еды и снова ждет, пока миска опустеет. Эти действия повторяются без конца.

Представьте птиц процессами и разработайте код, моделирующий их действия. Для синхронизации используйте семафоры.

4.36. *Медведь и пчелы.* Есть `n` пчел и медведь. Они пользуются одним горшком меда, вмещающим `H` порций меда. Сначала горшок пустой. Пока горшок не наполнится, медведь спит, потом съедает весь мед и засыпает. Каждая пчела многократно собирает одну порцию меда и кладет ее в горшок. Пчела, которая приносит последнюю порцию меда и заполняет горшок, будит медведя.

Представьте медведя и пчел процессами, разработайте код, моделирующий их действия. Для синхронизации используйте семафоры.

4.37. *Американские горки* (roller coaster) [Herman, 1989]. Есть `n` процессов-пассажиров и один процесс-вагончик. Пассажиры ждут очереди проехать в вагончике, вмещающем `C` человек, `C < n`. Вагончик может ехать только заполненным:

а) разработайте коды процесса-пассажира и процесса-вагончика. Для синхронизации используйте семафоры;

б) обобщите ответ к пункту *а*, чтобы использовались `m` процессов-вагончиков, `m > 1`. Дорога одна, так что обгон вагончиков невозможен, и заканчивать движение по дороге они должны в том же порядке, в котором начали. Как и ранее, вагончик может ехать только заполненным.

4.38. Для подсчета числа событий используется счетчик событий. Он принимает целые значения, инициализируется нулем и управляется следующими тремя операциями.

```
advance(ec): ⟨ec = ec+1;⟩
read(ec): ⟨return(ec);⟩
wait(ec, value): ⟨await (ec >= value);⟩
```

Генератор выдает уникальные числа. Он также представлен целым числом, инициализирован `0` и управляется следующей неделимой операцией:

```
ticket(seq): ⟨temp = seq; seq = seq+1; return(temp);⟩
```

Генератор часто используется в качестве второго аргумента операции `wait`:

а) используя эти операции, разработайте решение задачи о кольцевом буфере. Считайте, что есть один производитель и один потребитель, а буфер состоит из `n` ячеек (см. листинг 4.4);

б) обобщите ответ к пункту *а* для нескольких потребителей и производителей;

в) используя эти операции, разработайте реализацию операций `P` и `V` для обычного семафора.

Глава 5

Мониторы

Семафоры являются фундаментальным механизмом синхронизации. Как показано в главе 4, их использование облегчает программирование взаимного исключения и сигнализации, причем их можно применять систематически при решении любых задач синхронизации. Однако семафоры — низкоуровневый механизм; пользуясь ими, легко наделать ошибок. Например, программист должен следить за тем, чтобы случайно не пропустить вызовы операций P и V или задать их больше, чем нужно. Можно неправильно выбрать тип семафора или защитить не все критические секции. Семафоры глобальны по отношению ко всем процессам, поэтому, чтобы разобраться, как используется семафор или другая разделяемая переменная, необходимо просмотреть всю программу. Наконец, при использовании семафоров взаимное исключение и условная синхронизация программируются одной и той же парой примитивов. Из-за этого трудно понять, для чего предназначены конкретные P и V, не посмотрев на другие операции с данным семафором. Взаимное исключение и условная синхронизация — это разные понятия, потому и программировать их лучше разными способами.

Мониторы — это программные модули, которые обеспечивают большую структурированность, чем семафоры, хотя реализуются так же эффективно. В первую очередь, мониторы являются механизмом абстракции данных. Монитор инкапсулирует представление абстрактного объекта и обеспечивает набор операций, *только* с помощью которых оно обрабатывается. Монитор содержит переменные, хранящие состояние объекта, и процедуры, реализующие операции над ним. Процесс получает доступ к переменным в мониторе только путем вызова процедур этого монитора. Взаимное исключение обеспечивается неявно тем, что процедуры в одном мониторе не могут выполняться параллельно. Это похоже на неявное взаимное исключение, гарантируемое операторами `await`. Условная синхронизация в мониторах обеспечивается явно с помощью *условных переменных* (condition variable). Они аналогичны семафорам, но имеют существенные отличия в определении и, следовательно, в использовании для сигнализации.

Параллельная программа, использующая мониторы для взаимодействия и синхронизации, содержит два типа модулей: активные процессы и пассивные мониторы. При условии, что все разделяемые переменные находятся внутри мониторов, два процесса взаимодействуют, вызывая процедуры одного и того же монитора. Получаемая модульность имеет два важных преимущества. Первое — процесс, вызывающий процедуру монитора, может не знать о конкретной реализации процедуры; роль играют лишь видимые результаты вызова процедуры. Второе — программист монитора может не заботиться о том, где и как используются процедуры монитора, и свободно изменять его реализацию, не меняя при этом видимых процедур и результатов их работы. Эти преимущества позволяют разрабатывать процессы и мониторы относительно независимо, что облегчает создание и понимание параллельной программы.

В данной главе подробно описаны мониторы и на примерах показано их использование. Часть примеров уже встречалась, но есть и новые. В разделе 5.1 определены синтаксис и семантика мониторов. В разделе 5.2 представлен ряд полезных методов программирования и примеров их применения: кольцевые буферы, читатели и писатели, планирование типа "кратчайшая задача", интервальный таймер и классическая задача о спящем парикмахере. В разделе 5.3 взято несколько другое направление — в нем рассматривается *структура* решений задач параллельного программирования. Для демонстрации различных методов решения используется еще одна интересная задача — планирование доступа к диску с перемещаемыми головками.

Благодаря своей полезности и эффективности мониторы применяются в нескольких языках программирования. Примечательно их использование в языке Java, описанное в разделе 5.4. Лежащие в основе мониторов механизмы синхронизации (неявное исключение и условные переменные для сигнализации) реализованы также в операционной системе Unix. Наконец, условные переменные поддерживаются несколькими библиотеками программирования. В разделе 5.5 описаны соответствующие процедуры библиотеки потоков POSIX (Pthreads).

5.1. Синтаксис и семантика

Монитор используется, чтобы сгруппировать представление и реализацию разделяемого ресурса (класса). Он состоит из интерфейса и тела. Интерфейс определяет предоставляемые ресурсом операции (методы). Тело содержит переменные, представляющие состояние ресурса, и процедуры, реализующие операции интерфейса.

В разных языках программирования мониторы объявляются и создаются по-разному. Для простоты будем считать, что монитор является статичным объектом, а его тело и интерфейс описаны таким образом.

```
monitor mname {
   объявления постоянных переменных
   операторы инициализации
   процедуры
}
```

Процедуры реализуют видимые операции. Постоянные переменные разделяются всеми процедурами тела монитора. Они называются *постоянными*, поскольку существуют и сохраняют свое значение, пока существует монитор. В процедурах, как обычно, можно использовать локальные переменные, копии которых создаются для каждого вызова функции.

Монитор как представитель абстрактных типов данных обладает тремя свойствами. Во-первых, вне монитора видны только имена процедур — они представляют собой одно-единственное "окно в стене" объявления монитора. Таким образом, чтобы изменить состояние ресурса, представленное постоянными переменными, процесс должен вызвать одну из процедур монитора. Вызов процедуры монитора имеет следующий вид.

```
call mname.opname(arguments)
```

Здесь `mname` — имя монитора, `opname` — имя одной из его операций (процедур), вызываемой с аргументами `arguments`. Если имя `opname` уникально в области видимости вызывающего процедуру процесса, то часть "`mname.`" в вызове процедуры не обязательна.

Во-вторых, операторы внутри монитора (в объявлениях и процедурах) не могут обращаться к переменным, объявленным вне монитора.

В-третьих, постоянные переменные инициализируются до вызова его процедур. Это реализовано путем выполнения инициализирующих операторов при создании монитора и, следовательно, до вызова его процедур.

Одно из привлекательных свойств монитора, как и любого абстрактного типа данных, — возможность его относительно независимой разработки. Это означает, однако, что программист монитора может не знать заранее порядка вызова процедур. Поэтому полезно определить предикат, истинный независимо от порядка выполнения вызовов. *Инвариант монитора* — это предикат, определяющий "разумные" состояния постоянных переменных, когда процессы не обращаются к ним. Код инициализации монитора должен создать состояние, соответствующее инварианту, а каждая процедура должна его поддерживать. (Инвариант монитора аналогичен глобальному инварианту, но для переменных в пределах одного монитора.) Инварианты мониторов включены во все примеры главы. Строка инварианта начинается символами `##`.

Монитор отличается от механизма абстракции данных в языках последовательного программирования тем, что совместно используется параллельно выполняемыми процессами. По-

этому, чтобы избежать взаимного влияния в процессах, выполняемых в мониторах, может потребоваться взаимное исключение, а для приостановки работы до выполнения определенного условия — условная синхронизация. Рассмотрим, как процессы синхронизируются в мониторах.

5.1.1. Взаимное исключение

Синхронизацию проще всего понять и запрограммировать, если взаимное исключение и условная синхронизация выполняются разными способами. Лучше всего, если взаимное исключение происходит неявно, чем автоматически устраняется взаимное влияние. Кроме того, программы легче читать, поскольку в них нет явных протоколов входа в критические секции и выхода из них.

В отличие от взаимного исключения, условную синхронизацию нужно программировать явно, поскольку разные программы требуют различных условий синхронизации. Хотя зачастую проще синхронизировать с помощью логических условий, как в операторах `await`, низкоуровневые механизмы можно реализовать намного эффективнее. Они позволяют программисту более точно управлять порядком выполнения программы, что помогает в решении проблем распределения ресурсов и планирования.

В соответствии с этими замечаниями взаимное исключение в мониторах обеспечивается неявно, а условная синхронизация программируется с помощью так называемых *условных переменных*.

Внешний процесс вызывает процедуру монитора. Пока некоторый процесс выполняет операторы процедуры, она *активна*. В любой момент времени может быть активным только один экземпляр только одной процедуры монитора, т.е. одновременно не могут быть активными ни два вызова разных процедур, ни два вызова одной и той же процедуры.

Процедуры мониторов *по определению* выполняются со взаимным исключением. Оно обеспечивается реализацией языка, библиотекой или операционной системой, но *не* программистом, использующим мониторы. На практике взаимное исключение в языках и библиотеках реализуется с помощью блокировок и семафоров, в однопроцессорных операционных системах — запрета внешних прерываний, а в многопроцессорных операционных системах — межпроцессорных блокировок и запрета прерываний на уровне процессора. В главе 6 подробно описаны вопросы и методы реализации.

5.1.2. Условные переменные

Условная переменная используется для приостановки работы процесса, безопасное выполнение которого невозможно до перехода монитора в состояние, удовлетворяющее некоторому логическому условию. Условные переменные также применяются для запуска приостановленных процессов, когда условие становится истинным. Условная переменная объявляется следующим образом.

```
cond cv;
```

Таким образом, `cond` — это новый тип данных. Массив условных переменных объявляется, как обычно, указанием интервала индексов после имени переменной. Условные переменные можно объявлять и использовать только в пределах мониторов.

Значением условной переменной `cv` является очередь приостановленных процессов (очередь задержки). Вначале она пуста. Программист не может напрямую обращаться к значению переменной `cv`. Вместо этого он получает косвенный доступ к очереди с помощью нескольких специальных операций, описанных ниже.

Процесс может запросить состояние условной переменной с помощью вызова

```
empty(cv);
```

Если очередь переменной `cv` пуста, эта функция возвращает значение "истина", иначе — "ложь".

Процесс блокируется на условной переменной cv с помощью вызова

```
wait(cv);
```

Выполнение операции wait заставляет работающий процесс задержаться в конце очереди переменной cv. Чтобы другой процесс мог в конце концов войти в монитор для запуска приостановленного процесса, выполнение операции wait отбирает у процесса, вызвавшего ее, исключительный доступ к монитору.

Процессы, заблокированные на условных переменных, запускаются операторами signal. При выполнении вызова

```
signal(cv);
```

проверяется очередь задержки переменной cv. Если она пуста, никакие действия не производятся. Однако, если приостановленные процессы есть, оператор signal запускает процесс *в начале очереди*. Таким образом, операции wait и signal обеспечивают порядок сигнализации FIFO: процессы приостанавливаются в порядке вызовов операции wait, а запускаются в порядке вызовов операции signal. Позже будет показано, как добавить к очереди задержки приоритеты планирования, но по умолчанию принимается порядок FIFO.

5.1.3. Дисциплины сигнализации

Выполняя операцию signal, процесс работает в мониторе и, следовательно, может управлять блокировкой, неявно связанной с монитором. В результате возникает дилемма. Если операция signal запускает другой процесс, то получается, что могли бы выполняться два процесса: вызвавший операцию signal и запущенный ею. Но следующим может выполняться только один из них (даже на мультипроцессорах), поскольку лишь один процесс может иметь исключительный доступ к монитору. Таким образом, возможны два варианта:

- *сигнализировать и продолжить*: сигнализатор продолжает работу, а процесс, получивший сигнал, выполняется позже;
- *сигнализировать и ожидать*: сигнализатор ждет некоторое время, а процесс, получивший сигнал, выполняется сразу.

Дисциплина (порядок) "сигнализировать и продолжить" не *прерывает обслуживания*. Процесс, выполняющий операцию signal, сохраняет исключительный доступ к монитору, а запускаемый процесс начнет работу несколько позже, когда получит исключительный доступ к монитору. По существу, операция signal просто указывает запускаемому процессу на возможность выполнения, после чего он возвращается в очередь процессов, ожидающих на блокировке монитора.

Порядок "сигнализировать и ожидать" имеет свойство *прерывания обслуживания*. Процесс, выполняющий операцию signal, передает управление блокировкой монитора запускаемому процессу, т.е. запускаемый процесс прерывает работу процесса-сигнализатора. В этом случае сигнализатор переходит в очередь процессов, ожидающих на блокировке монитора. (Возможен вариант, когда сигнализатор помещается в начало очереди ожидающих процессов; это называется "сигнализировать и срочно (urgent) ожидать".)

Диаграмма состояний на рис. 5.1 иллюстрирует работу синхронизации в мониторах. Вызывая процедуру монитора, процесс помещается во входную очередь, если в мониторе выполняется еще один процесс; в противном случае вызвавший операцию процесс немедленно начинает выполнение в мониторе. Когда монитор освобождается (после возврата из процедуры или выполнения операции wait), один процесс из входной очереди может перейти к работе в мониторе. Выполняя операцию wait(cv), процесс переходит от работы в мониторе в очередь, связанную с условной переменной. Если процесс выполняет операцию signal(cv), то при порядке "сигнализировать и продолжить" (Signal and Continue — SC) процесс из начала очереди услов-

ной переменной переходит во входную. При порядке "сигнализировать и ожидать" (Signal and Wait — SW) процесс, выполняемый в мониторе, переходит во входную очередь, а процесс из начала очереди условной переменной переходит к выполнению в мониторе.

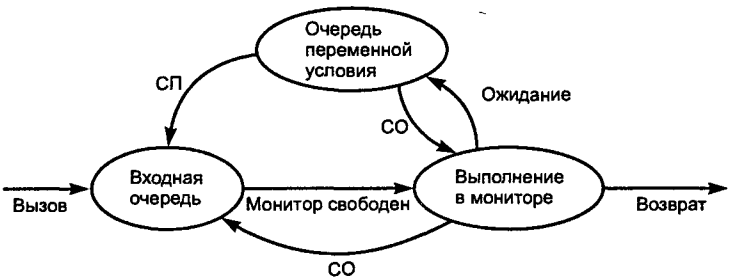


Рис. 5.1. Диаграмма состояний для синхронизации в мониторах

В листинге 5.1 показан монитор, реализующий семафор. Здесь представлены все компоненты монитора, поясняющие различия между порядком выработки сигналов SC и SW. Хотя вряд ли кому-нибудь потребуются мониторы для реализации семафоров, этот пример демонстрирует такую возможность. В главе 6 будет показано, как реализовать мониторы с помощью семафоров. Монитор и семафор дуальны в том смысле, что с помощью одного из них можно реализовать другой, и, следовательно, их можно использовать для решения одних и тех же задач синхронизации. Однако мониторы являются механизмом более высокого уровня, чем семафоры, по причинам, описанным в начале главы.

Листинг 5.1. Реализация семафора с помощью монитора

```
monitor Semaphore {
  int s = 0; ## s >= 0
  cond pos;  # получает сигнал, когда s > 0

  procedure Psem(); {
    while (s == 0) wait(pos);
    s = s-1;
  }

  procedure Vsem() {
    s = s+1;
    signal(pos);
  }
}
```

В листинге 5.1 целочисленная переменная `s` представляет значение семафора. Вызывая операцию `Psem`, процесс приостанавливается, пока значение переменной `s` не станет положительным, затем уменьшает его на 1. Задержка программируется с помощью цикла `while`, который приводит процесс к ожиданию на условной переменной `pos`, если `s` равна 0. Операция `Vsem` увеличивает `s` на 1 и вырабатывает сигнал для переменной `pos`. Если есть приостановленные процессы, запускается "самый старый" из них.

Программа 5.1 корректно работает как при порядке "сигнализировать и ожидать" (SW), так и при "сигнализировать и продолжить" (SC). Под корректностью работы понимается сохранение истинности инварианта семафора `s >= 0`. Порядки работы отличаются только последовательностью выполнения процессов. Вызывая `Psem`, процесс ждет, если `s` равна 0, а после запуска уменьшает значение `s`. Вызывая `Vsem`, процесс сначала увеличивает `s`, после чего запускает приостановленный процесс, если такой есть. При порядке SW запускаемый

процесс выполняется *сразу* и уменьшает значение семафора s. При порядке SC запускаемый процесс выполняется через некоторое время *после* процесса, выработавшего сигнал. Запускаемый процесс должен перепроверить значение семафора s и убедиться, что оно все еще положительно. Это необходимо сделать, поскольку возможно, что другой процесс из очереди входа до этого вызвал действие Psem и уменьшил s. Таким образом, код в листинге 5.1 обеспечивает последовательность обслуживания FIFO для порядка SW, но *не* для порядка SC.

Листинг 5.1 демонстрирует еще одно различие между порядками выработки сигналов SW и SC. При порядке SW цикл while в действии Psem можно заменить простым оператором if:

```
if (s == 0) wait(pos);
```

При этом процесс, получивший сигнал, сразу начинает работу. Это гарантирует, что значение s положительно, когда данный процесс его уменьшает.

Монитор, показанный в листинге 5.1, можно изменить так, чтобы он корректно работал при обоих порядках запуска процессов (SC и SW), не использовал цикл while и реализовывал семафор с порядком обслуживания FIFO. Чтобы понять, как это сделать, вернемся к программе в листинге 5.1. Когда процесс впервые вызывает операцию Psem, он должен приостановиться, если значение s равно нулю. Вызывая операцию Vsem, процесс собирается запустить приостановленный процесс, если такой есть. Различие между выработкой сигналов в порядке SC и SW состоит в том, что если процесс-сигнализатор продолжает выполняться, то уже увеличенное на единицу значение семафора s может прочитать не тот процесс, который только что был запущен. Избежать этой проблемы можно, если вызывающий операцию Vsem процесс примет следующее решение: если есть приостановленный процесс, то нужно сигнализировать для переменной pos, *не* увеличивая значение семафора s, иначе увеличить s. Соответственно, если процесс, вызывающий операцию Psem, должен ожидать, то в дальнейшем он не уменьшит s, поскольку к тому времени оно не будет увеличено процессом, выработавшим сигнал.

Монитор, использующий описанный способ, представлен в листинге 5.2. Этот метод называется *передачей условия*, поскольку, по существу, сигнализатор неявно передает значение условия (переменная s положительна) процессу, который он запускает. Условие не делается видимым, поэтому никакой другой процесс, кроме запускаемого операцией signal, не увидит, что условие стало истинным и не прекратит ожидания.

Листинг 5.2. Семафор FIFO с передачей условия

```
monitor FIFOsemaphore {
  int s = 0; ## s >= 0
  cond pos;  # получает сигнал, когда s > 0

  procedure Psem(); {
    if (s == 0)
      wait(pos);
    else
      s = s-1;
  }

  procedure Vsem() {
    if (empty(pos))
      s = s+1;
    else
      signal(pos);
  }
}
```

Метод передачи условия можно применять везде, где в процедурах с вызовами wait и signal есть действия, дополняющие друг друга. В листинге 5.2 таковыми являются увеличение перемен-

ной s в процедуре Psem и ее уменьшение в процедуре Vsem. В разделах 5.2 и 5.3 будут приведены дополнительные примеры с использованием этого метода в решении задач планирования.

Из листингов 5.1 и 5.2 видно, что условные переменные аналогичны операциям P и V с семафорами. Операция wait, подобно P, приостанавливает процесс, а операция signal, как и V, запускает его. Однако есть два существенных отличия. Первое — операция wait *всегда* приостанавливает процесс до последующего выполнения операции signal, тогда как операция P вызывает остановку процесса, только если текущее значение семафора равно нулю. Второе — операция signal не производит никаких действий, если нет процессов, приостановленных на условной переменной, тогда как V либо запускает приостановленный процесс, либо увеличивает значение семафора, т.е. факт выполнения операции signal не запоминается. Из-за этих отличий условная синхронизация с мониторами программируется не так, как с семафорами.

В оставшейся части данной главы будем предполагать, что мониторы используют порядок "сигнализировать и продолжить". Первым для мониторов был предложен порядок "сигнализировать и ожидать", однако SC был принят в операционной системе Unix, языке программирования Java и библиотеке Pthreads. Порядку SC было отдано предпочтение, поскольку он совместим с планированием процессов на основе приоритетов и имеет более простую формальную семантику. (Эти вопросы обсуждаются в исторической справке.)

5.1.4. Дополнительные операции с условными переменными

До сих пор с условными переменными использовались три операции: empty, wait и signal. Полезны еще три: приоритетная wait, minrank и signal_all. Все они имеют простую семантику и могут быть эффективно реализованы, поскольку обеспечивают лишь дополнительные действия над очередью, связанной с условной переменной. Все шесть операций представлены в табл. 5.1.

Таблица 5.1. Операции над условными переменными

wait(cv)	Ждать в конце очереди
wait(cv, rank)	Ждать в порядке возрастания значения ранга (rank)
signal(cv)	Запустить процесс из начала очереди и продолжить
signal_all(cv)	Запустить все процессы очереди и продолжить
empty(cv)	Истина, если очередь ожидания пуста, иначе — ложь
minrank(cv)	Значение ранга процесса в начале очереди ожидания

С помощью операций wait и signal приостановленные процессы запускаются в том же порядке, в котором они были задержаны, т.е. очередь задержки является FIFO-очередью. *Приоритетный* оператор wait позволяет программисту влиять на порядок постановки процессов в очередь и их запуска. Оператор имеет вид:

```
wait(cv, rank)
```

Параметр cv — это условная переменная, а rank — целочисленное выражение. Процессы приостанавливаются на переменной cv в порядке возрастания значения rank и, следовательно, в этом же порядке запускаются. При равенстве рангов запускается процесс, ожидавший дольше всех. Во избежание потенциальных проблем, связанных с совместным применением обычного и приоритетного операторов wait для одной переменной, программист должен всегда использовать только один тип оператора wait.

Для задач планирования, в которых используется приоритетный оператор wait, часто полезно иметь возможность определить ранг процесса в начале очереди задержки. Из вызова

```
minrank(cv)
```

возвращается ранг приостановки процесса в начале очереди задержки условной переменной cv при условии, что очередь не пуста и для процесса в начале очереди был использован приоритетный оператор wait. В противном случае возвращается некоторое произвольное целое число.

Оповещающая операция signal — последняя с условными переменными. Она используется, если можно возобновить более одного приостановленного процесса или если процесс-сигнализатор не знает, какие из приостановленных процессов могли бы продолжать работу (поскольку им самим нужно перепроверить условия приостановки). Эта операция имеет вид:

```
signal_all(cv)
```

Выполнение оператора signal_all запускает *все* процессы, приостановленные на условной переменной cv. При порядке "сигнализировать и продолжить" он аналогичен коду:

```
while (!empty(cv)) signal(cv);
```

Запускаемые процессы возобновляют выполнение в мониторе через некоторое время в соответствии с ограничениями взаимного исключения. Как и оператор signal, оператор signal_all не дает результата, если нет процессов, приостановленных на условной переменной cv. Процесс, выработавший сигнал, также продолжает выполняться в мониторе.

Операция signal_all вполне определена, когда мониторы используют порядок "сигнализировать и продолжить", поскольку процесс, выработавший сигнал, всегда продолжает работать в мониторе. Но при использовании порядка "сигнализировать и ожидать" эта операция определена не точно, поскольку становится возможным передать управление более, чем одному процессу, *и* дать каждому процессу исключительный доступ к монитору. Это еще одна причина, по которой в операционной системе Unix, языке Java, библиотеке Pthreads и данной книге используется порядок запуска "сигнализировать и продолжить".

5.2. Методы синхронизации

В этом разделе разработаны решения пяти задач: о кольцевых буферах, читателях и писателях, планировании типа "кратчайшее задание", интервальных таймерах и спящем парикмахере. Каждая из задач по-своему интересна и иллюстрирует технику программирования с мониторами.

5.2.1. Кольцевые буферы: базовая условная синхронизация

Вернемся к задаче о кольцевом буфере (см. раздел 4.2). Процесс-производитель и процесс-потребитель взаимодействуют с помощью разделяемого буфера, состоящего из n ячеек. Буфер содержит очередь сообщений. Производитель передает сообщение потребителю, помещая его в конец очереди. Потребитель получает сообщение, извлекая его из начала очереди. Чтобы сообщение нельзя было извлечь из пустой очереди или поместить в заполненный буфер, нужна синхронизация.

В листинге 5.3 приведен монитор, реализующий кольцевой буфер. Для представления очереди сообщений вновь использованы массив buf и две целочисленные переменные front и rear, которые указывают соответственно на первую заполненную и первую пустую ячейку. В целочисленной переменной count хранится количество сообщений в буфере. Операции с буфером deposit и fetch становятся процедурами монитора. Взаимное исключение неявно, поэтому семафорам не нужно защищать критические секции. Условная синхронизация, как показано, реализована с помощью двух условных переменных.

В листинге 5.3 оба оператора wait находятся в циклах. Это безопасный способ обеспечить истинность необходимого условия перед тем, как произойдет обращение к постоянным переменным. Это необходимо также при наличии нескольких производителей и потребителей. (Напомним, что используется порядок "сигнализировать и продолжить".)

Листинг 5.3. Реализация кольцевого буфера с помощью монитора

```
monitor Bounded_Buffer {
  typeT buf[n];    # массив некоторого типа T
  int front = 0,   # индекс первой заполненной ячейки
      rear = 0,    # индекс первой пустой ячейки
      count = 0;   # число заполненных ячеек
  ## rear == (front + count) % n
  cond not_full,   # получает сигнал, когда count < n
       not_empty; # получает сигнал, когда count > 0

  procedure deposit(typeT data) {
    while (count == n) wait(not_full);
    buf[rear] = data; rear = (rear+1) % n; count++;
    signal(not_empty);
  }

  procedure fetch(typeT &result) {
    while (count == n) wait(not_empty);
    result = buf[front]; front = (front+1) % n; count--;
    signal(not_full);
  }
}
```

Выполняя операцию `signal`, процесс просто сообщает, что теперь некоторое условие истинно. Поскольку процесс-сигнализатор и, возможно, другие процессы могут выполняться в мониторе до возобновления процесса, запущенного операцией `signal`, в момент начала его работы условие запуска может уже не выполняться. Например, процесс-производитель был приостановлен в ожидании свободной ячейки, затем процесс-потребитель извлек сообщение и запустил приостановленный процесс. Однако до того, как этому производителю пришла очередь выполняться, другой процесс-производитель мог уже войти в процедуру `deposit` и занять пустую ячейку. Аналогичная ситуация может возникнуть и с потребителями. Таким образом, условие приостановки необходимо перепроверять.

Операторы `signal` в процедурах `deposit` и `fetch` выполняются безусловно, поскольку в момент их выполнения условие, о котором они сигнализируют, является истинным. В действительности операторы `wait` находятся в циклах, поэтому операторы `signal` могут выполняться *в любой момент времени*, поскольку они просто дают подсказку приостановленным процессам. Однако программа выполняется более эффективно, когда `signal` выполняется, только если известно наверняка (или хотя бы с большой вероятностью), что некоторый приостановленный процесс может быть продолжен.

5.2.2. Читатели и писатели: сигнал оповещения

Задача о читателях и писателях была представлена в разделе 4.4. Напомним, что процесс-читатель может только читать записи базы данных, а процесс-писатель просматривает их и изменяет. Читатели могут обращаться к базе данных одновременно, писателям необходим исключительный доступ. Хотя база данных — общий ресурс, ее нельзя представить монитором, поскольку тогда читатели не смогут работать с ней параллельно (весь код внутри монитора выполняется со взаимным исключением). Вместо этого монитор используется для упорядочения доступа к базе данных. Сама база глобальна по отношению к читателям и писателям, она может находиться, например, в разделяемой памяти или во внешнем файле. Как будет показано ниже, такая базовая структура часто применяется в программах, основанных на мониторах.

В задаче о читателях и писателях упорядочивающий монитор дает разрешение на доступ к базе данных. Для этого необходимо, чтобы процессы информировали монитор о своем же-

лании получить доступ и о завершении работы с базой данных. Есть два типа процессов и по два вида действий на процесс, поэтому получаем четыре процедуры монитора: request_read, release_read, request_write, release_write. Использование этих процедур очевидно. Например, процесс-читатель перед чтением базы данных должен вызвать процедуру request_read, а после чтения — release_read.

Для синхронизации доступа к базе данных необходимо вести учет числа записывающих и читающих процессов. Как и раньше, пусть значение переменной nr — это число читателей, а nw — писателей. Это постоянные переменные монитора; при правильной синхронизации они должны удовлетворять инварианту монитора:

$$RW\text{: } (nr == 0 \vee nw == 0) \wedge nw <= 1$$

В начальном состоянии nr и nw равны 0. Их значения увеличиваются при вызове процедур запроса и уменьшаются при вызове процедур освобождения.

В листинге 5.4 представлен монитор, соответствующий этой спецификации. Для обеспечения инварианта *RW* использованы циклы while и операторы wait. В начале процедуры request_read процесс-читатель должен приостановиться, пока nw не станет равной 0; эта задержка происходит на условной переменной oktoread. Аналогично процесс-писатель в начале процедуры request_write до обнуления переменных nr и nw должен приостановиться на условной переменной oktowrite. В процедуре release_read для процесса-писателя вырабатывается сигнал, когда значение nr равно нулю. Поскольку писатели выполняют перепроверку условия своей задержки, данное решение является правильным, даже если процессы-писатели всегда получают сигнал. Однако это решение будет менее эффективным, поскольку получивший сигнал процесс-писатель при нулевом значении nr должен сно-

Листинг 5.4. Решение задачи о читателях и писателях с использованием мониторов

```
monitor RW_Controller {
  int nr = 0, nw = 0;   ## (nr == 0 ∨ nw == 0) ∧ nw <= 1
  cond oktoread;   # получает сигнал, когда nw == 0
  cond oktowrite;  # получает сигнал, когда nr == 0 и nw == 0

  procedure request_read() {
    while (nw > 0) wait(oktoread);
    nr = nr + 1;
  }

  procedure release_read() {
    nr = nr - 1;
    if (nr == 0) signal(oktowrite);
                 # запустить один процесс-писатель
  }

  procedure request_write() {
    while (nr > 0 || nw > 0) wait(oktowrite);
    nw = nw + 1;
  }

  procedure release_ write() {
    nw = nw - 1;
    signal(oktowrite); # запустить один процесс-писатель и
    signal_all(oktoread); # все процессы-читатели
  }
}
```

ва приостановиться. С другой стороны, в конце процедуры release_write точно известно, что значения обеих переменных nr и nw равны нулю. Следовательно, может продолжить работу любой приостановленный процесс. Решение в листинге 5.4 не устанавливает порядок чередования процессов-читателей и процессов-писателей. Вместо этого данная программа запускает *все* приостановленные процессы и позволяет стратегии планирования процессов определить, какой из них первым получит доступ к базе данных. Если это процесс-писатель, то приостановятся все запускаемые процессы-читатели. Если же первым получит доступ процесс-читатель, то приостановится запускаемый процесс-писатель.

5.2.3. Распределение ресурсов по схеме "кратчайшее задание": приоритетное ожидание

Условная переменная по умолчанию является FIFO-очередью, поэтому, выполняя оператор wait, процесс попадает в конец очереди ожидания. Оператор приоритетного ожидания wait(cv, rank) располагает приостановленные процессы в порядке возрастания ранга. Он используется для реализации стратегий планирования, отличных от FIFO. Здесь мы вновь обратимся к задаче распределения ресурсов по схеме "кратчайшее задание", представленной в разделе 4.5.

Для распределения ресурсов по схеме "кратчайшее задание" нужны две операции: request и release. Вызывая процедуру request, процесс либо приостанавливается до освобождения ресурса, либо получает затребованный ресурс. После получения и использования ресурса процесс вызывает процедуру release. Затем ресурс отдается тому процессу, который будет использовать его самое короткое время. Если ожидающих запросов нет, ресурс освобождается.

В листинге 5.5 представлен монитор, реализующий распределение ресурсов согласно стратегии КЗ. Постоянными переменными являются логическая переменная free для индикации того, что ресурс свободен, и условная переменная turn для приостановки процессов. Вместе они соответствуют инварианту монитора:

SJN: turn *упорядочена по времени* $\wedge$ (free $\Rightarrow$ turn *пуста*)

Процедуры в листинге 5.5 используют метод передачи условия. Приоритетный оператор wait применяется для сортировки приостановленных процессов по количеству времени, в течение которого они будут использовать ресурс. Функция empty используется для проверки, есть ли приостановленные процессы. Когда ресурс освобождается, при наличии приостановленных процессов запускается тот, которому нужно меньше всего времени, иначе ресурс помечается как свободный. Если процесс получает сигнал, то отметки об освобождении ресурса не делается, чтобы другой процесс не получил к нему доступ первым.

Листинг 5.5. Распределение ресурсов согласно стратегии "кратчайшее задание" с использованием мониторов

```
monitor Shortest_Job_Next {
  bool free = true; ## инвариант SJN: см. текст
  cond turn;        # получает сигнал, когда ресурс доступен

  procedure request(int time) {
    if (free)
      free = false;
    else
      wait(turn, time);
  }

  procedure release() {
    if (empty(turn))
```

```
      free = true;
    else
      signal(turn);
  }
}
```

5.2.4. Интервальный таймер: покрывающие условия

Обратимся к новой задаче — разработке интервального таймера, который позволяет процессу перейти в состояние сна на некоторое количество единиц времени. Такая возможность часто обеспечивается операционными системами, чтобы позволить пользователям, например, периодически выполнять служебные команды. Разработаем два решения, иллюстрирующие два полезных метода. В первом решении использованы так называемые покрывающие условия; во втором (для создания компактного и эффективного механизма задержки) — приоритетный оператор `wait`.

Монитор, реализующий интервальный таймер, представляет собой еще один пример контроллера ресурсов. Ресурсом являются логические часы. Возможны две операции с часами: `delay(interval)`, которая приостанавливает процесс на отрезок времени длительностью `interval` "тиков" таймера, и `tick`, инкрементирующая значение логических часов. Возможны и другие операции, например, получение значения часов или приостановка процесса до момента, когда часы достигнут определенного значения.

Прикладные процессы вызывают операцию `delay(interval)` с неотрицательным значением `interval`. Операцию `tick` вызывает процесс, который периодически запускается аппаратным таймером. Этот процесс обычно имеет большой приоритет выполнения, чтобы значение логических часов оставалось точным.

Для представления значения логических часов используем целочисленную переменную `tod` (time of day — время дня). Вначале ее значение равно нулю и удовлетворяет простому инварианту:

CLOCK: `tod >= 0` $\wedge$ `tod` *монотонно увеличивается на* `1`

Вызвав операцию `delay`, процесс не должен возвращаться из нее, пока часы не "натикают" как минимум `interval` раз. Абсолютная точность не нужна, поскольку приостановленный процесс не может начать работу до того, как высокоприоритетный процесс, вызывающий `tick`, сделает это еще раз.

Процесс, вызывающий операцию `delay`, сначала должен вычислить желаемое время запуска. Это делается с помощью очевидного кода:

```
wake_time = tod + interval;
```

Здесь переменная `wake_time` локальна по отношению к телу функции `delay`; следовательно, каждый процесс, вызывающий `delay`, вычисляет собственное значение времени запуска. Далее процесс должен ожидать, пока не будет достаточное число раз вызвана процедура `tick`. Для этого используется цикл `while` с условием окончания `wake_time >= tod`. Тело процедуры `tick` еще проще: она лишь увеличивает значение переменной `tod` и затем запускает приостановленные процессы.

Остается реализовать синхронизацию между приостановленными процессами и процессом, вызывающим `tick`. Один из методов состоит в использовании отдельной условной переменной для каждого условия задержки. Приостановленные процессы могут ожидать в течение разных промежутков времени, так что каждому из них нужна собственная (скрытая) условная переменная. Перед задержкой процесс записывает в постоянные переменные время, через которое он должен быть запущен. При вызове операции `tick` проверяются постоянные переменные, и при необходимости запуска процессов для их скрытых условных переменных вырабатываются сигналы. Описанный подход необходим для некоторых задач, но он более сложен и менее эффективен, чем необходимо для монитора `Timer`.

Требуемую синхронизацию намного проще реализовать, используя одну условную переменную и метод так называемого *покрывающего условия*. Логическое выражение, связанное с условной переменной, "покрывает" условия запуска всех ожидающих процессов. Когда какое-либо из покрываемых условий выполняется, запускаются все ожидающие процессы. Каждый такой процесс перепроверяет свое условие и возобновляется или вновь ожидает.

В мониторе `Timer` можно использовать одну условную переменную `check`, связанную с покрывающим условием "значение `tod` увеличено". Процессы ожидают на переменной `check` в теле функции `delay`. При каждом вызове процедуры `tick` запускаются все ожидающие процессы. Соответствующий этому описанию монитор `Timer` показан в листинге 5.6. В процедуре `tick` для запуска всех приостановленных процессов использована оповещающая операция `signal_all`.

Листинг 5.6. Интервальный таймер с покрывающим условием

```
monitor Timer {
  int tod = 0;  ## инвариант CLOCK - см. текст
  cond check;   # получает сигнал, когда tod увеличено

  procedure delay(int interval) {
    int wake_time;
    wake_time = tod + interval;
   while (wake_time > tod) wait(check);
  }

  procedure tick() {
    tod = tod + 1;
    signal_all(check);
  }
}
```

Компактное и простое решение, представленное в листинге 5.6, не достаточно эффективно для данной задачи. Применение покрывающих условий подходит только для ситуаций, когда затраты на ложные сигналы (запускается процесс, который определяет, что его условие ложно, и сразу возвращается в состояние ожидания) меньше, чем затраты на обслуживание условий всех ожидающих процессов и запуск только того процесса, для которого условие выполняется. Именно так обычно и бывает (см. упражнения в конце главы), но в данной ситуации вероятно, что процессы задерживаются на длительное время и, следовательно, будут без нужды многократно запускаться.

Используя приоритетный оператор `wait`, можно преобразовать программу в листинге 5.6 в более простую и эффективную. Для этого используем приоритетный `wait` везде, где есть статическая упорядоченность условий для различных ожидающих процессов. В данной ситуации ожидающие процессы можно упорядочить по времени их запуска. Вызванная процедура `tick` использует функцию `minrank`, чтобы определить, пришло ли время запустить первый процесс, приостановленный на переменной `check`. Если да, этот процесс получает сигнал. Этим поправкам соответствует новая версия монитора `Timer` (листинг 5.7). В процедуре `delay` теперь не нужен цикл `while`, поскольку `tick` запускает процесс только при выполнении его условия запуска. Однако операцию `signal` в процедуре `tick` нужно заключить в цикл, поскольку одного и того же времени запуска могут ожидать несколько процессов.

Итак, у нас есть три основных способа реализации условной синхронизации, при которых условия задержки зависят от переменных, локальных для ожидающих процессов. Лучше использовать приоритетное ожидание, поскольку оно дает эффективные и компактные решения, как в листингах 5.7 и 5.5. Этот способ можно применять всегда, когда условия задержки упорядочены статически.

Листинг 5.7. Интервальный таймер с приоритетным ожиданием

```
monitor Timer {
  int tod = 0; ## инвариант CLOCK - см. текст
  cond check;  # получает сигнал, когда minrank(check) <= tod

  procedure delay(int interval) {
    int wake_time;
    wake_time = tod + interval;
    if (wake_time > tod) wait(check, wake_time);
  }

  procedure tick() {
    tod = tod+1;
    while (!empty(check) && minrank(check) <= tod)
    signal(check);
  }
}
```

Второй по качеству способ — использовать переменную покрывающего условия. Он также позволяет получить компактное решение, когда у приостановленных процессов есть возможность перепроверять условия своей приостановки. Однако он неприменим, когда условия ожидания процессов зависят от состояний других ожидающих процессов. Использование переменных покрывающего условия приемлемо, пока затраты на ложные сигналы ниже, чем на ведение записей об условиях ожидания в постоянных переменных.

Третий способ — записывать условия ожидания процессов в постоянные переменные и использовать скрытые переменные условий для запуска приостановленных процессов в нужное время. Этот способ приводит к более сложным решениям, но необходим, если первые два способа неприменимы или эффективность второго способа слишком низка. В упражнениях приведены задачи, демонстрирующие преимущества и недостатки всех трех способов.

5.2.5. Спящий парикмахер: рандеву

В качестве последнего базового примера рассмотрим еще одну классическую задачу синхронизации: задачу о спящем парикмахере. У нее колоритное условие, как и у задачи об обедающих философах. Она представляет практические задачи, например планирование работы головки дискового накопителя, описанное в следующем разделе. Эта задача иллюстрирует важность отношений клиент-сервер, которые часто существуют между процессами. Для нее необходим особый тип синхронизации, называемый *рандеву*. Наконец, она прекрасно демонстрирует необходимость систематического подхода к решению задач синхронизации. Специализированные методы слишком подвержены ошибкам, чтобы использоваться для решения таких сложных задач, как эта.

(5.1) **Задача о спящем парикмахере.** В тихом городке есть парикмахерская с двумя дверями и несколькими креслами. Посетители входят через одну дверь и выходят через другую. Салон парикмахерской мал, и ходить по нему может только парикмахер и один посетитель. Парикмахер всю жизнь обслуживает посетителей. Когда в салоне никого нет, он спит в своем кресле. Когда посетитель приходит и видит спящего парикмахера, он будит его, садится в кресло и спит, пока тот занят стрижкой. Если парикмахер занят, когда приходит посетитель, тот садится в одно из свободных кресел и засыпает. После стрижки парикмахер открывает посетителю выходную дверь и закрывает ее за ним. Если есть ожидающие посетители, парикмахер будит одного из них и ждет, пока тот сядет в кресло парикмахера. Если никого нет, он снова идет спать до прихода следующего посетителя.

Посетители и парикмахер являются процессами, взаимодействующими в мониторе — парикмахерской (рис. 5.2). Посетители — это *клиенты*, которые запрашивают сервис (стрижку) у парикмахера. Парикмахер — это *сервер*, постоянно обеспечивающий сервис. Данный тип взаимодействия представляет собой пример отношений *клиент-сервер*.

Рис. 5.2. Задача о спящем парикмахере

Для реализации описанных взаимодействий парикмахерскую можно промоделировать монитором с тремя процедурами: `get_haircut` (постричься), `get_next_customer` (позвать следующего) и `finished_cut` (закончить стрижку). Посетители вызывают процедуру `get_haircut`; выход из нее происходит после того, как парикмахер закончит стрижку данного посетителя. Парикмахер циклически вызывает процедуру `get_next_customer`, приглашая клиента в свое кресло, стрижет его и выпускает из парикмахерской с помощью вызова процедуры `finished_cut`. Постоянные переменные служат для хранения состояния процессов и представления кресел, в которых процессы спят.

Действия парикмахера и посетителей необходимо синхронизировать в мониторе. Во-первых, парикмахеру и посетителю необходима встреча — *рандеву*, т.е. парикмахер должен дождаться прихода посетителя, а посетитель — освобождения парикмахера. Рандеву аналогично барьеру для двух процессов, поскольку для продолжения работы к нему должны прийти обе стороны. Однако рандеву отличается от двухпроцессного барьера тем, что парикмахер может встретиться с любым из посетителей.

Во-вторых, посетителю необходимо ждать, пока парикмахер закончит его стричь, что определяется открытием выходной двери для посетителя. Наконец, перед тем, как закрыть выходную дверь, парикмахер должен подождать, пока уйдет посетитель. Таким образом, парикмахер и посетитель проходят через последовательность синхронизированных этапов, начинающихся с рандеву.

Самый простой способ определить подобные этапы синхронизации — использовать возрастающие счетчики для запоминания числа процессов, достигших каждого этапа. У посетителей есть два важных этапа: пребывание в кресле парикмахера и выход из парикмахерской. Для этих этапов будем использовать счетчики `cinchair` и `cleave`. Парикмахер циклически проходит через три этапа: освобождение от работы, стрижка и завершение стрижки. Используем для них счетчики `bavail`, `bbusy` и `bdone`. Все счетчики в начальном состоянии имеют значение нуль. Поскольку процессы проходят свои этапы последовательно, для счетчиков выполняется следующий инвариант:

C1: `cinchair >= cleave ∧ bavail >= bbusy >= bdone`

Чтобы обеспечить рандеву посетителя и парикмахера перед началом стрижки, посетитель не может садиться в кресло парикмахера чаще, чем парикмахер освобождается от работы. Кроме того, парикмахер не может начинать стрижку чаще, чем посетители садятся в его кресло. Итак, выполняется условие:

C2: `cinchair <= bavail ∧ bbusy <= cinchair`

Наконец, посетители не могут выходить из парикмахерской чаще, чем парикмахер завершает стрижку:

C3: `cleave <= bdone`

Инвариант монитора для парикмахерской, таким образом, является конъюнкцией трех предикатов:

$$BARBER: C1 \wedge C2 \wedge C3$$

Возрастающие счетчики применимы для запоминания этапов, через которые проходят процессы, однако их значения могут возрастать неограниченно. Если синхронизация зависит только от разницы значений счетчиков, возрастания можно избежать, изменив переменные. В данной задаче есть три ключевые разности, для которых выделим три новые переменные barber, chair и open.

```
barber == bavail - cinchair
chair == cinchair - bbusy
open == bdone - cleave
```

Они инициализируются 0, а во время работы программы могут принимать значения 0 или 1. Значение barber равно 1, если парикмахер ожидает посетителя и сидит в своем кресле. Переменная chair имеет значение 1, если посетитель уже сел в кресло, но парикмахер еще не занят, а переменная open принимает значение 1, когда выходная дверь уже открыта, но посетитель еще не вышел.

Остается использовать эти условные переменные для реализации необходимой синхронизации между парикмахером и посетителями. Существуют четыре условия синхронизации: посетители дожидаются освобождения парикмахера; посетители ждут, когда парикмахер откроет дверь; парикмахер ждет прихода посетителя; парикмахер ждет ухода посетителя. Для представления этих условий нужны четыре условных переменных. Процессы ждут выполнения условий с помощью операторов wait, заключенных в циклы. В моменты, когда условия становятся истинными, процессы выполняют операцию signal.

Полное решение представлено в листинге 5.8. Эта задача значительно сложнее, чем рассмотренные ранее, поэтому имеет более сложное и длинное решение. Однако с помощью систематического подхода удается разделить всю синхронизацию на маленькие части, разработать решение для каждой из них и затем "склеить" решения.

Листинг 5.8. Монитор для задачи о спящем парикмахере

```
monitor Barber_Shop {
  int barber = 0, chair = 0, open = 0;
  cond barber_available;    # получает сигнал, когда barber > 0
  cond chair_occupied;      # получает сигнал, когда chair > 0
  cond door_open;           # получает сигнал, когда open > 0
  cond customer_left;       # получает сигнал, когда open == 0

  procedure get_haircut() {
    while (barber == 0) wait(barber_available);
    barber = barber - 1;
    chair = chair + 1; signal(chair_occupied);
    while (open == 0) wait(door_open);
    open = open - 1; signal(customer_left);
  }

  procedure get_next_customer() {
    barber = barber + 1; signal(barber_available);
    while (chair == 0) wait(chair_occupied);
    chair = chair - 1;
  }

  procedure finished_cut() {
    open = open + 1; signal(door_open);
    while (open > 0) wait(customer_left);
  }
}
```

В приведенном мониторе мы впервые видим процедуру `get_haircut`, содержащую два оператора `wait`. Дело в том, что посетитель проходит через два этапа: сначала он ждет, пока не освободится парикмахер, потом — пока не закончится стрижка.

5.3. Планирование работы диска: программные структуры

В предыдущих примерах были рассмотрены несколько небольших задач и представлены разнообразные полезные методы синхронизации. В данном разделе описаны способы организации процессов и мониторов для решения одной более крупной задачи. Затрагиваются вопросы "большого программирования", точнее, различные способы структурирования программы. Это смещение акцентов делается и в оставшейся части книги.

В качестве примера рассмотрим задачу планирования доступа к диску с перемещаемыми головками, который используется для хранения файлов. Покажем, как в разработке решения задачи применяются методы, описанные в предыдущих разделах. Особо важно, что рассматриваются три различных способа структурирования решения. Задача планирования доступа к диску — типичный представитель ряда задач планирования, и каждая из структур ее решения применима во многих других ситуациях. Вначале опишем диски с подвижными головками.

На рис. 5.3 приведена общая схема диска с подвижными головками. Он содержит несколько соединенных с центральным шпинделем пластин, которые вращаются с постоянной скоростью. Данные хранятся на поверхностях пластин. Пластины похожи на граммофонные пластинки, за исключением того, что дорожки на них образуют отдельные концентрические кольца, а не спираль. Дорожки с одинаковым относительным положением на пластинах образуют цилиндр. Доступ к данным осуществляется так: головка чтения-записи перемещается на нужную дорожку, затем ожидает, когда пластина повернется и необходимые данные пройдут у головки. Обычно одна пластина имеет одну головку чтения-записи. Головки объединены в рычаг выборки, который может двигаться по радиусу так, что их можно перемещать на любой цилиндр и, следовательно, на любую дорожку.

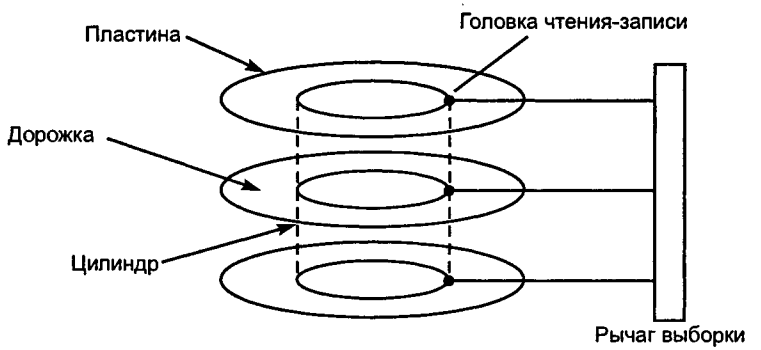


Рис. 5.3. Диск с подвижными головками

Физический адрес данных, записанных на диске, состоит из цилиндра, номера дорожки, определяющего пластину, и смещения, задающего расстояние от фиксированной точки отсчета на дорожке. Для обращения к диску программа выполняет машинозависимую инструкцию ввода-вывода. Параметрами инструкции являются физический дисковый адрес, число байтов для передачи, тип передачи (чтение или запись) и адрес буфера данных.

Время доступа к диску состоит из трех частей: *времени поиска* (перемещение головки чтения-записи на соответствующий цилиндр), *задержки вращения* и *времени передачи данных*. Время пе-

редачи данных целиком определяется количеством байтов передаваемых данных, а другие два интервала зависят от состояния диска. В лучшем случае головка чтения-записи уже находится на нужном цилиндре, а требуемая часть дорожки как раз начинает проходить под ней. В худшем случае головку чтения-записи нужно переместить через весь диск, а требуемая дорожка должна совершить полный оборот. Для дисков характерно то, что время перемещения головки от одного цилиндра к другому прямо пропорционально расстоянию между цилиндрами. Важно также, что время перемещения головки даже на один цилиндр намного больше, чем период вращения пластины. Таким образом, наиболее эффективный способ сократить время обращения к диску — минимизировать передвижения головки, сократив время поиска. (Можно сокращать и задержки вращения, но это сложно и неэффективно, поскольку они обычно очень малы.)

Предполагается, что диск используют несколько клиентов. Например, в мультипрограммной операционной системе это могут быть процессы, выполняющие команды пользователя, или системные процессы, реализующие управление виртуальной памятью. Если доступ к диску запрашивает всего один клиент, то ничего нельзя выиграть, не дав ему доступ немедленно, поскольку неизвестно, когда к диску обратится еще один клиент. Таким образом, планирование доступа к диску применяется, только когда приблизительно в одно время доступ запрашивают как минимум два процесса.

Напрашивается следующая стратегия планирования: всегда выбирать тот ожидающий запрос, который обращается к ближайшему относительно текущего положения головок цилиндру. Эта стратегия называется стратегией *кратчайшего времени поиска* (shortest-seek-time — SST), поскольку помогает минимизировать время поиска. Однако SST — несправедливая стратегия, поскольку непрерывный поток запросов для цилиндров, находящихся рядом с текущей позицией головки, может остановить обработку запросов к отдаленным цилиндрам. Хотя такая остановка обработки запросов крайне маловероятна, длительность ожидания обработки запроса здесь ничем не ограничена. Стратегия SST используется в операционной системе UNIX; системный администратор с причудами, желая добиться справедливости планирования, наверное, купил бы побольше дисков...

Еще одна, уже справедливая, стратегия планирования — перемещать головки в одном направлении, пока не будут обслужены все запросы в этом направлении движения. Выбирается клиентский запрос, ближайший к текущей позиции головки в направлении, в котором она двигалась перед этим. Если для этого направления запросов больше нет, оно изменяется. Эта стратегия встречается под разными названиями — SCAN (сканирование), LOOK (просмотр) или *алгоритм лифта*, поскольку перемещения головки похожи на работу лифта, который ездит по этажам, забирая и высаживая пассажиров. Единственная проблема этой стратегии — запрос, которому соответствует позиция сразу за текущим положением головки, не будет обслужен, пока головка не пойдет назад. Это приводит к большому разбросу времени ожидания выполнения запроса.

Третья стратегия аналогична второй, но существенно уменьшает разброс времени ожидания выполнения запроса. Ее называют CSCAN или CLOOK (буква C взята от "circular" — циклический). Запросы обслуживаются только в одном направлении, например, от внешнего к внутреннему цилиндру. Таким образом, существует только одно направление поиска, и выбирается запрос, ближайший к текущему положению головки в этом направлении. Когда запросов в направлении движения головки не остается, поиск возобновляется с внешнего цилиндра. Это похоже на лифт, который только поднимает пассажиров. (Вероятно, вниз они должны были бы идти пешком или прыгать!) В отношении сокращения времени поиска стратегия CSCAN эффективна почти так же, как и алгоритм лифта, поскольку для большинства дисков перемещение головок через все цилиндры занимает примерно вдвое больше времени, чем перемещение между соседними цилиндрами. Кроме того, стратегия CSCAN справедлива, если только поток запросов к текущей позиции головки не останавливает выполнение остальных запросов (что крайне маловероятно).

В оставшейся части данного раздела разработаны три различных по структуре решения задачи планирования доступа к диску. В первом решении планировщик (диспетчер) реализован отдельным монитором, как в решении задачи о читателях и писателях (см. листинг 5.4).

Во втором он реализован монитором, который работает как посредник между пользователями диска и процессом, выполняющим непосредственный доступ к диску. По структуре это решение аналогично решению задачи о спящем парикмахере (см. листинг 5.8). В третьем решении использованы вложенные мониторы: первый из них осуществляет планирование, а второй — доступ к диску.

Все три монитора-диспетчера реализуют стратегию CSCAN, но их нетрудно модифицировать для реализации любой стратегии планирования. Например, реализация стратегии SST приведена в главе 7.

5.3.1. Использование отдельного монитора

Реализуем диспетчер монитором, который отделен от управляемого им ресурса, т.е. диска (рис. 5.4). В решении есть три вида компонентов: пользовательские процессы, диспетчер, а также процедуры или процесс, выполняющие обмен данными с диском. Диспетчер реализован монитором, чтобы данные планирования были доступны одновременно только одному процессу. Монитор поддерживает две операции: `request` (запросить) и `release` (освободить).

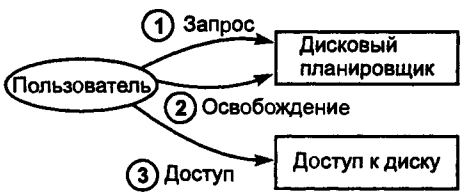


Рис. 5.4. Диспетчер доступа к диску как отдельный монитор

Для получения доступа к цилиндру `cyl` пользовательский процесс сначала вызывает процедуру `request(cyl)`, из которой возвращается только после того, как диспетчер выберет этот запрос. Затем пользовательский процесс работает с диском, например, вызывает соответствующие процедуры или взаимодействует с процессом драйвера диска. После работы с диском пользователь вызывает процедуру `release`, чтобы диспетчер мог выбрать новый запрос. Таким образом, диспетчер имеет следующий пользовательский интерфейс.

```
Disk_Scheduler.request(cyl)
работа с диском
Disk_Scheduler.release()
```

Монитор `Disk_Scheduler` играет двойную роль: он планирует обращения к диску и обеспечивает в любой момент времени доступ к диску только одному процессу. Таким образом, пользователи *обязаны* следовать указанному выше протоколу.

Предположим, что цилиндры диска пронумерованы от `0` до `MAXCYL`, и диспетчер использует стратегию CSCAN с направлением поиска от `0` до `MAXCYL`. Как обычно, важнейший шаг в разработке правильного решения — точно сформулировать его свойства. Здесь в любой момент времени только один процесс может использовать диск, а ожидающие запросы обслуживаются в порядке CSCAN.

Пусть переменная `position` указывает текущую позицию головки диска, т.е. номер цилиндра, к которому обращался процесс, использующий диск. Когда к диску нет обращений, переменной `cylinder` присваивается значение `-1`. (Можно использовать любой неправильный номер цилиндра или ввести дополнительную переменную.)

Для реализации стратегии планирования CSCAN необходимо различать запросы, которые нужно выполнить за текущий и за следующий проход через диск. Пусть эти запросы хранятся в непересекающихся множествах `C` и `N`. Оба множества упорядочены по возрастанию значе-

ния cyl, запросы к одному и тому же цилиндру упорядочены по времени вставки в множество. Таким образом, множество C содержит запросы для цилиндров, номера которых больше или равны текущей позиции, а N — для цилиндров с номерами, которые меньше или равны текущей позиции. Это выражается следующим предикатом, который является инвариантом монитора.

DISK: C *и* N *являются упорядоченными множествами* ∧
все элементы множества C >= position ∧
все элементы множества N <= position ∧
(position == -1) ⇒ (C == ∅ ∧ N == ∅)

Ожидающий запрос, для которого cyl равно position, мог бы быть в любом из множеств, но помещается в N, как описано в следующем абзаце.

Вызывая процедуру request, процесс выполняет одно из трех действий. Если переменная position имеет значение -1, диск свободен; процесс присваивает переменной position значение cyl и работает с диском. Если диск занят и выполняется условие cyl > position, то процесс вставляет значение cyl в C, иначе (cyl ≤ position) — в N. При равенстве значений cyl и position используется N, чтобы избежать возможной несправедливости планирования, поэтому запрос будет ожидать следующего прохода по диску. После записи значения cyl в подходящее множество процесс приостанавливается до тех пор, пока не получит доступ к диску, т.е. до момента, когда значения переменных position и cyl станут равными.

Вызывая процедуру release, процесс обновляет постоянные переменные так, чтобы выполнялось условие *DISK*. Если множество C не пусто, то еще есть запросы для текущего прохода. В этом случае процесс, освобождающий доступ к диску, удаляет первый элемент множества C и присваивает это значение переменной position. Если C пусто, а N — нет, то нужно начать новый проход, который становится текущим. Для этого освобождающий процесс меняет местами множества C и N (N при этом становится пустым), затем извлекает первый элемент из C и присваивает его значение переменной position. Если оба множества пусты, то для индикации освобождения диска процесс присваивает переменной position значение -1.

Последний шаг в разработке решения — реализовать синхронизацию между процедурами request и release. Здесь та же ситуация, что и в задаче с интервальным таймером: между условиями ожидания есть статический порядок, поэтому для реализации упорядоченных множеств можно использовать приоритетный оператор wait. Запросы в множествах C и N обслуживаются в порядке возрастания значения cyl. Эта ситуация похожа и на семафор FIFO: когда освобождается диск, разрешение на доступ к нему передается одному ожидающему процессу. Переменной position нужно присваивать значение того ожидающего запроса, который будет обработан следующим. По этим двум причинам синхронизацию можно реализовать эффективно, объединив свойства мониторов Timer (см. листинг 5.7) и FIFOsemaphore (см. листинг 5.2).

Для представления множеств C и N используем массив условных переменных scan[2], индексированный целыми c и n. Когда запрашивающий доступ процесс должен вставить свой параметр cyl в множество C и ждать, пока станут равны position и cyl, он просто выполняет процедуру wait(scan[c],cyl). Аналогично процесс вставляет свой запрос в множество N и приостанавливается, выполняя wait(scan[n],cyl). Кроме того, чтобы определить, пусто ли множество, используется функция empty, чтобы вычислить наименьшее значение в множестве — функция minrank, а для удаления первого элемента с одновременным запуском соответствующего процесса — процедура signal. Множества C и N меняются местами, когда это нужно, с помощью простого обмена значений c и n. (Именно поэтому для представления множеств выбран массив.)

Объединив перечисленные изменения, получим программу в листинге 5.9. В конце процедуры release значением c является индекс текущего множества обрабатываемых запросов, поэтому достаточно вставить только один оператор signal. Если в этот момент переменная position имеет значение -1, то множество scan[c] будет пустым, и вызов signal не даст результата.

Листинг 5.9. Отдельный монитор диспетчера доступа к диску

```
monitor Disk_Scheduler { ## инвариант DISK
  int position = -1, c = 0, n = 1;
  cond scan[2]; # scan[c] получает сигнал при освобождении диска

  procedure request(int cyl) {
    if (position == -1) # диск свободен, поэтому выход
      position = cyl;
    elseif (position != -1 && cyl > position)
      wait(scan[c],cyl);
    else
      wait(scan[n],cyl);
  }

  procedure release() {
    int temp;
    if (!empty(scan[c]))
      position = minrank(scan[c]);
    elseif (empty(scan[c]) && !empty(scan[n])) {
      temp = c; c = n; n = temp;   # обмен c and n
      position = minrank(scan[c]);
    }
    else
      position = -1;
    signal(scan[c]);
  }
}
```

Задачи планирования, подобные рассмотренной, наиболее трудны, какой бы механизм синхронизации не использовался. Главное в решении — точно определить порядок обслуживания процессов. Когда порядок статичен, как здесь, можно использовать приоритетные операторы `wait`. Но, как отмечалось ранее, при динамическом порядке обслуживания необходимо использовать либо скрытые условные переменные для запуска отдельных процессов, либо покрывающие условия, позволяя приостановленным процессам осуществлять планирование самостоятельно.

5.3.2. Использование посредника

Чтобы привести структуру решения к задаче планирования и распределения, желательно реализовать монитор `Disk_Scheduler` или другой контроллер ресурсов в виде отдельного монитора. Поскольку диспетчер изолирован, его можно разрабатывать независимо от других компонентов. Однако чрезмерная изоляция обычно приводит к двум проблемам:

- присутствие диспетчера видно процессам, использующим диск. Если удалить диспетчер, изменятся пользовательские процессы;
- все пользовательские процессы должны следовать необходимому протоколу: запрос диска, его использование, освобождение. Если хотя бы один процесс нарушает этот протокол, планирование невозможно.

Обе проблемы можно смягчить, если протокол использования диска поместить в процедуру, а пользовательским процессам не давать прямого доступа ни к диску, ни к диспетчеру. Однако это приводит к дополнительному уровню процедур и соответствующему снижению эффективности.

Еще одна проблема возникает, когда к диску обращается процесс драйвера диска, а не процедуры, напрямую вызываемые пользовательскими процессами. Получив доступ к диску, пользовательский процесс должен передать драйверу аргументы и получить результаты (см.

рис. 5.4). Взаимодействие пользовательского процесса и драйвера можно реализовать с помощью двух экземпляров монитора кольцевого буфера (см. листинг 5.3). Но тогда пользовательский интерфейс будет состоять из трех мониторов — диспетчера и двух кольцевых буферов, а пользователь при каждом использовании устройства должен будет делать по четыре вызова процедур монитора. Поскольку между пользователями и драйвером диска поддерживаются отношения клиент-сервер, интерфейс взаимодействия можно реализовать, используя вариант решения задачи о спящем парикмахере. Но все еще остаются два монитора — для планирования и для взаимодействия пользовательского процесса с драйвером диска.

Когда диск управляется процессом драйвера, лучше всего объединить диспетчер и интерфейс взаимодействия в один монитор. По существу, диспетчер становится посредником между пользовательскими процессами и драйвером диска, как показано на рис. 5.5. Монитор перенаправляет запросы пользователя драйверу в нужном порядке. Этот способ дает три преимущества. Первое: интерфейс диска использует только один монитор, и пользователь для получения доступа к диску должен сделать только один вызов процедуры монитора. Второе: не видно присутствия или отсутствия диспетчера. Третье: нет многошагового протокола, которому должен следовать пользователь. Таким образом, этот подход позволяет преодолеть все трудности, возникающие при выделении диспетчера в отдельный монитор.

Рис. 5.5. Диспетчер доступа к диску как посредник

В оставшейся части этого раздела показано, как преобразовать решение задачи о спящем парикмахере (листинг 5.8) в интерфейс драйвера диска, который и обеспечивает взаимодействие между клиентами и драйвером диска, и реализует стратегию планирования CSCAN. Решение задачи о спящем парикмахере нужно немного изменить. Первое: переименовать процессы, монитор и процедуры монитора, как описано ниже и показано в листинге 5.10. Второе: в процедуры монитора добавить параметры для передачи запросов пользователей (посетителей) к драйверу диска (парикмахеру) и обратной передачи результатов. По сути, "кресло парикмахера" и "выходную дверь" нужно превратить в буферы взаимодействия. Наконец, нужно добавить планирование к рандеву пользователь–драйвер диска, чтобы драйвер обслуживал предпочтительный запрос пользователя. Перечисленные изменения приводят к интерфейсу диска, схема которого показана в листинге 5.10.

Листинг 5.10. Схема монитора интерфейса диска

```
monitor Disk_Interface {
  постоянные переменные для состояния, планирования и передачи данных

  procedure use_disk(int cyl, параметры передачи и результата) {
  ждать очереди использовать драйвер
  сохранить параметры передачи в постоянных переменных
  ждать завершения передачи
  получить результаты из постоянных переменных
  }

  procedure get_next_request(someType &results) {
  выбрать следующий запрос
  ждать сохранения параметров передачи
  присвоить переменной results параметры передачи
  }
```

```
  procedure finished_transfer(someType results) {
  сохранить результаты в постоянные переменные
  ждать получения клиентом значения results
  }
}
```

Чтобы уточнить схему до полноценного решения, используем ту же базовую синхронизацию, что и в решении задачи о спящем парикмахере (см. листинг 5.8). К ней добавим планирование, как в мониторе `Disk_Scheduler` (см. листинг 5.9), и передачу параметров, как в кольцевом буфере (см. листинг 5.3). Инвариант монитора `Disk_Interface` становится, по существу, конъюнкцией инварианта для парикмахерской *BARBER*, инварианта диспетчера *DISK* и инварианта кольцевого буфера *BB* (упрощенного для одной ячейки).

Пользовательский процесс ждет очереди на доступ к диску, выполняя те же действия, что и процедура `request` монитора `Disk_Scheduler` (см. листинг 5.9). Аналогично процесс драйвера показывает, что он доступен, выполняя те же действия, что и процедура `release` монитора `Disk_Scheduler`. В начальном состоянии, однако, переменной `position` будет присваиваться значение `-2`, чтобы показать, что диск недоступен и не используется до того, как драйвер впервые вызовет процедуру `get_next_request`. Следовательно, пользователи должны ждать начала первого прохода.

Когда приходит очередь пользователя на доступ к диску, пользовательский процесс помещает свои аргументы передачи в постоянные переменные и ждет, чтобы затем извлечь результаты. После выбора следующего запроса пользователя процесс драйвера ждет получения аргументов пользователя. Затем драйвер выполняет требуемую дисковую передачу данных. После ее завершения драйвер помещает результаты и ждет их извлечения. Информация помещается и извлекается с помощью буфера с одной ячейкой. Перечисленные уточнения приводят к монитору, изображенному в листинге 5.11.

Листинг 5.11. Монитор интерфейса диска

```
monitor Disk_Interface {
  int position = -2, c = 0, n = 1, args = 0, results = 0;
  cond scan[2];
  cond args_stored, results_stored, results_retrieved;
  argType arg_area; resultType result_area;

  procedure use_disk(int cyl; argType transfer_params;
                     resultType &result_params) {
    if (position == -1)
      position = cyl;
    elseif (position != -1 and cyl > position)
      wait(scan[c],cyl);
    else
      wait(scan[n],cyl);
    arg_area = transfer_params;
    args = args+1; signal(args_stored);
    while (results == 0) wait(results_stored);
    result_params = result_area;
    results = results-1; signal(results_retrieved);
  }

  procedure get_next_request(argType &transfer_params) {
    int temp;
    if (!empty(scan[c]))
      position = minrank(scan[c]);
    elseif (empty(scan[c]) && !empty(scan[n])) {
```

```
      temp = c; c = n; n = temp;    # обмен местами c и n
      position = minrank(scan[c]);
    }
    else
      position = -1;
    signal(scan[c]);
    while (args == 0) wait(args_stored);
    transfer_params = arg_area; args = args-1;
  }

  procedure finished_transfer(resultType result_vals) {
    result_area = result_vals; results = results+1;
    signal(results_stored);
    while (results > 0) wait(results_retrieved);
  }
}
```

С помощью двух сравнительно простых изменений этот интерфейс между пользователем и драйвером диска можно сделать еще эффективнее. Во-первых, драйвер диска может быстрее начать обработку следующего пользовательского запроса, если в процедуре `finished_transfer` исключить ожидание извлечения результатов предыдущей передачи. Но это нужно делать осторожно, чтобы область результатов не была перезаписана, когда драйвер завершает следующую передачу данных, а результаты предыдущей еще не извлечены. Во-вторых, можно объединить две процедуры, вызываемые драйвером диска. Тогда при обращении к диску экономится один вызов процедуры монитора. Реализация этих преобразований требует изменить инициализацию переменной `results`. Внесение обеих поправок предоставляется читателю.

5.3.3. Использование вложенного монитора

Если диспетчер доступа к диску является отдельным монитором, пользовательские процессы должны следовать протоколу: запрос диска, работа с ним, освобождение. Сам по себе диск управляется процессом или монитором. Когда диспетчер доступа к диску является посредником, пользовательский интерфейс упрощается (пользователю достаточно сделать всего один запрос), но монитор становится значительно сложнее, как видно из сравнения листингов 5.9 и 5.11. Кроме того, решение в листинге 5.11 предполагает, что диском управляет процесс драйвера.

Третий способ состоит в объединении двух стилей с помощью *двух* мониторов: одного для планирования и одного для доступа к диску (рис. 5.6). Однако при использовании такой структуры необходимо, чтобы вызовы из монитора диспетчера освобождали исключение в мониторе доступа. Ниже мы исследуем это свойство вложенных вызовов мониторов и разработаем схему решения задачи планирования доступа к диску.

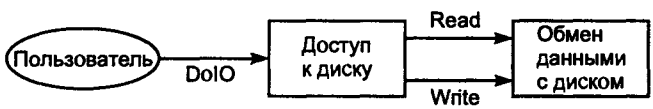


Рис. 5.6. Доступ к диску с помощью вложенных мониторов

Несколько процессов не могут получить одновременный доступ к постоянным переменным монитора, поскольку в процедурах монитора процессы выполняются со взаимным исключением. Но что произойдет, если процесс, выполняющий процедуру в одном мониторе, вызовет процедуру в другом мониторе и, следовательно, на время покинет первый? Если исключение монитора при таком вложенном вызове сохраняется, то вложенный вызов называется *закрытым*. Если при вложенном вызове исключение монитора снимается, а после него восстанавливается, то он называется *открытым*.

Ясно, что при закрытом вызове постоянные переменные монитора защищены от параллельного доступа, поскольку никакой другой процесс не может войти в монитор во время выполнения вложенного вызова. Постоянные переменные защищены от параллельного доступа и при открытом вызове, если только они не передаются по ссылке в качестве аргументов вызова. Однако открытый вызов снимает исключение, поэтому инвариант монитора должен быть истинным перед вызовом. Таким образом, у открытых вызовов семантика сложнее, чем у закрытых. С другой стороны, закрытый вызов в большей степени чреват тупиком. Например, если процесс после вложенного вызова приостановлен оператором `wait`, его уже не сможет запустить другой процесс, который должен выполнить тот же набор вложенных вызовов.

Задача планирования доступа к диску является конкретным примером, в котором возникают описанные проблемы. Как уже отмечалось, решение задачи можно изменить в соответствии с рис. 5.6. Монитор `Disk_Scheduler` из программы в листинге 5.9 заменен двумя мониторами. Пользовательский процесс делает *один* вызов операции `doIO` монитора `Disk_Access`. Этот монитор планирует доступ, как в листинге 5.9. Когда приходит очередь процесса на доступ к диску, он делает второй вызов операции `read` или `write` монитора `Disk_Transfer`. Этот второй вызов происходит *из* монитора `Disk_Access`, имеющего следующую структуру.

```
monitor Disk_Access {
  постоянные переменные такие же, как в мониторе Disk_Scheduler;

  procedure doIO(int cyl; аргументы передачи и результата) {
    действия процедуры Disk_Scheduler.request;
    вызов Disk_Transfer.read или Disk_Transfer.write;
    действия процедуры Disk_Scheduler.release;
  }
}
```

Вызовы монитора `Disk_Transfer` являются вложенными. Для планирования доступа к диску они должны быть открытыми, иначе в процедуре `doIO` не смогут одновременно находиться несколько процессов, и действия `request` и `release` станут ненужными. Здесь можно использовать открытые вызовы, поскольку в качестве аргументов для монитора `Disk_Transfer` передаются только локальные переменные (параметры процедуры `doIO`), а инвариант диспетчера доступа *DISK* перед вызовом операций `read` или `write` остается истинным.

Независимо от семантики вложенных вызовов, остается проблема взаимного исключения внутри монитора. При последовательном выполнении процедур монитора параллельный доступ к его постоянным переменным невозможен. Однако это не всегда обязательно для исключения взаимного влияния процедур. Если процедура считывает, но не изменяет постоянные переменные, то ее разные вызовы могут выполняться параллельно. Или, если процедура просто возвращает значение некоторой постоянной переменной, и оно может читаться неделимым образом, . . эта процедура может выполняться параллельно с другими процедурами монитора. Значение, возвращенное вызвавшему процедуру процессу, может не совпадать с текущим значением постоянной переменной, но так всегда бывает в параллельных программах. К примеру, можно добавить процедуру `read_clock` к монитору `Timer` в листинге 5.6 или 5.7. Независимо от того, выполняется процедура `read_clock` со взаимным исключением или нет, вызвавший ее процесс знает лишь, что возвращаемое значение не больше текущего значения переменной `tod`.

Иногда возможно одновременное безопасное выполнение даже разных процедур монитора, изменяющих постоянные переменные. Например, в предыдущих главах было показано, что потребитель и производитель могут одновременно обращаться к разным ячейкам кольцевого буфера (например, см. листинг 4.5). Если процедуры монитора должны выполняться со взаимным исключением, то такой буфер запрограммировать очень сложно. Необходимо либо представить каждую ячейку буфера в отдельном мониторе, либо буфер должен быть глобальным по отношению к процессам, которые синхронизируются с помощью мониторов, реализующих семафоры. К счастью, такие ситуации встречаются редко.

В этом разделе возникли две новые проблемы. Во-первых, используя мониторы, программист должен знать, являются вложенные вызовы (из одного монитора в другой) закрытыми или открытыми. Во-вторых, программисту нужно знать, существует ли способ ослабить неявное исключение монитора в ситуациях, когда одновременное выполнение некоторых процедур в действительности безопасно. В следующем разделе эти вопросы будут рассмотрены подробнее в контексте языка программирования Java.

5.4. Учебные примеры: язык Java

Язык программирования Java — это самое последнее увлечение компьютерного мира. Этот язык воплотил желание создавать переносимые программы для всемирной паутины WWW. Как большинство инноваций в области компьютерных наук, он многое заимствует у своих предшественников — интерпретируемых, объектно-ориентированных и параллельных языков. Он объединяет старые возможности и добавляет несколько новых, порождая интересное сочетание. Он также обеспечивает обширные библиотеки для графики, распределенного программирования и других приложений.

Язык Java является объектно-ориентированным. Хотя подробности объектно-ориентированных языков выходят за рамки данной книги, их основная идея заключается в том, что программа состоит из набора взаимодействующих объектов. Каждый объект является экземпляром класса. Класс имеет два типа членов: поля данных и методы. Поля данных представляют состояние экземпляров класса (объектов) или самого класса. Методы являются процедурами, которые используются для управления полями данных, обычно в отдельных объектах.

Нам в языке Java наиболее интересно то, что он поддерживает потоки, мониторы, описанные в этой главе, и распределенное программирование (см. часть 2). В данном разделе рассматриваются механизмы языка Java для процессов (потоков) и мониторов (синхронизированных методов), а затем их использование иллюстрируется в решении вариантов задачи о читателях и писателях. (В разделах 7.9 и 8.5 описаны механизмы языка Java для передачи сообщений и удаленного вызова методов.) В исторической справке даны ссылки на более обширную информацию по языку Java в целом и по параллельному программированию на нем в частности.

5.4.1. Класс потоков

В языке Java поток является облегченным процессом; у него есть не только собственный стек и контекст выполнения, но и прямой доступ ко всем переменным его области видимости. Потоки программируются путем расширения класса `Thread` или реализации интерфейса `Runnable`. Класс `Thread` и интерфейс `Runnable` являются частью стандартных библиотек языка Java, например пакета `java.lang`.

Поток создается как экземпляр класса `Thread`:

```
Thread foo = new Thread();
```

В результате создается новый поток `foo`. Чтобы запустить его, нужно выполнить код:

```
foo.start()
```

Операция `start` является одним из методов, определенных в классе `Thread`; существует еще множество других методов, например `stop` и `sleep`.

Приведенный выше пример в действительности не будет делать ничего, поскольку по умолчанию тело потока пусто. Точнее, метод `start` класса `Thread` вызывает метод `run` класса `Thread`, а в определении по умолчанию метод `run` не делает ничего. Таким образом, чтобы создать полезный поток, необходимо определить новый класс, который расширяет класс `Thread` (или использует интерфейс `Runnable`) и обеспечивает новое определение метода `run`. Рассмотрим, например, следующее определение класса `Simple`.

```
class Simple extends Thread {
  public void run() {
    System.out.println("this is a thread");
  }
}
```

Теперь можно создать экземпляр класса и начать выполнение нового потока таким образом.

```
Simple s = new Simple();
s.start();   //вызывает метод run() объекта s
```

Поток выведет на терминал одну строку. Если ссылка s на поток не нужна, можно упростить последние две строки до следующей:

```
new Simple().start();
```

Предыдущий пример можно запрограммировать иначе, используя интерфейс Runnable. Сначала изменим класс Simple.

```
class Simple implements Runnable {
  public void run() {
    System.out.println("this is a thread");
  }
}
```

Затем создадим экземпляр класса, передадим его конструктору класса Thread и запустим поток.

```
Runnable s = new Simple();
new Thread(s).start();
```

Преимущество использования интерфейса Runnable состоит в том, что класс Simple может расширять некоторый системный или пользовательский класс. Это невозможно при использовании первого метода, поскольку язык Java не позволяет классу расширять более одного класса.

Итак, создание и запуск потоков в языке Java состоит из четырех этапов: 1) определить новый класс, который расширяет класс Thread или реализует интерфейс Runnable; 2) определить метод run нового класса (он будет содержать все тело потока); 3) создать экземпляр нового класса, применив оператор new; 4) запустить поток с помощью метода start.

5.4.2. Синхронизированные методы

Потоки в языке Java выполняются параллельно, по крайней мере, теоретически. Следовательно, они могут одновременно обращаться к разделяемым переменным. Рассмотрим следующий класс.

```
class Interfere {
  private int data = 0;
  public void update() {
    data++;
  }
}
```

Этот класс содержит скрытое поле data, доступное только внутри класса, и предоставляет открытый метод update, который при вызове увеличивает на единицу значение переменной data. Если метод update одновременно вызовут два потока, то они, естественно, будут влиять друг на друга.

Язык Java поддерживает взаимное исключение с помощью ключевого слова synchronized, которое используется для всего метода или для последовательности операторов. Например, чтобы сделать метод update неделимым, указанный выше код можно изменить так.

```
class Interfere {
  private int data = 0;
  public synchronized void update() {
    data++;
  }
}
```

Это простой пример программирования мониторов на языке Java: постоянные переменные являются скрытыми данными класса, а процедуры монитора реализованы с помощью синхронизированных методов. В языке Java на один объект приходится по одной блокировке. Когда вызывается метод, определенный с ключевым словом `synchronized`, он ждет получения этой блокировки, выполняет тело метода и снимает блокировку.

Указанный выше пример можно запрограммировать иначе, используя ключевое слово `synchronized` для операторов внутри метода.

```
class Interfere {
  private int data = 0;
  public void update() {
    synchronized (this) {  // блокировка данного объекта
      data++;
    }
  }
}
```

Ключевое слово `this` ссылается на объект, для которого был вызван метод `update`, и, следовательно, на блокировку этого объекта. Синхронизированный оператор (с ключевым словом `synchronized`), таким образом, аналогичен оператору `await`, а синхронизированный метод — процедуре монитора.

Язык Java поддерживает условную синхронизацию с помощью операторов `wait` и `notify`; они очень похожи на операторы `wait` и `signal`, использованные ранее в этой главе. Но операторы `wait` и `notify` в действительности являются методами класса `Object`, родительского для всех классов языка Java. И метод `wait`, и метод `notify` должны выполняться внутри кода с описанием `synchronized`, т.е. при заблокированном объекте.

Метод `wait` снимает блокировку объекта и приостанавливает выполнение потока. У каждого объекта есть *одна* очередь задержки. Обычно (но не обязательно) это FIFO-очередь. Язык Java не поддерживает условные переменные , но можно считать, что на каждый синхронизированный объект приходится по одной (неявно объявленной) условной переменной.

Метод `notify` запускает поток из начала очереди задержки, если он есть. Поток, вызвавший метод `notify`, продолжает удерживать блокировку объекта, так что запускаемый поток начнет работу через некоторое время, когда получит блокировку объекта. Это значит, что метод `notify` имеет семантику "сигнализировать и продолжить". Язык Java также поддерживает оповещающий сигнал с помощью метода `notifyAll`, аналогичного процедуре `signal_all`. Поскольку у объекта есть только одна (неявная) переменная, методы `wait`, `notify` и `notifyAll` не имеют параметров.

Если синхронизированный метод (или оператор) одного объекта содержит вызов метода в другом объекте, то блокировка первого объекта во время выполнения вызова удерживается. Таким образом, вложенные вызовы из синхронизированных методов в языке Java являются закрытыми. Это не позволяет для решения задачи планирования доступа к диску с вложенными мониторами использовать структуру, изображенную на рис. 5.6. Это также может привести к зависанию программы, если из синхронизированного метода одного объекта вызывается синхронизированный метод другого объекта и наоборот.

5.4.3. Читатели и писатели с параллельным доступом

В данном и следующих двух подразделах представлен ряд примеров, иллюстрирующих аспекты параллелизма и синхронизации программ на языке Java, использование классов, деклараций и операторов. Все три программы являются завершенными: их можно откомпилировать компилятором `javac` и выполнить с помощью интерпретатора `java`. (За подробностями использования языка Java обращайтесь к своей локальной инсталляции; на Web-странице этой книги также есть исходные коды программ.)

Сначала рассмотрим параллельную версию программы читателей и писателей, в которой читатели и писатели могут обращаться к базе данных параллельно. Хотя в этой программе возможно взаимное влияние процессов, она служит для иллюстрации структуры программ на языке Java и использования потоков.

Исходный пункт программы — это класс, инкапсулирующий базу данных. Используем очень простую базу данных — одно целочисленное значение. Класс предоставляет две операции (метода), `read` и `write`. Класс определен так.

```
// базовые операции чтения или записи; без исключений
class RWbasic {
  protected int data = 0; // "база данных"

  public void read() {
    System.out.println("считано: " + data);
  }

  public void write() {
    data++;
    System.out.println("записано: " + data);
  }
}
```

Членами класса являются поле `data` и два метода, `read` и `write`. Поле `data` объявлено с ключевым словом `protected`, т.е. оно доступно только внутри класса или в подклассах, наследующих этот класс (или в других классах, определенных в том же модуле). Методы `read` и `write` объявлены с ключевым словом `public`; это значит, что они доступны везде, где доступен класс. Каждый метод при запуске выводит одну строку, которая показывает текущее значение поля `data`; метод `write` увеличивает его значение.

Следующие классы в нашем примере — `Reader` и `Writer`. Они содержат коды процессов читателя и писателя и являются расширениями класса `Thread`. Каждый из них содержит метод инициализации с тем же именем, что и у класса; этот метод выполняется при создании нового экземпляра класса. Каждый класс имеет метод `run`, в котором находится код потока. Класс `Reader` определен так.

```
class Reader extends Thread {
  int rounds;
  RWbasic RW; // ссылка на объект RWbasic

  public Reader(int rounds, RWbasic RW) {
    this.rounds = rounds;
    this.RW = RW;
  }

  public void run() {
    for (int i = 0; i < rounds; i++) {
      RW.read();
    }
  }
}
```

Аналогично объявлен класс `Writer`.

```
class Writer extends Thread {
  int rounds;
  RWbasic RW;

  public Writer(int rounds, RWbasic RW) {
    this.rounds = rounds;
```

```
    this.RW = RW;
  }

  public void run() {
    for (int i = 0; i < rounds; i++) {
      RW.write();
    }
  }
}
```

Когда создается экземпляр любого из этих классов, новый объект получает два параметра: число циклов выполнения rounds и экземпляр класса RWbasic. Методы инициализации сохраняют параметры в постоянные переменные rounds и RW. Внутри методов инициализации имена переменных предваряются ключевым словом this, чтобы различать постоянную переменную и параметр с тем же именем.

Три определенных выше класса Rwbasic, Reader и Writer — это "строительные блоки" программы для задачи о читателях и писателях, в которой читатели и писатели могут параллельно обращаться к одному экземпляру класса RWbasic. Чтобы закончить программу, нужен главный класс, который создает по одному экземпляру каждого класса и запускает потоки Reader и Writer на выполнение.

```
class Main {
  static RWbasic RW = new RWbasic();

  public static void main(String[] args) {
    int rounds = Integer.parseInt(args[0],10);
    new Reader(rounds, RW).start();
    new Writer(rounds, RW).start();
  }
}
```

Программа начинает выполнение с метода main, который имеет параметр args, содержащий аргументы командной строки. Здесь это один аргумент, задающий число циклов, которые должен выполнить каждый поток. Программа выводит последовательность строк со считанными и записанными значениями. Всего выводится 2*rounds строк, поскольку работают два потока и каждый выполняет rounds итераций цикла.

5.4.4. Читатели и писатели с исключительным доступом

Приведенная выше программа позволяет потокам параллельно работать с полем data. Изменим ее, чтобы обеспечить взаимно исключающий доступ к этому полю. Сначала определим новый класс RWexclusive, который расширяет класс RWbasic для использования синхронизированных методов read и write.

```
// взаимно исключающие методы read и write
class RWexclusive extends RWbasic {

  public synchronized void read() {
    System.out.println("считано: " + data);
  }

  public synchronized void write() {
    data++;
    System.out.println("записано: " + data);
  }
}
```

Поскольку класс RWexclusive расширяет класс RWbasic, он наследует поле data, но методы read и write переопределены для их выполнения со взаимным исключением.

Теперь изменим классы Reader и Writer, чтобы их локальные переменные RW были экземплярами класса Rwexclusive, а не класса RWbasic. Например, класс Reader примет следующий вид.

```
class Reader extends Thread {
  int rounds;
  RWexclusive RW;

  public Reader(int rounds, RWexclusive RW) {
     this.rounds = rounds;
     this.RW = RW;
  }

  public void run() {
    for (int i = 0; i < rounds; i++) {
      RW.read();
    }
  }
}
```

Аналогично изменится и класс Writer.

Наконец, изменим класс Main, чтобы он создавал экземпляры класса Rwexclusive вместо RWbasic. Для этого модифицируем его первую строку:

```
static RWexclusive RW = new RWexclusive();
```

После этих изменений потоки в новой программе будут работать с полем data класса RWbasic косвенно, вызывая синхронизированные (synchronized) методы класса RWexclusive.

5.4.5. Подлинная задача о читателях и писателях

Первый пример программы о читателях и писателях разрешал параллельный доступ к полю data, во втором доступ к полю data был взаимно исключающим. В подлинной формулировке задачи о читателях и писателях разрешены либо параллельные операции чтения, либо одна операция записи. Это можно реализовать, изменив класс Rwexclusive.

```
// параллельное чтение или исключающая запись
class ReadersWriters extends RWbasic {
  private int nr = 0;

  private synchronized void startRead() {
    nr++;
  }

  private synchronized void endRead() {
    nr--;
    if (nr == 0) notify(); // запустить ожидающие потоки
Writer
  }

  public void read() {
    startRead();
    System.out.println("считано: " + data);
    endRead();
  }

  public synchronized void write() {
```

```
        while (nr > 0) //приостановка, если есть активные потоки
    Reader
          try { wait(); }
            catch (InterruptedException ex) {return;}
        data++;
        System.out.println("записано: " + data);
        notify(); // запустить еще один ожидающий Writer
      }
    }
```

Нужно также изменить классы `Reader`, `Writer` и `Main`, чтобы они использовали этот класс вместо `RWexclusive`, но *больше ничего менять не нужно*. (Это одно из преимуществ объектно-ориентированных языков программирования.)

В классе `ReadersWriters` появились два новых локальных (с ключевым словом `private`) метода: `startRead` и `endRead`. Их вызывает метод `read` перед обращением к базе данных и после него. Метод `startRead` увеличивает значение скрытой переменной `nr`, которая учитывает число активных потоков-читателей. Метод `endRead` уменьшает значение переменной `nr`. Если она становится равной нулю, то для запуска ожидающего писателя (если он есть) вызывается процедура `notify`.

Методы `startRead`, `endRead` и `write` синхронизированы, поэтому в любой момент времени может выполняться только один из них. Следовательно, когда активен метод `startRead` или `endRead`, поток писателя выполняться не может. Метод `read` не синхронизирован, поэтому его одновременно могут вызывать несколько потоков. Если поток писателя вызывает метод `write`, когда поток читателя считывает данные, значение `nr` положительно, поэтому писатель перейдет в состояние ожидания. Писатель запускается, когда значение `nr` становится равным нулю. После работы с полем `data` писатель запускает следующий ожидающий процесс-писатель с помощью метода `notify`. Поскольку метод `notify` имеет семантику "сигнализировать и продолжить", писатель не сможет выполняться, если еще один читатель увеличит значение `nr`, поэтому писатель перепроверяет значение `nr`.

В приведенном выше методе `write` вызов `wait` находится внутри так называемого оператора `try`. Это механизм обработки исключений языка Java, который помогает программисту обрабатывать нештатные ситуации. Поскольку ожидающий поток может быть остановлен или завершен ненормально, оператор `wait` *необходимо* использовать внутри оператора `try`, обрабатывающего исключительную ситуацию `InterruptedException`. В данном коде просто происходит выход из метода `write`, если во время ожидания потока возникла исключительная ситуация.

Преимущество приведенного решения задачи о читателях и писателях по отношению к показанному ранее в листинге 5.4 состоит в том, что интерфейс потоков писателей организован в одну процедуру `write`, а не в две, `request_write()` и `release_write()`. Тем не менее в обоих решениях читатели имеют преимущество перед писателями. Подумайте, как изменить приведенное решение, чтобы отдать преимущество писателям или сделать решение справедливым (см. упражнения в конце главы).

5.5. Учебные примеры: библиотека Pthreads

Библиотека Pthreads была представлена в разделе 4.6. Там показано, как создавать потоки и синхронизировать их выполнение с помощью семафоров. Эта библиотека также поддерживает блокировки и условные переменные. Блокировки можно использовать отдельно для защиты критических секций или в комбинации с условными переменными для имитации мониторов. Ниже описаны функции библиотеки Pthreads для блокировок и условных переменных и показано, как их использовать для программирования монитора, реализующего барьер-счетчик. Заголовки описанных здесь типов данных и функций находятся в файле `pthread.h`.

5.5.1. Блокировки и условные переменные

Блокировки в библиотеке Pthreads называются *мьютексными блокировками* или просто *мьютексами*, поскольку используются для реализации взаимного исключения (*mut*ual *ex*clusion — взаимное исключение). Основной код для объявления и инициализации мьютекса по структуре аналогичен коду для потока. Сначала объявляются глобальные переменные для дескриптора мьютекса и дескриптора атрибутов мьютекса, потом инициализируются дескрипторы, а затем используются мьютексные примитивы.

Если мьютексная блокировка используется потоками одного (полновесного) процесса, но не используется потоками других процессов, то первые два шага можно упростить до таких строк.

```
pthread_mutex_t mutex;
 ...
pthread_mutex_init(&mutex, NULL);
```

Этот код инициализирует мьютекс с атрибутами по умолчанию. Критическая секция кода, использующая мьютекс, выглядит так.

```
pthread_mutex_lock(&mutex);
критическая секция;
pthread_mutex_unlock(&mutex);
```

Разблокировать мьютекс можно *только* из потока, удерживающего блокировку. Существует и неблокирующая версия оператора `lock`.

Условные переменные библиотеки Pthreads очень похожи на использованные выше в данной книге. Как и мьютексы, они имеют дескрипторы и атрибуты. Условная переменная объявляется и инициализируется атрибутами по умолчанию так.

```
pthread_cond_t cond;
 ...
pthread_cond_init(&cond, NULL);
```

Главные операции с условными переменными — ожидание (`wait`), сигнализация (`signal`) и оповещение (`broadcast`), аналогичное оператору `signal_all`. Они должны выполняться при блокировании мьютекса. Например, процедура монитора при использовании библиотеки Pthreads имитируется с помощью блокировки мьютекса в начале процедуры и ее снятия в конце.

Параметрами процедуры `pthread_cond_wait` являются условная переменная и блокировка мьютекса. Поток, который собирается ожидать, должен сначала получить блокировку. Пусть, например, поток уже выполнил вызов

```
pthread_mutex_lock(&mutex);
```

и после этого выполняет

```
pthread_cond_wait(&cond, &mutex);
```

В результате поток освобождает `mutex` и ожидает на переменной `cond`. Когда процесс возобновит выполнение после сигнализирования или оповещения, поток снова завладеет мьютексом, и тот будет заблокирован. Если код

```
pthread_cond_signal(&cond);
```

выполняется в другом потоке, запускается один поток (если был заблокирован), но сигнализирующий поток продолжает работу и удерживает мьютекс.

5.5.2. Пример: суммирование элементов матрицы

Листинг 5.12 содержит законченную программу, в которой используются блокировки и условные переменные. Программа использует `numWorkers` потоков для суммирования элементов разделяемого массива `matrix` с размером колонок и строк `size`. Хотя особой пользы от этой программы нет, ее структура и компоненты типичны для синхронных параллельных итерационных вычислений.

Листинг 5.12. Параллельное суммирование элементов матрицы с использованием библиотеки Pthreads

```
#include <pthread.h>
#include <stdio.h>
#define SHARED 1
#define MAXSIZE 2000 /* максимальный размер матрицы */
#define MAXWORKERS 4 /* максимальное число рабочих потоков */

pthread_mutex_t barrier; /* блокировка для барьера */
pthread_cond_t go;       /* условная переменная */
int numWorkers;          /* число рабочих потоков */
int numArrived = 0;      /* количество прибывших */

/* повторно используемый барьер-счетчик */
void Barrier() {
  pthread_mutex_lock(&barrier);
  numArrived++;
  if (numArrived < numWorkers)
    pthread_cond_wait(&go, &barrier);
  else {
    numArrived = 0; /* последний рабочий запускает остальные потоки */
    pthread_cond_broadcast(&go);
  }
  pthread_mutex_unlock(&barrier);
}

void *Worker(void *);
int size, stripSize;  /* size == stripSize*numWorkers */
int sums[MAXWORKERS]; /* суммы, вычисляемые каждым рабочим */
int matrix[MAXSIZE][MAXSIZE];

/*считать командную строку, инициализировать и создать потоки*/
int main(int argc, char *argv[]) {
  int i, j;
  pthread_attr_t attr;
  pthread_t workerid[MAXWORKERS];

  /* установить глобальные атрибуты потока */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* инициализировать мьютекс и условную переменную */
  pthread_mutex_init(&barrier, NULL);
  pthread_cond_init(&go, NULL);

  /* прочитать командную строку */
  size = atoi(argv[1]);
  numWorkers = atoi(argv[2]);
  stripSize = size/numWorkers;

  /* инициализировать матрицу */
  for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
      matrix[i][j] = 1;

  /* создать рабочие потоки и выйти из главного потока*/
  for (i = 0; i < numWorkers; i++)
    pthread_create(&workerid[i], &attr,
```

```
                    Worker, (void *) i);
  pthread_exit(NULL);
}

/* Каждый рабочий поток суммирует значения в одной полосе.
   После барьера поток worker(0) выводит общую сумму */
void *Worker(void *arg) {
  int myid = (int) arg;
  int total, i, j, first, last;

  /* определить первую и последнюю строки полосы */
  first = myid*stripSize;
  last = first + stripSize - 1;

  /* сложить значения в полосе */
  total = 0;
  for (i = first; i <= last; i++)
    for (j = 0; j < size; j++)
      total += matrix[i][j];
  sums[myid] = total;
  Barrier();
  if (myid == 0) { /* рабочий 0 вычисляет общую сумму */
    total = 0;
  for (i = 0; i < numWorkers; i++)
    total += sums[i];
  printf("the total is %d\n", total);
  }
}
```

Целочисленные переменные и функция `Barrier` в начале листинга реализуют повторно используемый барьер-счетчик. Они не инкапсулированы в объявление монитора, но ведут себя так, будто находятся в мониторе. Тело функции `Barrier` начинается блокировкой мьютекса и заканчивается его освобождением. Внутри функции рабочие, которые первыми придут к барьеру, ожидают на условной переменной `go` (и освобождают блокировку). Рабочий процесс, пришедший к барьеру последним, заново инициализирует (обнуляет) переменную `numArrived` и запускает остальные рабочие процессы с помощью примитива оповещения. Эти рабочие процессы будут продолжать выполнение по одному, удерживая мьютекс; каждый из них снимает блокировку с мьютекса и выходит из функции `Barrier`.

Функция `main` инициализирует разделяемые переменные и создает потоки. Последний аргумент функции `pthread_create` используется для передачи рабочему потоку уникального идентификатора. Последняя операция в функции `main` — вызов функции `pthread_exit`, в результате чего главный поток завершается, но остальные продолжают работу. Если бы вызова функции `pthread_exit` не было, то просто произошел бы выход из функции `main` и вся программа (включая все потоки) завершилась.

Каждый рабочий поток вычисляет сумму всех значений своей полосы матрицы и записывает результат в свой элемент глобального массива `sums`. Используется массив, а не отдельная переменная, чтобы избежать появления критических секций. Когда все рабочие процессы дойдут до точки барьерной синхронизации, рабочий процесс 0 вычислит и напечатает сумму.

Историческая справка

Понятие инкапсуляции данных происходит от конструкции `class` языка Simula-67. Эдсгер Дейкстра [Dijkstra, 1971] считается первым, кто начал использовать инкапсуляцию данных для управления доступом к разделяемым переменным в параллельной программе. Он на-

звал такой модуль "секретарем", но не предложил синтаксического механизма для программирования секретарей. Бринч Хансен [Brinch Hansen, 1972] выступил с той же идеей, а в своей работе [Brinch Hansen, 1973] предложил особую языковую конструкцию `shared class`.

Хоар в своей замечательной статье [Hoare, 1974] дал название мониторам и популяризировал их с помощью интересных примеров, включавших кольцевой буфер, интервальный таймер и диспетчер доступа к диску (с использованием алгоритма лифта). Условная синхронизация в варианте Хоара поддерживала порядок сигнализации "сигнализировать и срочно ожидать". Полезно сравнить решения Хоара с решениями из этой главы, в которых использован порядок SC. Хоар также представил понятие разделенного двоичного семафора и показал, как его использовать для реализации мониторов.

Язык Concurrent Pascal [Brinch Hansen, 1975] стал первым языком параллельного программирования, в который были включены мониторы. В нем есть три структурных компонента: процессы, мониторы и классы. Классы похожи на мониторы, но не могут совместно использоваться процессами и, следовательно, не нуждаются в условной синхронизации или взаимном исключении. Язык Concurrent Pascal был использован для написания нескольких операционных систем [Brinch Hansen, 1977]. Устройства ввода-вывода в нем рассматриваются как особые мониторы, реализованные с помощью системы времени выполнения (run-time system) этого языка, скрывавшей понятие прерывания.

Мониторы были включены еще в несколько языков программирования. Язык Modula был разработан создателем языка Pascal Никлаусом Виртом (Nicklaus Wirth) как системный язык для задач программирования, связанных с компьютерными системами, включая приложения для управления процессами [Wirth, 1977]. (Первый вариант языка Modula весьма отличается от своих преемников, Modula-2 и Modula-3.) В Xerox PARC был разработан язык Mesa [Mitchell et al., 1979]. Лампсон и Ределл [Lampson and Redell, 1980] превосходно описали эксперименты с процессами и мониторами в языке Mesa и исследовали технику использования *покрывающего условия* для запуска приостановленных процессов (см. раздел 5.2). В языке Pascal Plus [Welsh and Bustard, 1979] появилась операция `minrank`, которую разработчики назвали функцией `PRIORITY`. В двух книгах, [Welsh and McKeag, 1980] и [Bustard et al., 1988], есть несколько больших примеров системных программ, написанных на языке Pascal Plus.

Бринч Хансен описал интересную историю своей разработки идеи монитора и его реализации в языке Concurrent Pascal. Книга [Gehani and McGettric, 1988] содержит перепечатки многих важных работ по параллельному программированию, включая три упомянутые здесь — [Hoare, 1974], [Brinch Hansen, 1975] и [Lampson and Redell, 1980].

Рик Холт (Ric Holt) и его коллеги в университете Торонто разработали серию языков, основанных на мониторах. Например, язык CSP/k [Holt et al., 1978] — это расширение языка SP/k, который, в свою очередь, является структурированным подмножеством языка PL/I. Расширением языка Euclid для параллельного программирования стал язык Concurrent Euclid [Holt, 1983]; он использовался для UNIX-совместимого ядра, названного Tunis. Книга [Holt, 1983] содержит прекрасный обзор параллельного программирования и описания Concurrent Euclid, ОС UNIX, разработки операционных систем и ядер. Самый последний язык Холта — язык Turing Plus, расширение языка Turing [Holt and Cordy, 1988]. В языках CSP/k, Concurrent Euclid и Turing Plus используется порядок выработки сигнала "сигнализировать и ожидать" (SW). Язык Turing Plus также поддерживает порядок "сигнализировать и продолжить" (SC) и требует его использования в мониторах устройств, чтобы не задерживались обработчики прерываний.

Язык Emerald [Raj et al., 1991] несколько отличается от перечисленных выше. Он является распределенным объектно-ориентированным языком программирования. Этот язык не основан на мониторах, но включает их как механизм синхронизации. Как и в других объектно-ориентированных языках программирования, в нем объект имеет представление и управляется вызовом операций. Но в языке Emerald могут работать параллельно как объекты, так и вызовы их операций. Если необходимы взаимное исключение и условная синхронизация, то переменные и операции, которые к ним обращаются, можно определять внутри монитора. Объ-

екты языка Emerald могут быть мобильными, т.е. перемещаться во время выполнения программы. Таким образом, язык Emerald явился предшественником языка Java.

Два современных и широко распространенных языка — Java и Ada 95 — также обеспечивают механизмы для программирования мониторов. Обзор механизмов языка Java дан в разделе 5.4. Общие сведения о языке Java можно найти во многих книгах о нем; обширная информация доступна на Web-узле `www.javasoft.com`. Параллельному программированию на языке Java посвящены следующие две книги. В [Lea, 1997] описаны принципы разработки и шаблоны, а в [Hartley, 1998] затронуты многие из тем, представленных в данной книге (мониторы, семафоры и т.д.), и приведено множество примеров программ. В новой книге [Magee and Kramer, 1999] параллельное программирование рассмотрено в более широком смысле, включая моделирование параллельного поведения, и для иллюстрации идей использован язык Java.

Первая версия языка Ada, Ada 83, поддерживала параллельное программирование с помощью задач (процессов) и рандеву (см. раздел 8.2). В последней версии языка, Ada 95, появились защищенные типы, аналогичные мониторам. Механизмы параллельного программирования на языке Ada описаны в разделе 8.6. Например, в листинге 8.16 показана реализация барьера-счетчика на языке Ada с использованием защищенного типа. Основной источник сведений о языке Ada в Web расположен по адресу `adahome.com`.

Дейкстра в своей работе о семафорах [Dijkstra, 1968] представил задачу о спящем парикмахере. В статьях [Teory and Pinkerton, 1972] и [Geist and Daniel, 1987] исследованы и проанализированы многие алгоритмы планирования работы диска. В разделе 5.3 был использован алгоритм циклического сканирования (CSCAN). Ниже будет показано, как реализовать кратчайшее время поиска (раздел 7.3). Работа [Hoare, 1974] иллюстрирует алгоритм лифта.

Листер [Lister, 1977] впервые поднял вопрос о том, что делать с вложенными вызовами мониторов. Это привело к шквалу статей и писем в журнале *Operating Systems Review*, квартальном информационном бюллетене группы ACM Special Interest Group on Operating Systems (SIGOPS). Эти работы обсуждали несколько возможностей, включая следующие: запретить вложенные вызовы; использовать открытые вызовы; использовать закрытые вызовы и освобождать блокировку только в последнем мониторе; использовать закрытые вызовы и освобождать все блокировки, когда процесс блокируется, а затем получать их перед возобновлением процесса; дать возможность программисту указывать вид конкретного вызова (открытый или закрытый).

В ответ на работу Листера Парнас [Parnas, 1978] утверждал, что фундаментальным вопросом является целостность данных, которая не всегда требует взаимного исключения. В частности, параллельное выполнение процедур монитора возможно, если процессы монитора не оказывают взаимного влияния. Примерно в то же время в работе [Andrews and McGraw, 1977] была определена мониторoподобная конструкция, позволявшая программисту указать, какие процедуры могут выполняться параллельно. Другой подход состоит в использовании *маршрутных формул* [Campbell and Kolstad, 1980] — высокоуровневого механизма, который позволяет определять порядок выполнения процедур и устраняет потребность в условных переменных (см. упражнение 5.26). Язык Mesa предоставляет программисту механизмы для управления модульностью исключения. Позже, как показано в разделе 5.4, язык Java позволил программисту решать, какие методы могут выполняться параллельно, а какие — синхронизируясь.

Литература

Andrews, G. R., and J. R. McGraw. 1977. Language features for process interaction. *Proc. ACM Conference on Language Design for Reliable Software. SIGPLA.N Notices* 12, 3 (March): 114–127.

Brinch Hansen, P. 1972. Structured multiprogramming. *Comm. ACM* 15, 7 (July), 574–578.

Brinch Hansen, P. 1973. *Operating System Principles.* Englewood Cliffs. NJ: Prentice-Hall.

Brinch Hansen, P. 1975. The programming language Concurrent Pascal. *IEEE Trans. on Software Engr. SE-1*, 2 (June): 199–206.

Brinch Hansen, P. 1977. *The Architecture of Concurrent Programs.* Englewood Cliffs, NJ: Prentice-Hall.

Brinch Hansen, P. 1993. Monitors and Concurrent Pascal: A personal history. History of Programming Languages Conference (HOPL-II), *ACM Sigplan Notices* 28, 3 (March): 1–70.

Bustard, D., J. Elder, and J. Welsh. 1988. *Concurrent Program Structures.* New York: Prentice-Hall International.

Campbell, R. H., and R. B. Kolstad. 1980. An overview of Path Pascal's design and Path Pascal user manual. *SIGPLAN Notices* 15, 9 (September): 13–24.

Dijkstra, E. W. 1968. Cooperating sequential processes. In F. Genuys, ed., *Programming Languages.* New York: Academic Press, pp. 43–112.

Dijkstra, E. W. 1971. Hierarchical ordering of sequential processes. *Acta Informatica* 1, 115–138.

Gehani, N. H., and A. D. McGettrick. 1988. *Concurrent Programming.* Reading, MA: Addison-Wesley.

Geist, R., and S. Daniel. 1987. A continuum of disk scheduling algorithms. *ACM Trans. on Computer Systems* 5, 1 (February): 77–92.

Habermann, A. N. 1972. Synchronization of communicating processes. *Comm. ACM* 15, 3 (March): 171–176.

Hartley, S. J. 1998. *Concurrent Programming: The Java Programming Language.* New York: Oxford University Press.

Hoare, C. A. R. 1974. Monitors: An operating system structuring concept. *Comm. ACM* 17, 10 (October): 549–557.

Holt, R. C. 1983. *Concurrent Euclid, The UNIX System, and Tunis.* Reading, MA: Addison-Wesley.

Holt, R. C., and J. R. Cordy. 1988. The Turing programming language. *Comm. ACM* 31, 12 (December): 1410–1423.

Holt, R. C., G. S. Graham, E. D. Lazowska, and M. A. Scott. 1978. *Structured Concurrent Programming with Operating System Applications.* Reading, MA: Addison-Wesley.

Lampson, B. W., and D. D. Redell. 1980. Experience with processes and monitors in Mesa. *Comm. ACM* 23, 2 (February): 105–117.

Lea, D. L. 1997. *Concurrent Programming in Java: Design Principles and Patterns.* Reading, MA: Addison-Wesley.

Lister, A. 1977. The problem of nested monitor calls. *Operating Systems Review* 11, 3 (July): 5–7.

Magee, J., and J. Kramer. 1999. *Concurrency: State Models and Java Programs.* New York: Wiley.

Mitchell, J. G., W. Maybury, and R. Sweet. 1979. Mesa language manual, version 5.0. Xerox Palo Alto Research Center Report CSL-79-3. April.

Pamas, D. L. 1978. The non-problem of nested monitor calls. *Operating Systems Review* 12, 1 (January): 12–14.

Raj, R. K., E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. 1991. Emerald: A general purpose programming language. *Software — Practice and Experience* 21, 1 (January): 91–118.

Teorey, T. J., and T. B. Pinkerton. 1972. A comparative analysis of disk scheduling policies. *Comm. ACM* 15, 3 (March): 177–184.

Welsh, J., and D. W. Bustard. 1979. Pascal-Plus — another language for modular multiprogramming. *Software — Practice and Experience* 9, 947–957.

Welsh, J., and M. McKeag. 1980. *Structured System Programming.* New York: Prentice-Hall International.

Wirth, N. 1977. Modula: A language for modular multiprogramming. *Software — Practice and Experience* 7, 3–35.

Упражнения

5.1. Предположим, что функция `empty` недоступна, но нужно определить, есть ли процесс, ожидающий в очереди условной переменной. Разработайте способ имитации `empty`. Покажите, какой код нужно добавить перед каждым оператором `wait(cv)` и `signal(cv)` и после них. (Операцию `signal_all` можно не рассматривать.) Ваше решение должно работать при обоих порядках сигнализации — "сигнализировать и продолжить" и "сигнализировать и ожидать".

5.2. Рассмотрим монитор `Shortest_Job_Next` в листинге 5.5. Будет ли этот монитор правильно работать при порядке сигнализации "сигнализировать и ожидать"? Если да, объясните, почему. Если нет, внесите необходимые изменения.

5.3. Рассмотрим следующее решение задачи распределения ресурсов со стратегией планирования "кратчайшее задание" из раздела 5.2.

```
monitor SJN {
  bool free = true;
  cond turn;

  procedure request(int time) {
    if (!free)
      wait(turn, time);
    free = false;
  }

  procedure release() {
    free = true;
    signal(turn);
  }
}
```

Правильно ли это решение при порядке "сигнализировать и продолжить"? Правильно ли оно при порядке "сигнализировать и ожидать"? Обоснуйте свои ответы.

5.4. Следующие задачи касаются монитора в листинге 5.4. Предположим, что во всех четырех задачах использована семантика "сигнализировать и продолжить":

а) пусть нет операции `signal_all`. Измените решеиие, чтобы использовалась только операция `signal`;

б) измените решение, чтобы преимущество получали писатели, а не читатели;

в) измените решение, чтобы при одновременной попытке доступа к базе данных писатели и читатели работали с ней по очереди;

г) измените решение, чтобы читатели и писатели получали право доступа к базе данных в порядке FCFS (first come, first served — первым пришел, первым обслужен). Разрешите читателям параллельно работать с базой данных, если это не нарушает порядка FCFS получения права доступа.

5.5. Рассмотрим следующее определение семафоров [Haberman, 1972]. Пусть `na` — число попыток выполнения операции `P` процессом, `np` — число завершенных операций `P`, а `nv` — число завершенных операций `V`. Инвариант семафора имеет вид

```
np == min(na, nv).
```

Этот инвариант указывает, что приостановленный операцией `P` процесс должен запуститься и получить разрешение на выполнение, как только будет выполнено достаточное число операций `V`:

а) разработайте монитор, реализующий семафоры на основе этого определения и инварианта. Используйте порядок "сигнализировать и продолжить";

б) разработайте монитор, реализующий семафоры на основе этого определения и инварианта. Используйте порядок "сигнализировать и ожидать".

5.6. Рассмотрим задачу об обедающих философах (раздел 4.3):

а) разработайте монитор для реализации необходимой синхронизации. Монитор должен иметь две операции: `getforks(id)` и `relforks(id)`, где `id` — идентификатор философа, вызывающего операцию. Сначала определите инвариант монитора, затем разработайте тело моиитора. Ваше решение может не быть справедливым. Используйте порядок "сигнализировать и продолжить";

б) измените ответ к пункту *а*, чтобы программа стала справедливой, т.е. чтобы каждый голодный философ когда-нибудь смог поесть.

5.7. *Узкий мост.* На узкий мост едут машины с севера и юга. Машины, едущие в одном направлении, могут переезжать мост одновременно, а в противоположных — нет:

а) разработайте решение этой задачи. Машины моделируйте процессами, для синхронизации используйте монитор. Сначала определите инвариант монитора, затем разработайте тело монитора. Не беспокойтесь о справедливости планирования, не давайте преимущества ни одному из направлений. Используйте дисциплину "сигнализировать и продолжить";

б) измените ответ к пункту *а* так, чтобы обеспечить справедливость. (*Указание.* Машиины должны двигаться по очереди.)

5.8. *Задача о счете.* Несколько людей (процессов) используют общий счет. Каждый может помещать средства на счет и снимать их. Текущий баланс равен сумме всех вложенных средств минус сумма всех снятых средств. Баланс никогда не должен становиться отрицательным. Помещать средства можно без задержки (кроме случаев взаимного исключения), при изъятии средств возможна пауза, пока на счету не будет достаточной суммы:

а) разработайте монитор для решения поставленной задачи. Он должен иметь две процедуры: `deposit(amount)` и `withdraw(amount)`. Сначала определите инвариант монитора. Считайте, что аргументы функций `deposit` и `withdraw` являются положительными. Используйте порядок "сигнализировать и продолжить";

б) измените ответ к пункту *а* так, чтобы извлечение средств происходило в порядке FCFS. Например, пусть текущий баланс составляет $200, а клиент ждет извлечения $300. Если прибывает другой клиент, он должен ждать, даже если хочет снять не больше $200. Можно предположить, что есть "волшебная" функция `amount(cv)`, которая возвращает значение параметра первого процесса, приостановленного на переменной `cv`;

в) предположим, что "волшебной" функции `amount(cv)` нет. Измените ответ к пункту *б*, чтобы промоделировать ее в своем решении.

5.9. *Задача о молекуле воды.* Предположим, что атомы водорода и кислорода колеблются в пространстве, пытаясь собраться в молекулы воды. Для этого нужна взаимная синхронизация двух атомов водорода и одного атома кислорода. Пусть атомы кислорода (O) и водорода (H) моделируются процессами (потоками). Каждый атом водорода H вызывает процедуру `Hready`, когда он готов объединиться в молекулу. Аналогично каждый готовый соединиться в молекулу атом кислорода O вызывает процедуру `Oready`.

Напишите монитор, реализующий эти две процедуры. Атом водорода в процедуре `Hready` должен приостановиться, пока другой атом водорода *и* один атом кислорода не вызовут соответственно процедуры `Hready` и `Oready`. *Один* из процессов (скажем, атом кислорода O) должен вызвать процедуру `makeWater`. После выхода из процедуры `makeWater` *все три* процесса должны вернуться из своих вызовов `Hready` и `Oready`. Обеспечьте отсутствие взаимной блокировки и зависаний. Это непростая задача, так что будьте внимательны.

5.10. *Неделимое оповещение.* Пусть один процесс-производитель и `n` процессов-потребителей совместно используют кольцевой буфер, состоящий из `b` ячеек. Производитель помещает в буфер сообщения, потребители извлекают их. Каждое сообщение, помещаемое производителем, должны получить все `n` потребителей. Кроме того, каждый потребитель должен получать сообщения в том же порядке, в котором они помещались в буфер, хотя потребители могут получать сообщения в разное время. Например, разница в количестве полученных сообщений между "быстрым" и "медленным" потребителями может достигать `b`.

Разработайте монитор, который обеспечивает такое взаимодействие. Используйте порядок "сигнализировать и продолжить".

5.11. Разработайте монитор, позволяющий парам процессов обмениваться значениями. Он должен иметь только одну процедуру `exchange(int *value)`. После того как функцию `exchange` вызовут два процесса, монитор меняет местами значения аргументов и возвращает их в процессы. Монитор должен допускать повторное использование, т.е. должен обменивать параметры первой пары вызвавших процессов, второй пары и т.д. Можно использовать порядок как "сигнализировать и продолжить", так и "сигнализировать и ожидать", но *определите вначале, какую дисциплину вы используете.*

5.12. *Ужин дикарей.* Племя дикарей ест вместе из большого горшка, который вмещает `M` кусков тушеного миссионера. Когда дикарь хочет есть, он ест из горшка, если только тот не пуст. Если горшок пуст, дикарь будит повара и ждет, пока он не наполнит горшок. Поведение дикарей и повара описывается такими процессами.

```
process Savage[1:n] {
  while (true) { взять кусок из котла; есть; }
}
process Cook {
  while (true) { спать; положить в котел M кусков; }
}
```

Разработайте код для действий дикарей и повара. Используйте монитор для синхронизации. Обеспечьте отсутствие взаимной блокировки и пробуждение повара, только если котел пуст. Используйте порядок "сигнализировать и продолжить".

5.13. *Поиск, вставка и удаление.* Процессы трех типов (поиска, вставки и удаления) разделяют доступ к однонаправленному списку. Процесс поиска просматривает список, поэтому несколько таких процессов могут работать параллельно. Процесс вставки добавляет новые элементы в конец списка. Операции вставки должны исключать друг друга, чтобы не было нескольких вставок примерно в одно и то же время, но один процесс вставки может работать параллельно с любым количеством процессов поиска. Процесс удаления исключает элемент на любой позиции в списке. В каждый момент времени только один процесс удаления может иметь доступ к списку. Удаления должны взаимно исключать поиски и вставки.

Разработайте монитор для реализации описанного типа синхронизации. Сначала определите инвариант монитора. Используйте порядок "сигнализировать и продолжить".

5.14. *Выделение памяти.* Предположим, что есть две операции: `request(amount)` и `release(amount)`, где `amount` — положительное целое число. Когда процесс вызывает операцию `request`, его выполнение приостанавливается, пока не станут доступными как минимум `amount` свободных страниц. После этого процесс получает `amount` страниц памяти. Вызывая операцию `release`, процесс освобождает `amount` страниц (страницы могут выделяться и освобождаться в разных количествах):

а) разработайте монитор, реализующий процедуры `request` и `release`. Сначала определите глобальный инвариант. Не беспокойтесь о порядке обслуживания процессов. Используйте порядок "сигнализировать и продолжить" (примените покрывающее условие);

б) измените ответ к пункту *а* так, чтобы программа использовала стратегию "кратчайшая задача". Запросам меньшего числа страниц отдается предпочтение перед "большими" запросами;

в) измените ответ к пункту *а* так, чтобы программа использовала стратегию обслуживания FCFS (первым пришел — первым обслужен). Это значит, что даже при достаточном объеме доступной памяти некоторые запросы могут задерживаться;

г) предположим, что операции `request` и `release` выделяют и возвращают непрерывные страницы памяти, т.е., запрашивая две страницы, процесс ждет, пока не станут доступными две смежные страницы. Разработайте монитор, реализующий данные версии процедур `request` и `release`. Сначала выберите представление для состояния страниц памяти и определите инвариант монитора.

5.15. Пусть n процессов `P[1:n]` разделяют два принтера. Перед использованием принтера процесс `P[i]` вызывает операцию `request(printer)`. Она возвращает идентификатор свободного принтера. После использования принтера процесс `P[i]` освобождает его с помощью функции `release(printer)`. Обе функции выполняются неделимым образом:

а) разработайте монитор, реализующий операции `request` и `release`. Сначала определите инвариант монитора. Используйте порядок "сигнализировать и продолжить";

б) предположим, что каждый процесс имеет некоторый приоритет, который он передает как дополнительный аргумент функции `request`. Измените функции `request` и `release` так, чтобы принтер выделялся для ожидающего процесса с максимальным приоритетом. Если два процесса имеют одинаковые приоритеты, их запросы должны быть обработаны в порядке FCFS.

5.16. Предположим, что в компьютерном центре есть два принтера, `A` и `B`, которые похожи, но не одинаковы. Есть три типа процессов, использующих принтеры: только типа `A`, только типа `B`, обоих типов.

Разработайте монитор для выделения этих принтеров и напишите код, который процессы должны выполнять для получения или освобождения принтера. Решение должно быть справедливым при условии, что принтеры в конце концов освобождаются. Используйте порядок "сигнализировать и продолжить".

5.17. *Американские горки.* Есть `n` процессов-пассажиров и один процесс-вагончик. Пассажиры ждут очереди проехать в вагончике, который вмещает `C` человек, `C < n`. Вагончик может ехать только заполненным:

а) напишите коды процессов-пассажиров и процесса-вагончика и разработайте монитор для их синхронизации. Монитор должен иметь три операции: `takeRide`, которую вызывают пассажиры, `load` и `unload`, которые вызывает процесс-вагончик. Определите инвариант монитора. Используйте порядок "сигнализировать и продолжить";

б) измените ответ к пункту *а*, чтобы использовались `m` процессов-вагончиков, `m > 1`. Поскольку есть только одна дорога, обгон вагончиков невозможен, т.е. заканчивать движение по дороге вагончики должны в том же порядке, в котором начали. Как и ранее, вагончик может ехать только заполненным.

5.18. *Распределение файлового буфера.* Многие операционные системы типа UNIX поддерживают кэширование буферов доступа к файлам. Каждый буфер имеет размер дискового блока. Когда пользовательский процесс собирается считать дисковый блок, файловая система сначала заглядывает в кэш-память. Если блок там, файловая система просто возвращает данные пользователю. Иначе она выбирает буфер, который не использовался дольше всех, считывает в него блок данных и возвращает данные пользователю.

Аналогично, если пользовательский процесс хочет записать дисковый блок, который находится в кэш-памяти, система просто обновляет этот блок. Иначе файловая система выбирает буфер, который не использовался дольше всех, и записывает в него. Файловая система отслеживает, какие буферы изменяются (там появляются новые данные), и записывает их на диск перед тем, как использовать для другого дискового блока. (Эта стратегия кэширования называется *обратной записью* (*write-back*).)

Разработайте монитор, реализующий кэш-память для буферов в соответствии с приведенным описанием. Сначала определите необходимые процедуры и их параметры, затем разработайте тело монитора. Используйте порядок "сигнализировать и продолжить". Объясните, какие дополнительные механизмы вам понадобятся — например, часы и процесс доступа к диску.

5.19. Следующие задачи связаны с монитором спящего парикмахера (см. листинг 5.8):

а) некоторые циклы `while` можно заменить операторами `if`. Укажите, какие, и соответственно измените монитор. Считайте, что используется порядок "сигнализировать и продолжить";

б) является ли приведенный в листинге 5.8 монитор правильным при порядке "сигнализировать и ожидать"? Если да, аргументируйте свой ответ. Если нет, исправьте монитор;

в) является ли приведенный в листинге 5.8 моиитор правильным при порядке "сигнализировать и срочно ждать"? Если да, аргументируйте свой ответ. Если нет, исправьте монитор.

5.20. Предположим, что в задаче о спящем парикмахере (раздел 5.2) есть не один, а несколько парикмахеров. Разработайте монитор для синхронизации действий посетителей и парикмахеров. Сначала определите глобальный инвариант. Монитор должен иметь те же процедуры, что и в листинге 5.8. Обратите внимание на то, что там процедура `finished_cut` запускала выполнение того же клиента, с которым парикмахер встретился в процедуре `get_next_customer`. Используйте порядок "сигнализировать и продолжить".

5.21. Следующие задачи связаны с изолированным монитором диспетчера доступа к диску (см. листинг 5.9):

а) измените монитор, чтобы использовался алгоритм лифта. Сначала определите инвариант монитора, затем разработайте решение;

б) измените монитор, чтобы использовался алгоритм кратчайшего времени поиска. Сначала определите инвариант монитора, затем разработайте решение.

5.22. Следующие задачи связаны с монитором дискового интерфейса (см. листинг 5.11):

а) уточните инвариант монитора;

б) измените процедуру `finished_transfer`, чтобы процесс драйвера диска ие ждал, пока процесс пользователя извлечет результаты. Следите за тем, чтобы драйвер диска не перезаписал область результатов;

в) объедините процедуры `get_next_request` и `finished_transfer`, вызываемые драйвером диска. Тщательно инициализируйте переменные монитора.

5.23. На рис. 5.6 показано использование монитора `Disk_Access`, а в тексте описана схема его реализации. Разработайте полную реализацию монитора `Disk_Access`. Используйте стратегию планирования доступа к диску SCAN (алгоритм лифта). Сначала определите соответствующий инвариант монитора, затем разработайте тело процедуры `doIO`. Не беспокойтесь о реализации монитора `Disk_Transfer`, просто укажите его вызовы в соответствующих точках монитора `Disk_Access` (считайте эти вызовы открытыми).

5.24. Основное различие между дисциплинами сигнализации в мониторах состоит в порядке выполнения процессов. Но можно промоделировать семантику одной дисциплины сигнализации, используя другую дисциплину. Например, можно преобразовать монитор, использующий одну дисциплину, в монитор, который ведет себя так же, но использует другую дисциплину. Для этого нужно изменить код и добавить дополнительные переменные:

а) покажите, как промоделировать дисциплину "сигнализировать и продолжить" с помощью "сигнализировать и ожидать". Проиллюстрируйте свое решение, используя один из мониторов, приведенных в данной главе;

б) покажите, как промоделировать дисциплину "сигнализировать и ожидать" с помощью "сигнализировать и продолжить". Постройте пример, иллюстрирующий вашу модель;

в) покажите, как промоделировать дисциплиию "сигнализировать и ожидать" с помощью "сигнализировать и срочно ожидать". Постройте пример, иллюстрирующий вашу модель.

5.25. Предположим, что ввод и вывод терминала обеспечиваются двумя процедурами:

```
getline(string &str, int &count)
putline(string str)
```

Прикладной процесс вызывает процедуру `getline` для получения следующей входной строки, а процедуру `putline` — для отправки строки на дисплей. Выход из вызова процедуры `getline` происходит после появления следующей входной строки; в результате аргументу `str` присваивается содержание этой строки, а переменной `count` — ее длина. Строка содержит не больше, чем `MAXLINE` символов, она заканчивается символом `NEWLINE`, который сам не входит в строку.

Предположим, что для ввода и вывода используются буферы, т.е. до `n` строк ввода могут находиться в ожидании извлечения функцией `getline` и до `n` строк вывода могут запоминаться перед их печатью. Также предположим, что входные строки дают эхо на дисплей, т.е. каждая полная входная строка отсылается еще и на дисплей. Строки "подготавливаются", т.е. символы удаления строки и предыдущего символа обрабатываются процедурой `getline` и не возвращаются прикладному процессу.

Разработайте монитор, реализующий процедуры `getline` и `putline`. Считайте, что есть два процесса драйверов устройств. Один считывает символы с клавиатуры, другой записывает строки на дисплей. Для вызова этих процессов ваш монитор должен иметь дополнительные процедуры.

5.26. Маршрутные формулы (МФ) являются высокоуровневым механизмом для определения синхронизации между процедурами модуля [Campbell and Kolstad, 1980]. Их можно использовать, чтобы указывать, какие процедуры должны выполняться со взаимным исключением, а какие могут работать параллельно. МФ делают ненужными условные переменные и явные вызовы операторов `wait` и `signal`. Синтаксис открытой МФ определяется следующей грамматикой (в виде форм Бэкуса–Наура).

описание_маршрута ::= "path" *список* "end"
список ::= *последовательность* {"," *последовательность*}
последовательность ::= *элемент* {";" *элемент*}
элемент ::= *предел* ":" "(" *список* ")" | "[" *список* "]" |
"(" *список* ")" | *идентификатор*

Фигурные скобки обозначают нуль или несколько вхождений элементов, взятых в скобки, а кавычки выделяют буквенные (терминальные) символы. В альтернативах для *элемента* *предел* является положительным целым числом, *идентификатор* — именем процедуры.

Оператор "запятая" в списке не налагает никаких ограничений синхронизации. Оператор "точка с запятой" в последовательности устанавливает ограничение: первый элемент должен быть выполнен до начала любого выполнения второго элемента, которое должно быть закончено до выполнения третьего элемента и т.д. *Предел* (перед ":") ограничивает число одновременно активных элементов из списка в скобках. Оператор "квадратные скобки" допускает одновременную работу любого числа заключенных в эти скобки элементов. Например, следующая МФ определяет, что любое количество вызовов процедур `a` или `b` могут выполняться параллельно, но процедуры `a` и `b` выполняются со взаимным исключением:

```
path 1: ([a], [b]) end
```

Приведите маршрутную формулу, которая выражает синхронизацию:

а) кольцевого буфера с `n` ячейками (см. листинг 5.3). Буфер обрабатывается операциями `deposit` и `fetch`, которые должны выполняться со взаимным исключением;

б) кольцевого буфера с `n` ячейками при максимальном параллелизме, т.е. экземпляры операций `deposit` и `fetch`, обращаясь к разным ячейкам, могут выполняться параллельно;

в) в задаче об обедающих философах (см. раздел 4.3). Объясните ответ;

г) в задаче о читателях и писателях (см. листинг 5.4). Объясните ответ;

д) в задаче о спящем парикмахере (см. листинг 5.8). Объясните ответ.

Покажите, как реализовать маршрутные формулы с помощью семафоров. Пусть даны маршрутные формулы, определяющие синхронизацию набора процедур. Покажите, какой код необходимо вставить в начало и конец каждой процедуры для обеспечения требуемой синхронизации. Начните с простых примеров и постепенно обобщите свое решение.

5.27. Следующие задачи касаются программ на языке Java из раздела 5.4. Исходный код программ можно загрузить с Web-узла этой книги (см. предисловие):

а) напишите простую программу с двумя классами. Один определяет поток с методом `run`, который печатает строку, второй является главным и создает поток. В главном классе создайте поток и запустите его на выполнение, как описано в тексте книги. Затем попробуйте вызвать метод `run` прямо, а не косвенно через процедуру `start`. (Если поток имеет имя `s`, то метод вызывается как `s.run`.) Что происходит? Почему?

б) разработайте более реалистичные модели программ для задач читателей и писателей. Используйте несколько читателей и писателей и измените их, чтобы можно было следить за правильностью синхронизации. Возможно, измените базу данных на что-то более реальное или хотя бы обеспечьте большую длительность работы процедур `read` и `write`. Пусть также каждый поток на некоторое случайное время переходит в режим сна до (или после) каждого обращения к базе данных. Язык Java обеспечивает несколько методов, которые можно использовать в вашей модели — `nap`, `age`, `random` и `seed`. Напишите краткий отчет о своих наблюдениях;

в) измените класс `ReadersWriters`, чтобы предпочтение отдавалось писателям. Повторите моделирование из пункта *б* и подведите итоги своих наблюдений;

г) измените класс `ReadersWriters`, чтобы он обеспечивал справедливое планирование. Повторите моделирование из пункта *б* и подведите итоги своих наблюдений.

5.28. Разработайте программу на языке Java для моделирования задачи об обедающих философах (см. раздел 4.3). В программе должно быть `5` потоков философов и класс, реализующий монитор для их синхронизации. В мониторе должно быть два метода: `getforks(id)` и `relforks(id)`, где `id` — целое число от `1` до `5`. Философы могут есть и спать в течение случайных промежутков времени. Добавьте к программе вывод сообщений для отслеживания действий философов. Напишите краткий отчет с итогами ваших наблюдений.

Глава 6

Реализация

В предыдущих главах были определены механизмы параллельного программирования с разделяемыми переменными и приведены примеры их использования. Современные мультипроцессоры предоставляют системному программисту необходимый минимум машинных инструкций для реализации блокировок и барьеров (глава 3). Некоторые мультипроцессоры обеспечивают аппаратную поддержку процессов, переключения контекста, блокировок, барьеров, а иногда даже и циклической блокировки для семафоров. Однако в целом механизмы параллельного программирования реализуются программно.

В данной главе описаны программные реализации процессов, семафоров и мониторов. Основой их реализации служит *ядро* — небольшой набор структур данных и подпрограмм, необходимых для любой параллельной программы. (Термин "ядро" используется, чтобы показать, что код и данные — общие для всех программных модулей.) Задача ядра — обеспечить каждый процесс виртуальным процессором, чтобы у процесса возникала иллюзия, что он выполняется на отдельном процессоре. Структуры данных представляют состояния процессов, семафоров и условных переменных. Подпрограммы реализуют операции над этими структурами данных; каждая такая операция называется *примитивной*, поскольку выполняется неделимым образом.

В разделе 6.1 описано ядро, реализующее процессы на одном процессоре. В разделе 6.2 это ядро расширено для использования с мультипроцессором. В разделах 6.3 и 6.4 в ядро добавлены базовые операции для поддержки семафоров и мониторов соответственно. Наконец, в разделе 6.5 описана реализация мониторов с помощью семафоров. Наша цель состоит в реализации процессов и синхронизации, поэтому здесь не затронуты такие практические вопросы, как динамическое распределение ресурсов и приоритетное планирование.

6.1. Однопроцессорное ядро

В предыдущих главах для определения параллельной работы использовались операторы `co` и декларации `process`. Процессы — это просто частные случаи операторов `co`, поэтому здесь основное внимание уделяется реализации операторов `co`. Рассмотрим следующий фрагмент программы.

```
S0;
co  P1: S1; // ... // Pn: Sn; oc
Sn+1;
```

`Pi` — это имена процессов. `Si` обозначают списки операторов и необязательные декларации локальных переменных процесса `Pi`.

Для реализации приведенного фрагмента программы необходимы три механизма:

- создания процессов и их запуска на выполнение;
- остановки (и уничтожения) процесса;
- определения момента завершения оператора `co`.

Примитив — это процедура, реализованная ядром так, что она выполняется как неделимое действие. Процессы создаются и уничтожаются с помощью двух примитивов ядра: `fork` и `quit`.

Когда процесс запускает примитив fork, создается еще один процесс, готовый к запуску. Аргументы примитива fork передают адрес первой выполнимой инструкции нового процесса и любые другие данные, необходимые для определения его начального состояния, например, параметры процесса. Новый процесс называется *сыновним*, а процесс, выполняющий примитив fork, — *родительским*. Вызывая примитив quit, процесс прекращает свое существование. У примитива quit аргументов нет.

Третий примитив ядра, join, используется для ожидания завершения процессов и, следовательно, для определения момента завершения оператора co. В частности, когда родительский процесс выполняет примитив join, он ждет завершения сыновнего процесса, который до этого был порожден операцией fork. Аргументом операции join является имя сыновнего процесса. (Примитив join используется и без аргументов — тогда процесс ждет завершения *любого* из сыновних процессов и, возможно, возвращает его идентификатор.)

Итак, для реализации указанного выше фрагмента можно использовать три описанных примитива fork, join и quit. Каждый сыновний процесс Pi выполняет следующий код:

```
Si; quit();
```

Главный процесс выполняет такой код.

```
S0;
for [i = 1 to n]    # создать сыновние процессы
  fork(Pi);
for [i = 1 to n]    # ожидать завершения каждого из них
  join(Pi);
Sn+1;
```

Предполагается, что главный процесс создается неявно и автоматически начинает выполняться. Считается также, что к моменту запуска главного процесса код и данные всех процессов уже записаны в память.

Во втором цикле for приведенная программа ждет выхода из сыновнего процесса 1, затем выхода из процесса 2 и т.д. При использовании примитива join без параметров ожидается завершение всех n сыновних процессов в произвольном порядке. Если сыновние процессы были объявлены с помощью декларации process и, следовательно, должны выполняться в фоновом режиме, то главный процесс создаст их таким же образом, но не будет ждать выхода из них.

Теперь представим однопроцессорное ядро, реализующее операции fork, join и quit. Также опишем, как планировать процессы, чтобы каждый процесс периодически получал возможность выполняться, т.е. представим диспетчер со стратегией планирования, справедливой в слабом смысле (см. определение в разделе 2.8).

Любое ядро содержит структуры данных, которые представляют процессы и три базовых типа процедур: обработчики прерываний, сами примитивы и диспетчер. Ядро может включать и другие структуры данных или функции — например, дескрипторы файлов и процедуры доступа к файлам. Сосредоточимся, однако, на той части ядра, которая реализует процессы.

Есть два основных способа организовать ядро:

- в виде монолитного модуля, в котором каждый примитив ядра выполняется неделимым образом;
- в виде параллельной программы, в которой несколько пользовательских процессов одновременно могут выполнять примитивы ядра.

Здесь используется первый метод, поскольку для небольшого однопроцессорного ядра он самый простой. Второй метод будет использован для многопроцессорного ядра в следующем разделе.

В ядре каждый процесс представлен *дескриптором процесса*. Когда процесс простаивает, его дескриптор хранит *состояние* процесса, или *контекст*, — всю информацию, необходимую для выполнения процесса. Состояние (контекст) включает адрес следующей инструкции, которую должен выполнять процесс, и содержимое регистров процессора.

Ядро начинает выполняться, когда происходит прерывание. Прерывания можно разделить на две широкие категории: внешние прерывания от периферийных устройств и внутренние прерывания, или ловушки, которые возбуждаются выполняемым процессом. Когда происходит прерывание, процессор автоматически сохраняет необходимую информацию о состоянии процесса, чтобы прерываемый процесс можно было продолжить. Затем процессор входит в *обработчик прерывания*; обычно каждый тип прерываний имеет свой обработчик.

Для запуска примитива ядра процесс вызывает внутреннее прерывание, выполняя машинную инструкцию, которая называется *вызовом супервизора* (supervisor call — SVC) или *ловушкой* (trap). В инструкции SVC процесс передает аргумент, который задает вызываемый примитив; остальные аргументы передаются в регистрах. Обработчик прерывания SVC сначала сохраняет состояние выполняемого процесса, затем вызывает соответствующий примитив, реализованный в виде процедуры ядра. Примитив, завершаясь, вызывает процедуру `dispatcher` (процессорный диспетчер). Процедура `dispatcher` выбирает процесс для выполнения и загружает его состояние. Состояние процесса называется контекстом, поэтому последнее действие процедуры `dispatcher` носит название *переключение контекста*.

Чтобы обеспечить неделимое выполнение примитивов, обработчик прерывания в начале своей работы должен запретить дальнейшие прерывания, а диспетчер в конце — разрешить их. Когда возникает прерывание, все остальные прерывания автоматически запрещаются аппаратной частью; в качестве побочного эффекта нагрузки состояния процесса ядро вновь разрешает прерывания. (В некоторых машинах прерывания разделены на несколько уровней, или классов. В этой ситуации обработчик прерывания запрещает только те прерывания, которые могут повлиять на обрабатываемое.)

На рис. 6.1 показаны компоненты ядра и поток управления внутри него. Поток идет в одном направлении: от обработчиков прерываний через примитивы к процедуре `dispatcher` и затем обратно к активному процессу. Следовательно, возвращений из вызовов внутри ядра не происходит, поскольку вместо возврата к тому процессу, который выполнялся при входе в ядро, оно часто начинает выполнение другого процесса.

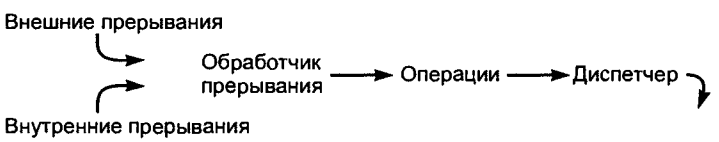


Рис. 6.1. Компоненты ядра и направление потока команд

Дескрипторы процессов представим таким массивом:

```
processType processDescriptor[maxProcs];
```

Тип `processType` — это тип структуры (записи), описывающий поля в дескрипторе процесса. Родительский процесс просит ядро создать сыновний процесс, вызывая примитив `fork`, который выделяет и инициализирует пустой дескриптор. Когда процедура ядра `dispatcher` планирует процессы, она ищет дескриптор процесса, имеющий право выполнения. Процедуры `fork` и `dispatcher` можно реализовать путем поиска в массиве дескрипторов процессов при условии, что каждая запись содержит поле, указывающее, занят ли данный элемент массива. Однако обычно используются два списка: *список свободных* (пустых дескрипторов) и *список готовых к работе* (там содержатся дескрипторы процессов, ожидающих своей очереди на выполнение). Родительские процессы, ожидающие завершения работы сыновних процессов, запоминаются в дополнительном *списке ожидания*. Наконец, ядро содержит переменную `executing`, значением которой является индекс дескриптора процесса, выполняемого в данный момент.

Предполагается, что при инициализации ядра, которая происходит при "загрузке" процессора, создается один процесс и переменная `executing` указывает на его дескриптор. Все остальные дескрипторы помещаются в список свободных. Таким образом, в начале выполнения списки готовых и ожидающих процессов пусты.

При описанном представлении структур данных ядра примитив `fork` удаляет дескриптор из списка свободных, инициализирует его и вставляет в конец списка готовых к выполнению. Примитив `join` проверяет, произошел ли уже выход из сыновнего процесса, и, если нет, блокирует выполняющийся (родительский) процесс. Примитив `quit` запоминает, что произошел выход из процесса, помещает дескриптор выполняемого процесса в список свободных, запускает родительский процесс, если он ждет, и присваивает переменной `executing` значение нуль, чтобы указать процедуре `dispatcher`, что процесс больше не будет выполняться.

Процедура `dispatcher`, вызываемая в конце примитива, проверяет значение переменной `executing`. Если это значение не равно нулю, то текущий процесс должен продолжить работу. Иначе процедура `dispatcher` удаляет первый дескриптор из списка готовых к выполнению и присваивает переменной `executing` значение, указывающее на этот процесс. Затем процедура `dispatcher` загружает состояние этого процесса. Здесь предполагается, что список готовых к выполнению процессов является очередью с обработкой типа "первым пришел — первым ушел" (FIFO).

Осталось обеспечить справедливость выполнения процесса. Если бы выполняемые процессы всегда завершались за конечное время, то описанная реализация ядра обеспечивала бы справедливость, поскольку список готовых к выполнению процессов имеет очередность FIFO. Но если есть процесс, ожидающий выполнения некоторого условия, которое в данный момент ложно, он заблокируется навсегда, если только его не заставить освободить процессор. (Процесс также зацикливается навсегда, если в результате ошибки программирования содержит бесконечный цикл.) Для того чтобы процесс периодически освобождал управление процессором, можно использовать интервальный таймер, если, конечно, он реализован аппаратно. Тогда в сочетании с очередностью FIFO обработки списка готовых процессов каждый процесс периодически будет получать возможность выполнения, т.е. стратегия планирования будет справедливой в слабом смысле.

Интервальный таймер — это аппаратное устройство, которое инициализируется положительным целым числом, затем уменьшает свое значение с определенной частотой и возбуждает прерывание таймера, когда значение становится нулевым. Такой таймер используется в ядре следующим образом. Вначале, перед загрузкой состояния следующего выполняемого процесса, процедура `dispatcher` инициализирует таймер. Затем, когда возникает прерывание таймера, обработчик этого прерывания помещает дескриптор процесса, на который указывает переменная `executing`, в конец списка готовых к выполнению процессов, присваивает переменной `executing` значение `0` и вызывает процедуру `dispatcher`. В результате образуется круговая очередность выполнения процессов.

Собрав вместе все описанные выше части, получим схему однопроцессорного ядра (листинг 6.1). Предполагается, что побочный результат запуска интервального таймера в процедуре `dispatcher` состоит в запрете всех прерываний, которые могут возникнуть в результате достижения таймером нулевого значения во время выполнения ядра. При таком необходимом дополнительном условии выбранный для выполнения процесс не будет прерван немедленно и не потеряет управления процессором.

Листинг 6.1. Схема однопроцессорного ядра

```
processType processDescriptor[maxProcs];
int executing = 0;      # индекс выполняемого процесса
объявления переменных для списков свободных, готовых к работе и ожидающих процессов;

SVC_Handler: {     # вход с запрещенными прерываниями
  сохранить состояние выполняемого процесса (с индексом executing);
  определить, какой примитив был вызван, и запустить его;
}
```

```
Timer_Handler: {     # вход с запрещенными прерываниями
  вставить дескриптор процесса executing в конец списка готовых к работе;
  executing = 0;
  dispatcher();
}

procedure fork(начальное состояние процесса) {
  извлечь дескриптор из списка свободных и инициализировать его;
  вставить дескриптор в конец списка готовых к работе;
  dispatcher();
}

procedure quit() {
  записать, что процесс executing завершился;
  вставить дескриптор процесса executing в конец списка свободных;
  executing = 0;
  if (родительский процесс ждет завершения этого сыновнего) {
    удалить родительский процесс из списка ожидающих;
    поместить родительский процесс в список готовых к работе; }
  dispatcher();
}

procedure join(имя сыновнего процесса) {
  if (сыновний процесс еще не завершился) {
    поместить дескриптор процесса executing в список ожидания;
    executing = 0;
  }
  dispatcher();
}

procedure dispatcher() {
  if (executing == 0) { # текущий процесс заблокирован или завершен
    удалить дескриптор из начала списка готовых к работе;
    поместить указание на него в переменную executing;
  }
  запустить интервальный таймер;
  загрузить состояние процесса, на который указывает executing;
        # с разрешенными прерываниями
}
```

В ядре (см. листинг 6.1) игнорируются такие возможные особые ситуации, как пустой список свободных дескрипторов при вызове примитива `fork`. В данной реализации также предполагается, что всегда есть хотя бы один готовый к работе процесс. На практике ядро всегда имеет один процесс-"бездельник", который запускается, когда другой работы нет; как это сделать, показано в следующем разделе. Не учтено еще несколько моментов, возникающих на практике, — например, обработка прерываний ввода-вывода, управление устройствами, доступ к файлам и управление памятью.

6.2. Многопроцессорное ядро

Мультипроцессор с разделяемой памятью содержит несколько процессоров и память, доступную каждому из них. Расширить однопроцессорное ядро для работы на мультипроцессорных машинах относительно нетрудно. Вот наиболее важные изменения:

- процедуры и структуры данных ядра должны храниться в разделяемой памяти;
- доступ к структурам данных должен осуществляться со взаимным исключением, когда необходимо избегать взаимного влияния;
- процедуру `dispatcher` следует преобразовать для использования с несколькими процессорами.

Однако есть нюансы, которые связаны с особенностями мультипроцессоров и описаны ниже.

Предположим, что внутренние прерывания (ловушки) обслуживаются тем процессором, на котором выполняется процесс, вызвавший такое прерывание, и у каждого процессора есть интервальный таймер. Допустим также, что операции ядра и процессы могут быть выполнены на любом процессоре. (В конце раздела описано, как привязать процессы к процессорам, что делать с эффектами кэширования и неодинаковыми временами доступа к памяти.)

Когда процессор прерывается, он входит в ядро и запрещает дальнейшие прерывания *для данного процессора*. Вследствие этого выполнение в ядре становится неделимым на данном процессоре, но остальные процессоры, тем не менее, могут одновременно выполняться в ядре.

Чтобы предотвратить взаимное влияние процессоров, можно сделать критической секцией все ядро, но это плохо по двум причинам. Во-первых, нет необходимости запрещать некоторые безопасные параллельные операции, поскольку критичен только доступ к таким разделяемым ресурсам, как списки дескрипторов свободных и готовых к работе процессов. Во-вторых, если все ядро заключить в критическую секцию, она будет слишком длинной, что снижает производительность и увеличивает число конфликтов в памяти из-за переменных, реализующих протокол критической секции. Следующий принцип позволяет получить намного более удачный вариант решения.

(6.1) **Принцип блокировки мультипроцессора**. Делайте критические секции короткими, защищая каждую критическую структуру данных отдельно. Для каждой структуры данных используйте отдельные критические секции с отдельными переменными для протоколов входа и выхода.

В нашем ядре критическими данными являются списки дескрипторов свободных, готовых к работе и ожидающих процессов. Для защиты доступа к ним можно использовать любой из протоколов критической секции, описанных выше в книге. Для конкретного мультипроцессора выбор делается на основе доступных машинных инструкций. Например, если есть инструкция "извлечь и сложить", можно использовать простой и справедливый алгоритм билета (см. листинг 3.9).

Предполагается, что ловушки обрабатываются тем же процессором, в котором они возникают, и каждый процессор имеет свой собственный интервальный таймер, поэтому обработчики прерываний от ловушек и от таймера остаются почти такими же, как и для однопроцессорного ядра. У них есть только два отличия: переменная `executing` теперь должна быть массивом с отдельным элементом для каждого из процессоров, а процедура `Timer_Handler` должна блокировать и освобождать список готовых к выполнению процессов.

Код всех трех примитивов ядра в основном также остается неизменным. Здесь нужно учесть, что переменная `executing` стала массивом, и защитить критическими секциями списки дескрипторов свободных, готовых к работе и ожидающих процессов.

Наибольшие изменения касаются процедуры `dispatcher`. До этого в нашем распоряжении был один процессор, и предполагалось, что всегда есть процесс для выполнения на нем. Теперь процессов может быть меньше, чем процессоров, поэтому некоторые процессоры могут простаивать. Когда создается новый процесс (или запускается после прерывания ввода-вывода), он должен быть привязан к незанятому процессору, если такой есть. Это можно сделать одним из трех способов:

- заставить каждый незанятый процессор выполнять специальный процесс, который периодически просматривает список дескрипторов готовых к работе процессов, пока не найдет готовый к работе процесс;

- заставить процессор, выполняющий `fork`, искать свободный процессор и назначать ему новый процесс;
- использовать отдельный процесс-диспетчер, который выполняется на отдельном процессоре и постоянно пытается назначать свободным процессорам готовые к работе процессы.

Первый метод наиболее эффективен, в частности, поскольку свободным процессорам нечего делать до тех пор, пока они не найдут процесс для выполнения.

Когда диспетчер определяет, что список готовых к работе процессов пуст, он присваивает переменной `executing[i]` значение, указывающее на дескриптор бездействующего процесса, и загружает его состояние. Код бездействующего процесса показан в листинге 6.2. По существу, процесс `Idle` — "сам себе диспетчер". Сначала, пока в списке готовых к работе процессов нет элементов, он зацикливается, затем удаляет из списка дескриптор процесса и начинает выполнение этого процесса. Чтобы не было конфликтов в памяти, процесс `Idle` не должен непрерывно просматривать или блокировать и разблокировать список готовых к работе процессов, поэтому используем протокол "проверить-проверить-установить", аналогичный по структуре приведенному в листинге 3.4. Поскольку список готовых к работе процессов может стать пустым после того, как процесс `Idle` захватит его блокировку, необходима повторная проверка.

Листинг 6.2. Код бездействующего процесса

```
process Idle {
  while (executing[i] == бездействующий процесс) {
    while (список готовых пуст) Задержка;
    заблокировать список готовых;
    if (список готовых не пуст) {
      удалить дескриптор из начала списка готовых;
      установить executing[i] на него;
    }
    снять блокировку со списка готовых;
  }
  запустить интервальный таймер на процессоре i;
  загрузить состояние executing[i]; # с разрешенными прерываниями
}
```

Осталось обеспечить справедливость планирования. Вновь используем таймеры, чтобы заставить процессы, выполняемые вне ядра, освобождать процессоры. Предполагаем, что каждый процессор имеет свой таймер, который используется так же, как и в однопроцессорном ядре. Но одних таймеров недостаточно, поскольку теперь процессы могут быть приостановлены в ядре в ожидании доступа к разделяемым структурам данных ядра. Таким образом, нужно использовать справедливое решение для задачи критической секции, например, алгоритм разрыва узлов, алгоритм билета или алгоритм поликлиники (глава 3). Если вместо этого использовать протокол "проверить-установить", появится возможность "зависания" процессов. Однако это маловероятно, поскольку критические секции в ядре являются очень короткими.

В листинге 6.3 показана схема многопроцессорного ядра, включающая все перечисленные выше предположения и решения. Переменная `i` используется как индекс процессора, на котором выполняются процедуры, а операции "заблокировать" и "освободить" реализуют протоколы входа в критические секции и выхода из них. В этом решении не учтены возможные особые ситуации, а также не включен код для обработки прерываний ввода-вывода, управления памятью и т.д.

Листинг 6.3. Схема ядра для мультипроцессора с разделяемой памятью

```
processType processDescriptor[maxProcs];
int executing[maxProcs];  # по одному элементу для процессора
```

объявления списков свободных, готовых к работе и ожидающих процессов и их блокировок;

```
SVC_Handler: {
  # вход с прерываниями, запрещенными на процессоре i
  сохранить состояние процесса, указываемого executing[i];
  определить, какой примитив был вызван, и запустить его;
}

Timer_Handler: {
# вход с прерываниями, запрещенными на процессоре i
  заблокировать список готовых к работе;
  вставить executing[i] в конец списка готовых;
  разблокировать список готовых;
  executing[i] = 0;
  dispatcher();
}

procedure fork(начальное сосотояние процесса) {
  заблокировать список свободных; удалить дескриптор;
  разблокировать список свободных;
  инициализировать дескриптор;
  заблокировать список готовых; вставить дескриптор в конец списка готовых;
  разблокировать список готовых;
  dispatcher();
}

procedure quit() {
  заблокировать список свободных; вставить executing[i] в конец списка свободных;
  разблокировать список свободных;
  записать, что процесс executing[i] завершен; executing[i] = 0;
  if (родительский процесс ожидает завершения сыновнего) {
    заблокировать список ожидающих процессов;
    удалить родительский процесс из списка ожидающих;
    разблокировать список ожидающих;
    заблокировать список готовых;
    поместить родительский процесс в список готовых;
    разблокировать список готовых;
  }
  dispatcher();
}

procedure join(имя сыновнего процесса) {
  if (сыновний процесс уже завершен)
    return;
  заблокировать список ожидающих;
  вставить executing[i] в список ожидающих;
  разблокировать список ожидающих;
  dispatcher();
}

procedure dispatcher() {
  if (executing[i] == 0) {
    заблокировать список готовых;
    if (список готовых не пуст) {
      удалить дескриптор из списка готовых;
```

```
    установить на него переменную executing[i];
  }
  else # список готовых пуст
    установить переменную executing[i] на процесс Idle;
    разблокировать список готовых;
  }
  if (executing[i] не указывает на процесс Idle)
    запустить таймер на процессоре i;
  загрузить состояние процесса, указываемого executing[i];
          # с разрешенными прерываниями
}
```

В многопроцессорном ядре в листинге 6.3 использован один список готовых к работе процессов с очередностью обработки FIFO. Если процессы имеют разные приоритеты, то список готовых к работе процессов должен обслуживаться с учетом приоритетов. Однако тогда процессор будет затрачивать больше времени на работу со списком готовых, по крайней мере, на вставку элементов в список, поскольку новый процесс нужно вставить в позицию очереди, соответствующую его приоритету. Таким образом, список готовых к работе процессов может стать "узким местом" программы. Если число уровней приоритетов фиксировано, то эффективное решение — использовать по одной очереди на каждый уровень приоритета и по одной блокировке на каждую очередь. При таком представлении вставка процесса в список готовых к работе требует только вставки в конец соответствующей очереди. Но если число уровней приоритета изменяется динамически, то чаще всего используют единый список готовых к работе процессов.

В ядре (см. листинг 6.3) также предполагается, что процесс может выполняться на любом процессоре, поэтому диспетчер всегда выбирает первый готовый к выполнению процесс. В некоторых мультипроцессорах такие процессы, как драйверы устройств или файловые серверы, могут работать только на специальном процессоре, к которому присоединены периферийные устройства. В этой ситуации для такого процессора создается отдельный список готовых к работе процессов и, возможно, собственный диспетчер. (Ситуация значительно усложняется, если на специализированном процессоре могут выполняться и обычные процессы, поскольку их тоже нужно планировать.)

Даже если процессу подходит любой процессор, может быть очень неэффективно планировать его для случайного процессора. На машине с неоднородным доступом к памяти, например, процессоры могут получать доступ к локальной памяти значительно быстрее, чем к удаленной. Следовательно, процесс в идеальном случае должен выполняться на том же процессоре, в локальной памяти которого находится его код. Это предполагает наличие отдельного списка готовых к работе процессов для каждого из процессоров и назначение процессов на процессоры в зависимости от того, где хранится их код. Но тогда возникает вопрос *балансировки нагрузки*, т.е. процессоры должны нагружаться работой примерно одинаково. Просто назначать всем процессорам поровну процессов неэффективно; обычно различные процессы создают неодинаковую нагрузку, которая динамически изменяется.

Независимо от того, однородно или нет время доступа к памяти, процессоры обычно имеют кэш-память и буферы трансляции адресов виртуальной памяти. В этой ситуации процесс должен планироваться на тот процессор, на котором он выполнялся последний раз (предполагается, что часть состояния процесса находится в кэш-памяти и буферах трансляции процессора). Кроме того, если два процесса разделяют данные, находящиеся в кэш-памяти, гораздо эффективнее выполнять эти процессы по очереди на одном процессоре, чем на разных. Это называется *совместным планированием*. Здесь также предполагается наличие отдельного списка готовых к работе процессов у каждого процессора, что в свою очередь приводит к необходимости балансировки нагрузки. Дополнительная информация по этим вопросам дана в ссылках и упражнениях в конце главы.

6.3. Реализация семафоров в ядре

Поскольку операции с семафорами являются частным случаем операторов `await`, их можно реализовать с помощью активного ожидания и методов из главы 3. Но единственная причина, по которой это может понадобиться, — потребность в написании параллельных программ с помощью семафоров, а не низкоуровневых циклических блокировок и флагов. Поэтому просто покажем, как добавить семафоры к ядру, описанному в предыдущих разделах. Для этого необходимо дополнить ядро дескрипторами семафоров и тремя дополнительными примитивами: `createSem`, `Psem` и `Vsem`. (Библиотеки наподобие Pthreads реализованы аналогично, но выполняются на основе операционной системы, поэтому используются с помощью обычных вызовов процедур и включают программные обработчики сигналов, а не аппаратные обработчики прерываний.)

Дескриптор семафора содержит значение одного семафора; его инициализация происходит при вызове процедуры `createSem`. Примитивы `Psem` и `Vsem` реализуют операции `P` и `V`. Предполагается, что все семафоры являются обычными. Сначала покажем, как добавить семафоры к однопроцессорному ядру из раздела 6.1, а затем — как изменить полученное ядро, чтобы оно поддерживало мультипроцессоры, как в разделе 6.2.

Напомним, что в однопроцессорном ядре в любой момент времени выполняется только один процесс, а все остальные либо готовы к выполнению, либо ожидают завершения работы своих сыновних процессов. Как и раньше, индекс дескриптора выполняемого процесса хранится в переменной `executing`, а дескрипторы всех готовых к работе процессов хранятся в списке готовых к работе.

После добавления к ядру семафоров у процесса появляется еще одно возможное состояние: заблокированный на семафоре. Процесс переходит в это состояние, когда ждет завершения операции `P`. Чтобы отслеживать заблокированные процессы, каждый дескриптор семафора содержит связанный список дескрипторов процессов, заблокированных на этом семафоре. На отдельном процессоре выполняется только один процесс, который не входит ни в один из списков, дескрипторы всех остальных процессов находятся в списке либо готовых к работе, либо ожидающих, либо заблокированных на семафоре процессов.

Для каждого объявления семафора в параллельной программе вырабатывается один вызов примитива `createSem`; в качестве его аргумента передается начальное значение семафора. Примитив `createSem` находит пустой дескриптор семафора, инициализирует значение семафора и список заблокированных процессов и возвращает "имя" дескриптора. Обычно этим именем является адрес дескриптора или его индекс в таблице адресов.

Созданный семафор используется с помощью вызовов примитивов `Psem` и `Vsem`, которые являются процедурами ядра, реализующими операции `P` и `V`. Обе процедуры имеют один аргумент — имя дескриптора семафора. Примитив `Psem` проверяет значение в дескрипторе. Если значение положительно, оно уменьшается на единицу, иначе дескриптор выполняемого процесса вставляется в список блокировки семафора. Аналогично процедура `Vsem` проверяет список блокировки дескриптора семафора. Если он пуст, значение семафора увеличивается, иначе из списка блокировки дескриптора семафора удаляется один дескриптор процесса и вставляется в список готовых к работе. Обычно списки заблокированных процессов реализуются в виде очереди с последовательностью обработки FIFO, поскольку тогда гарантируется справедливость семафорных операций.

В листинге 6.4 даны схемы перечисленных примитивов, которые нужно добавить к однопроцессорному ядру (см. листинг 6.1). Здесь также в конце каждого примитива вызывается процедура `dispatcher`; она работает так же, как и раньше.

Для простоты реализации семафорных примитивов в листинге 6.4 дескрипторы семафоров повторно не используются. Если семафоры создаются один раз, этого достаточно. Однако обычно дескрипторы семафоров, как и дескрипторы процессов, приходится использовать повторно. Для того чтобы ядро поддерживало повторное использование дескрипторов семафоров, можно добавить примитив `destroySem`; он должен вызываться процессом, когда семафор больше не нужен. Другой подход — записывать в дескрипторы процессов имена всех соз-

данных ими семафоров и при вызове процедуры quit уничтожать эти семафоры. При обоих подходах необходимо, чтобы после уничтожения семафор не использовался. Чтобы определить ошибочное использование семафоров, нужно каждому семафору присваивать уникальное имя и проверять его при вызовах процедур Psem и Vsem. Это можно реализовать, используя в качестве имени дескриптора комбинацию индекса дескриптора и уникального порядкового номера.

Листинг 6.4. Примитивы семафора для однопроцессорного ядра

```
procedure createSem(начальное значение, int *name) {
  получить пустой дескриптор семафора;
  инициализировать дескриптор;
  присвоить переменной name имя (индекс) дескриптора;
  dispatcher();
 }

procedure Psem(name) {
  найти дескриптор семафора name;
  if (value > 0)
     value = value - 1;
  else {
    вставить дескриптор процесса executing в конец списка блокировки;
    executing = 0; # показать, что executing заблокирован
  }
  dispatcher();
}

procedure Vsem(name) {
  найти дескриптор семафора name;
  if (список блокировки пуст)
     value = value + 1;
  else {
    удалить дескриптор процесса из списка блокировки;
    вставить дескриптор в конец списка готовых к работе;
  }
  dispatcher();
}
```

Однопроцессорную реализацию примитивов семафора (см. листинг 6.4) можно расширить для мультипроцессора так же, как описано в разделе 6.2 и показано в листинге 6.3. Здесь также необходимо блокировать разделяемые структуры данных, поэтому для каждого дескриптора семафора используется отдельная блокировка. Дескриптор семафора блокируется в процедурах Psem и Vsem непосредственно перед доступом; блокировка снимается, как только исчезает потребность в дескрипторе. Блокировки устанавливаются и снимаются с помощью активного ожидания (см. решения задачи критической секции).

Вопросы, которые обсуждались в конце раздела 6.2, возникают и при реализации семафоров в многопроцессорном ядре. Например, процесс может требовать выполнения на определенном процессоре или на том же, на котором он выполнялся последний раз; может понадобиться совместное планирование процессов на одном процессоре. Чтобы выполнить эти требования или избежать конфликтов из-за разделяемого списка готовых к выполнению процессов, процессоры должны использовать отдельные списки готовых к работе процессов. В этой ситуации процесс, запускаясь примитивом Vsem, помещается в соответствующий список готовых к работе. Для этого примитиву Vsem нужно либо блокировать удаленный список готовых к работе, либо информировать другой процессор и позволить ему поместить разблокированный процесс в его список готовых к работе. При первом подходе нужна удаленная блокировка, при втором — использование механизма типа прерываний процессора для обмена сообщениями между процессорами.

6.4. Реализация мониторов в ядре

Мониторы тоже легко реализуются в ядре; в данном разделе показано, как это сделать. Их можно также моделировать с помощью семафоров; этому посвящен следующий раздел. Блокировки и условные переменные в таких библиотеках, как Pthreads, и таких языках программирования, как Java, реализованы аналогично описанному здесь ядру.

Используем семантику мониторов, определенную в главе 5. В частности, процедуры выполняются со взаимным исключением, а условная синхронизация использует дисциплину "сигнализировать и продолжить". Постоянные переменные мониторов хранятся в области памяти, доступной всем процессам, вызывающим процедуры монитора. Код, реализующий процедуры, может храниться в разделяемой памяти или копироваться в локальную память каждого процессора, который выполняет процессы, использующие монитор. Наконец, постоянные переменные инициализируются перед вызовом процедур. Для этого, например, можно выделять и инициализировать постоянные переменные перед созданием процессов, обращающихся к ним. (Или можно выполнять код инициализации при первом вызове процедуры монитора. Но этот способ менее эффективен, поскольку при каждом вызове придется проверять, первый ли он.)

Для реализации мониторов нужно добавить к ядру примитивы входа в монитор, выхода из него и операций с условными переменными. Нужны также примитивы для создания дескрипторов каждого монитора и каждой условной переменной (если они не создаются при инициализации ядра); эти примитивы здесь не показаны, поскольку аналогичны примитиву `createSem` в листинге 6.4.

Каждый дескриптор монитора `mName` содержит блокировку `mLock` и входную очередь дескрипторов процессов, ожидающих входа (или повторного входа) в монитор. Блокировка используется, чтобы обеспечить взаимное исключение. Если она установлена, в мониторе выполняется только один процесс, иначе — ни одного.

Дескриптор условной переменной содержит начало очереди дескрипторов процессов, ожидающих на этой переменной. Таким образом, каждый дескриптор процесса, за исключением, возможно, выполняющихся процессов, связан либо со списком готовых к работе, либо с очередью входа в монитор, либо с очередью условной переменной. Дескрипторы условных переменных обычно хранятся вместе с дескриптором монитора, в котором объявлена условная переменная, чтобы избежать избыточной фрагментации памяти ядра и иметь возможность идентифицировать условную переменную просто в виде смещения от начала дескриптора соответствующего монитора.

Примитив входа в монитор `enter(mName)` находит дескриптор монитора `mName`, затем либо устанавливает блокировку монитора и разрешает выполняемому процессу продолжать работу, либо блокирует процесс в очереди входа в монитор. Чтобы обеспечить быстрый поиск дескриптора, в качестве идентификатора монитора `mName` во время выполнения программы обычно используют адрес его дескриптора. Примитив выхода из монитора `exit(mName)` либо перемещает один процесс из очереди входа в список готовых к работе, либо снимает блокировку монитора.

Оператор `wait(cv)` реализован с помощью вызова примитива ядра `wait(mName,cName)`, а оператор `signal(cv)` — примитива ядра `signal(mName,cName)`. В обоих примитивах `mName` — это "имя" монитора (его индекс или адрес дескриптора), в котором вызывается примитив, а `cName` — индекс или адрес дескриптора соответствующей условной переменной. Оператор `wait` приостанавливает выполнение процесса на указанной условной переменной, затем либо запускает процесс из очереди входа в монитор, либо снимает блокировку монитора. Выполнение процедуры `signal` заключается в проверке очереди условной переменной. Если очередь пуста, то выполняется просто выход из примитива, иначе дескриптор из начала очереди условной переменной перемещается в конец очереди входа в монитор.

В листинге 6.5 приведены схемы этих примитивов. Как и в предыдущих ядрах, вход в примитивы происходит в результате вызова супервизора, переменная `executing` указывает на дескриптор выполняемого в данный момент процесса, а, когда он заблокирован, ее значение равно `0`. Поскольку процесс, вызвавший примитив `wait`, выходит из монитора, прими-

тив wait просто вызывает примитив exit после блокировки выполняемого процесса. Своим последним действием примитивы enter, exit и signal вызывают диспетчер.

Листинг 6.5. Примитивы ядра для монитора

```
procedure enter(int mName) {
  найти дескриптор монитора mName;
  if (mLock == 1) {
    вставить дескриптор, указываемый executing, в конец очереди входа;
    executing = 0;
  }
  else
    mLock = 1; # получить исключительный доступ к mName
  dispatcher();
}

procedure exit(int mName) {
  найти дескриптор монитора mName;
  if (очередь входа не пуста)
    переместить процесс из начала очереди входа в конец списка готовых к работе;
  else
    mLock = 0;    # снять блокировку
  dispatcher();
 }

procedure wait(int mName; int cName) {
  найти дескриптор условной переменной cName;
  вставить дескриптор, указываемый executing, в конец очереди ожидания на cName;
  executing = 0;
  exit(mName);
}

procedure signal(int mName; int cName) {
  найти дескриптор монитора mName;
  найти дескриптор условной переменной cName;
  if (очередь ожидания не пуста)
    переместить процесс из начала очереди ожидания в конец очереди входа;
  dispatcher();
}
```

Несложно реализовать остальные операции с условными переменными. Например, для реализации empty(cv) достаточно проверить, есть ли элементы в очереди ожидания на cv. В действительности, если очередь задержки доступна процессам напрямую, для реализации операции empty не обязательно использовать примитив ядра. Причина в том, что выполняемый процесс уже заблокировал монитор, поэтому другие процессы не могут изменить содержание очереди условной переменной. Реализация операции empty без блокировки позволяет избежать затрат на вызов супервизора и возврат из него.

Операцию signal также можно реализовать более эффективно, чем показано в листинге 6.5. Например, процедура signal может *всегда* перемещать дескриптор из начала соответствующей очереди задержки в конец соответствующей очереди входа. Затем процедура signal в программе должна транслироваться в код, который проверяет очередь задержки и вызывает процедуру ядра mSignal, только если очередь пуста. После таких изменений отсутствуют затраты на вход в ядро и выход из него, когда операция signal ни на что не влияет. Независимо от реализации signal операция signal_all реализуется примитивом ядра, который перемещает все дескрипторы из указанной очереди задержки в конец списка готовых к выполнению.

Приоритетный оператор `wait` реализуется аналогично неприоритетному. Отличие состоит лишь в том, что дескриптор выполняемого процесса должен быть вставлен в соответствующую позицию очереди задержки. Чтобы сохранять упорядоченность очереди, необходимо знать ранги ожидающих процессов. Логично хранить ранг процесса вместе с его дескриптором, что делает реализацию функции `minrank` тривиальной. Фактически `minrank`, как и `empty`, можно реализовать без входа в ядро, поскольку минимальный ранг можно прочитать непосредственно из выполняемого процесса.

Это ядро можно расширить для мультипроцессора, используя методику, описанную в разделе 6.2. Здесь также основным будет требование защиты структур данных ядра от одновременного доступа из процессов, выполняемых на разных процессорах. Необходимо также обеспечить отсутствие конфликтов в памяти, использование кэш-памяти, совместное планирование процессов и балансировку нагрузки процессоров.

На одном процессоре мониторы иногда можно реализовать значительно эффективнее, не используя ядра. Если вложенных вызовов мониторов нет или все вложенные вызовы являются открытыми, все процедуры мониторов коротки и гарантированно завершаемы, то взаимное исключение стоит реализовать с помощью запрета прерываний. Делается это следующим образом. На входе в монитор выполняемый процесс запрещает все прерывания. Выходя из процедуры монитора, процесс разрешает прерывания. Если процесс должен ожидать внутри монитора, он блокируется в очереди условной переменной; при этом фиксируется, что процесс выполняется с запрещенными прерываниями. (Обычно флаг запрета прерываний находится в регистре состояния процессора, который сохраняется при блокировке процесса.) Запускаясь в результате операции `signal`, ожидающий процесс, перемещается из очереди условной переменной в список готовых к выполнению, а процесс, выработавший сигнал, продолжает работу. Когда готовый к работе процесс ставится диспетчером на выполнение, он продолжает работу с запрещенными или разрешенными прерываниями, в зависимости от состояния процесса в момент блокировки. (Вновь созданные процессы начинают выполнение с разрешенными прерываниями.)

При такой реализации уже не нужны примитивы ядра (они превращаются или во встроенный код, или в стандартные подпрограммы) и дескрипторы мониторов. Поскольку во время работы процесса в мониторе прерывания запрещены, нельзя заставить процесс освободить процессор. Таким образом, процесс получает исключительный доступ к монитору до перехода в состояние ожидания или выхода из процедуры. При условии, что процедуры монитора завершаемы, в конце концов процесс переходит в состояние ожидания или выходит из процедуры. Если процесс ожидает, то при его запуске и продолжении работы прерывания вновь запрещаются. Следовательно, процесс снова получает исключительное управление монитором, в котором он ожидал. Однако вложенные вызовы процедур монитора недопустимы. Если бы они были разрешены, то в мониторе, из которого был сделан вложенный вызов, мог начать выполнение еще один процесс, в то время как в другом мониторе есть ожидающий процесс.

По существу, описанным способом мониторы реализованы в операционной системе UNIX. При входе в процедуру монитора запрещаются прерывания только от тех устройств, которые могут привести к вызову процедуры этого же монитора до того, как прерываемый процесс перейдет в состояние ожидания или выйдет из монитора. В общем случае, однако, необходима реализация мониторов в ядре, поскольку не все программы удовлетворяют условиям этой специфической реализации. Например, только в такой "надежной" программе, как операционная система, можно надеяться на то, что все процедуры монитора завершаемы. Кроме того, описанная реализация может работать только на одном процессоре. На мультипроцессоре будут нужны блокировки в той или иной форме, чтобы обеспечить взаимное исключение процессов, выполняемых на разных процессорах.

6.5. Реализация мониторов с помощью семафоров

В предыдущем разделе было описано, как реализовать мониторы с помощью ядра. Здесь показано, как реализовать их, используя семафоры. Это может понадобиться по двум причи-

нам: 1) библиотека программирования поддерживает семафоры и не поддерживает мониторы, или 2) язык программирования обеспечивает семафоры и не обеспечивает мониторы. Так или иначе, описываемое решение иллюстрирует еще одно применение семафоров.

Вновь используем семантику мониторов, определенную в разделе 5.1. Следовательно, нужно реализовать взаимное исключение процедур монитора и условную синхронизацию между ними. В частности, нужно разработать: 1) код входа, который выполняется после вызова процедуры монитора из процесса, но перед началом выполнения тела процедуры; 2) код выхода, выполняемый перед выходом процесса из тела процедуры; 3) код, реализующий операции `wait`, `signal` и другие операции с условными переменными. Для простоты предположим, что есть только одна условная переменная; разработанный ниже код несложно обобщить на случай нескольких условных переменных, используя массивы очередей задержки и счетчиков.

Для реализации исключения в мониторах используем по одному входному семафору на монитор. Пусть `e` — семафор, связанный с монитором `M`. Поскольку семафор `e` должен использоваться для получения взаимного исключения, он инициализируется значением `1`, и его значение всегда должно быть или `0` или `1`. Цель протокола входа каждой процедуры монитора `M` — обеспечить исключительный доступ к `M`. Как всегда, для этого используется операция `P(e)`. Аналогично протокол выхода каждой процедуры снимает блокировку монитора, поэтому реализуется операцией `V(e)`.

Выполнение операции `wait(cv)` снимает блокировку монитора и задерживает выполняемый процесс на условной переменной `cv`. Процесс продолжает работу в мониторе после получения сигнала и нового исключительного доступа к монитору. Поскольку условная переменная, по сути, является очередью приостановленных процессов, можно считать, что есть тип данных `queue`, реализующий очередь процессов с порядком обработки FIFO, и можно использовать массив `delayQ` этого типа. Для учета числа приостановленных процессов используем целочисленный счетчик `nc`. В начальном состоянии массив `delayQ` пуст, а значение переменной `nc` равно нулю, поскольку ожидающих процессов нет.

Когда процесс выполняет операцию `wait(cv)`, он увеличивает счетчик `cv` и после этого добавляет свой дескриптор в массив `delayQ`. Затем процесс снимает блокировку монитора, выполняя операцию `V(e)`, и сам блокируется на *скрытом* семафоре, на котором ожидать может только данный процесс. После запуска процесса он ждет повторного вхождения в монитор, выполняя операцию `P(e)`.

Выполняя процедуру `signal(cv)`, процесс сначала проверяет значение `nc`, чтобы узнать, есть ли ожидающие процессы. Если их нет, процедура `signal(cv)` не дает никаких результатов. Если ожидающие процессы есть, процесс-сигнализатор уменьшает `nc`, берет из начала массива `delayQ` процесс, ожидавший дольше всех, и вырабатывает сигнал для его скрытого семафора.[11]

Код, реализующий синхронизацию мониторов с помощью семафоров, приведен в листинге 6.6. Взаимное исключение обеспечено, поскольку семафор `e` имеет начальное значение `1`, а на любой ветви выполнения программа попеременно выполняет операции `P(e)` и `V(e)`. Значение `nc` положительно, если задержан или должен быть задержан хотя бы один процесс.

При описанном представлении условных переменных легко реализуется примитив `empty`. Он возвращает значение "истина", если значение `nc` равно нулю, и "ложь", если оно положительно. Примитив `signal_all` тоже реализуется просто: нужно удалить из массива `delayQ` все ожидающие процессы, выработать сигналы для их скрытых семафоров и присвоить переменной `nc` значение `0`. Чтобы реализовать приоритетное ожидание, достаточно сделать `delayQ` приори-

[11] Можно подумать, что для очереди задержки достаточно было бы использовать семафор, поскольку он является, по существу, очередью заблокированных процессов. В результате отпала бы необходимость неявной очереди задержки и массива скрытых семафоров. Но такое решение создает еще одну проблему: хотя операции `P` и `V` неделимы, их последовательность неделимой не является. Например, процесс может быть прерван в процедуре `wait` после выполнения операции `V(e)` и перед блокировкой. Этому вопросу посвящено упражнение 6.12.

тетной очередью, упорядоченной по рангам элементов. С таким дополнением операция minrank реализуется тривиально: просто возвращается ранг процесса в начале очереди delayQ.

Листинг 6.6. Реализация мониторов с помощью семафоров

```
разделяемые переменные: sem e = 1;        # по одной копии на монитор
                        int nc = 0;       # по одной копии на условие
                        queue delayQ;     # переменная cv
                        sem private[N]; # по одной копии на процесс

вход в монитор: P(e);

wait(cv):     nc = nc+1; вставить myid в delayQ; V(e);
              P(private[myid]); P(e);

signal(cv):   if (nc > 0) {
                nc = nc-1;
                удалить otherid из delayQ;
                V(private[otherid]);
              }

выход из монитора: V(e);
```

Историческая справка

Примитивы fork, join и quit впервые были представлены Деннисом и Ван Хорном [Dennis and Van Horn, 1966]. Различные варианты этих примитивов есть в большинстве операционных систем. Например, в операционной системе UNIX [Ritchie and Thompson, 1974] обеспечены соответствующие системные вызовы fork, wait и exit. Похожие примитивы были включены в такие языки программирования, как PL/I, Mesa и Modula-3.

Реализация однопроцессорного ядра особенно хорошо описана в книгах [Bic and Shaw, 1988] и [Holt, 1983]. В них рассмотрены и другие функции, которые должна обеспечивать операционная система (файловая система и управление памятью), и их взаимосвязь с ядром. В [Thompson, 1978] описана реализация ядра системы UNIX, а в [Holt, 1983] — UNIX-совместимая система Tunis.

К сожалению, в работах по операционным системам недостаточно подробно рассмотрена тема многопроцессорного ядра. Однако прекрасный отчет о ранних мультипроцессорах, разработанных в университете Карнеги-Меллон (Carnegie-Mellon University), представлен в обзорной статье [Jones and Schwartz, 1980]. Из этой же работы взят принцип блокировки мультипроцессора (6.1). Несколько многопроцессорных операционных систем описаны в работах [Hwang, 1993] и [Almasi and Gottlieb, 1994]. В [Tucker and Gupta, 1989] рассмотрены вопросы управления процессами и планирования для мультипроцессоров с разделяемой памятью и равномерным распределением времени доступа к памяти, в [Scott et al., 1990] — вопросы ядра для мультипроцессоров с неравномерным временем доступа к памяти, включая использование нескольких списков готовых к работе процессов.

Хорошими источниками информации по последним работам в области языков программирования и программного обеспечения, связанной с мультипроцессорами, являются доклады следующих трех конференций: "Архитектурная поддержка языков программирования и операционных систем" (Architectural Support for Programming Languages and Operating Systems, ASPLOS), "Симпозиум по принципам операционных систем" (Symposium on Operating Systems Principles — SOSP) и "Принципы и практика параллельного программирования" (Principles and Practice of Parallel Programming — PPoPP).

Реализация мониторов в ядре описана в нескольких работах. В [Wirth, 1977] представлено ядро Modula, в [Holt et al., 1978] — однопроцессорное и многопроцессорное ядра для CSP/k, а в [Holt, 1983] — ядра для языка Concurrent Euclid. В книге [Joseph et al., 1984] описана разработка и реализация полной операционной системы для мультипроцессорной системы с разделяемой памятью, в которой мониторы использовались внутри операционной системы и были реализованы в ядре. В статье [Thompson, 1978] и книге [Holt, 1983] представлена реализация системы UNIX.

Взаимное исключение в ядре операционной системы UNIX реализовано с помощью переключения контекста, происходящего только когда пользовательский процесс блокируется или выходит из ядра, а также с помощью запрета внешних прерываний в критических точках. Эквивалент условной переменной в системе UNIX называется *событием* — произвольное целое число, которое обычно является адресом дескриптора процесса или дескриптором файла. В ядре процесс блокируется, вызывая процедуру `sleep(e)`, и продолжает работу после вызова процедуры `wakeup(e)` из другого процесса. Примитив `wakeup` имеет семантику "сигнализировать и продолжить". Он используется и для оповещения, т.е. процедура `wakeup(e)` запускает все процессы, заблокированные на событии `e`. В системе UNIX нет эквивалента примитиву `signal` для запуска только одного процесса. Таким образом, если несколько процессов ждут выполнения одного события, то каждый из них должен проверить истинность ожидаемого условия и, если оно не выполняется, возвратиться в состояние задержки.

В классической работе по мониторам Хоара [Hoare, 1974] описано, как их реализовать с помощью семафоров, но при порядке "сигнализировать и срочно ждать", а не "сигнализировать и продолжить" или "сигнализировать и ожидать".

Литература

Almasi, G. S., and A. Gottlieb. 1994. *Highly Parallel Computing*, 2nd ed. Menlo Park, CA: Benjamin/Cummings.

Bic, L., and A. C. Shaw. 1988. *The Logical Design of Operating Systems*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall.

Dennis, J. B., and E. C. Van Horn. 1966. Programming semantics for multi-programmed computations. *Comm. ACM* 9, 3 (March): 143–155.

Hoare, C. A. R. 1974. Monitors: An operating system structuring concept. *Comm. ACM* 17, 10 (October): 549–557.

Holt, R. C. 1983. *Concurrent Euclid, The UNIX System, and Tunis*. Reading, MA: Addison-Wesley.

Holt, R. C., G. S. Graham, E. D. Lazowska, and M. A. Scott. 1978. *Structured Concurrent Programming with Operating System Applications*. Reading, MA: Addison-Wesley.

Hwang, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. New York: McGraw-Hill.

Jones, A. K., and P. Schwarz. 1980. Experience using multiprocessor systems — a status report. *ACM Computing Surveys* 12, 2 (June): 121–165.

Joseph, M., V. R. Prasad, and N. Natarajan. 1984. *A Multiprocessor Operating System*. New York: Prentice-Hall International.

Ritchie, D. M., and K. Thompson. 1974. The UNIX timesharing system. *Comm. ACM* 17, 7 (July): 365–375.

Scott, M. L., T. J. LeBlanc, and B. D. Marsh. 1990. Multi-model parallel programming in Psyche. *Proc. Second ACM Symp. on Principles & Practice of Parallel Prog.*, March, pp. 70–78.

Thompson, K. 1978. UNIX implementation. *The Bell System Technical Journal* 57, 6, part 2 (July-August): 1931–1946.

Tucker, A., and A. Gupta. 1989. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *Proc. Twelfth ACM Symp. on Operating Systems Principles*, December, pp. 159–166.

Wirth, N. 1977. Design and implementation of Modula. *Software — Practice and Experience* 7, 67–84.

Упражнения

6.1. В многопроцессорном ядре, описанном в разделе 6.2, процессор выполняет процесс `Idle` (см. листинг 6.2), когда определяет, что список готовых к работе процессов пуст. На некоторых машинах есть бит в регистре (слове) состояния процессора. Его установка означает, что процессор должен бездействовать до возникновения прерывания (бит бездействия). В таких машинах есть и межпроцессорные прерывания, т.е. один процессор может быть прерван другим, в результате чего он входит в обработчик прерывания ядра на центральном процессоре.

Измените многопроцессорное ядро в листинге 6.3 так, чтобы процессор сам устанавливал бит бездействия, когда список готовых к работе процессов пуст. Таким образом, для запуска процесса на одном процессоре потребуется выполнить команду на другом.

6.2. Предположим, что планирование процессов ведется одним главным процессором, выполняющим только процесс-диспетчер. Другие процессоры выполняют обычные процессы и процедуры ядра.

Разработайте процесс-диспетчер и соответственно измените многопроцессорное ядро в листинге 6.3. Определите все необходимые структуры данных. Помните, что бездействующий процессор не должен блокироваться внутри ядра, поскольку тогда в ядро не смогут войти остальные процессы.

6.3. Предположим, что все процессоры мультипроцессорной системы имеют собственные списки готовых к работе процессов и выполняют процессы только из них. Как было отмечено в тексте, возникает проблема балансировки нагрузки процессоров, поскольку новый процесс приходится назначать какому-нибудь из процессоров.

Существует много схем балансировки нагрузки процессоров, например, можно назначать новый процесс случайному процессору, "соседу" или поддерживать приблизительно одинаковую длину списков готовых к работе процессов. Выберите одну из схем, обоснуйте свой выбор и измените многопроцессорное ядро в листинге 6.3 так, чтобы в нем использовались несколько списков готовых к работе процессов и выбранная схема балансировки нагрузки процессоров. Покажите, как работать с бездействующими процессорами. Постарайтесь уменьшить затраты на дополнительный код и конфликты, связанные с блокировками. (Например, если каждый процессор, и только он, может обращаться к своему списку готовых к работе процессов, блокировка этих списков не нужна.)

6.4. Измените многопроцессорное ядро в листинге 6.3 так, чтобы процесс всегда возобновлялся на том же процессоре, на котором он выполнялся последний раз. Но в вашем решении не должно быть "зависших" процессов, т.е. каждый процесс периодически должен получать возможность выполнения. Не забудьте о простаивающих процессорах. Определите необходимые типы данных. Обоснуйте свое решение.

6.5. Добавьте примитив `multifork(initialState[*])` к многопроцессорному ядру (см. листинг 6.3). Аргумент является массивом начальных состояний. Выполнение примитива `multifork` создает по одному процессу для каждого аргумента и определяет, что

вновь созданные процессы совместно планируются для одного процессора, — например, когда они разделяют кэшируемые данные.

Сделайте необходимые для совместного планирования изменения и в других частях ядра. В вашем решении не должно быть "зависших" процессов, т.е. каждый процесс периодически должен получать возможность выполнения. Не забудьте о простаивающих процессорах.

6.6. Примитивы семафора в листинге 6.4 и монитора в листинге 6.5 были частью однопроцессорного ядра, показанного в листинге 6.1. Измените эти примитивы, чтобы их можно было добавить к многопроцессорному ядру (см. листинг 6.3). Внесите необходимые изменения.

6.7. Пусть ввод и вывод терминала обслуживаются двумя процедурами:

```
char getc()
putc(char ch)
```

Процесс приложения вызывает функцию `getc`, чтобы получить следующий символ ввода, а функцию `putc` — чтобы послать символ на дисплей. Предположим, что для ввода и вывода используются буферы, т.е. вплоть до `n` входных символов могут храниться в ожидании их получения функцией `getc` и вплоть до `n` выходных символов — в ожидании вывода на дисплей:

а) разработайте реализацию функций `getc` и `putc`, предполагая, что они выполняются вне ядра. Для синхронизации эти функции должны использовать семафоры. Определите все необходимые дополнительные процессы и покажите, какие операции должны выполняться ядром при возникновении прерывания ввода или вывода. Для инициализации считывания с клавиатуры используйте примитив `startread`, записи на дисплей — примитив `startwrite`;

б) разработайте реализацию функций `getc` и `putc` в виде примитивов ядра. Определите, какие операции должны выполняться ядром при возникновении прерывания ввода или вывода. Для инициализации считывания с клавиатуры используйте примитив `startread`, записи на дисплей — примитив `startwrite`;

в) проанализируйте и сравните эффективность ответов к пунктам *а* и *б*. Рассмотрите такие факторы, как число операторов, которые необходимо выполнить в каждом случае, количество переключений контекста и время, в течение которого ядро остается заблокированным;

г) расширьте ответ к пункту *а*, чтобы обеспечить эхо символов ввода. Для этого каждый введенный с клавиатуры символ должен автоматически записываться на дисплей;

д) расширьте ответ к пункту *б*, чтобы обеспечить эхо символов ввода.

6.8. Предположим, что ввод и вывод на терминал поддерживаются двумя функциями:

```
int getline(string *str)
putline(string *str)
```

Прикладной процесс вызывает функцию `getline` для считывания строк ввода; функция `getline` записывает эту строку в переменную `str` и возвращает число символов в строке. Для вывода на дисплей строки из переменной `str` прикладной процесс вызывает функцию `putline`. Строка содержит не больше MAXLINE символов и завершается символом новой строки, который сам в строку не входит.

Предположим, что для ввода и вывода используются буферы, т.е. вплоть до `n` входных строк могут храниться в ожидании их получения функцией `getline`, и вплоть до `n` выходных строк — в ожидании вывода на дисплей. Предположим также, что входные строки дают эхо на дисплей, т.е. каждая законченная входная строка также посылается и на дисплей. Наконец, входные строки подготавливаются для чтения, т.е. символы

удаления строки и предыдущего символа обрабатываются функцией `getline` и не возвращаются прикладному процессу:

а) разработайте реализацию функций `getline` и `putline` при условии, что они выполняются вие ядра. Для синхронизации они должны нспользовать семафоры. Определите все иеобходимые дополнительные процессы, покажите, какие операции должны выполияться ядром при возникновении прерывания ввода или вывода. Для иинициализации считывания с клавиатуры используйте примитив `startread`, записи на дисплей — примитив `startwrite`;

б) разработайте реализацию функций `getline` и `putline` в виде примитивов ядра. Определите все необходимые дополнительные процессы, покажите, какие операции должны выполняться ядром при возникновении прерывания ввода или вывода. Для инициалнзации считывания с клавиатуры используйте примитив `startread`, записи на дисплей — примитив `startwrite`.

6.9. Некоторые машины обеспечивают инструкции, реализующие операции `P` и `V` с семафорами. Эти инструкции изменяют значение семафора, затем, если процесс должен быть заблокирован или запущен, прерывают ядро. Разработайте реализацию этих машинных инструкций, напишите код ядра для обработки соответствующих прерываний. (*Указание.* Значение семафора может становиться отрицательным, тогда его абсолютное значение показывает число заблокированных процессов.)

6.10. В листинге 6.5 представлены примитивы ядра, реализующие вход в монитор и выход из него, а также процедуры `wait` и `signal` для порядка "сигнализировать и продолжить":

а) измените примитивы для использования с порядком "сигнализировать и ждать";

б) измените примитивы для использования с порядком "сигнализировать и срочно ожидать".

6.11. В листинге 6.6 показано, как использовать семафоры для реализации входа в монитор, выхода из монитора, процедур `wait` и `signal` с порядком "сигнализировать и продолжить":

а) измените реализацию, чтобы использовать порядок "сигнализировать и ждать";

б) измените реализацию, чтобы использовать порядок "сигнализнровать и срочно ожидать".

6.12. В листинге 6.6 показано использование семафоров для реализации входа в монитор, выхода из монитора, процедур `wait` и `signal` для порядка "сигнализировать и продолжить". Можно было бы подумать о более простой реализации с использованием для очереди задержки семафора с следующим образом.

```
разделяемые переменные:  sem e = 1, c = 0; int nc = 0;
вход в монитор:  P(e);
wait(cv):  nc = nc+1; V(e); P(c); P(e);
signal(cv):  if (nc > 0) { nc = nc-1; V(c); }
выход из монитора:  V(e);
```

К сожалению, эта упрощенная реализация ошибочна. Проблема возникает, если процесс прерывается в процедуре `wait(cv)` после `V(e)` и перед `P(c)`. Объясните, что здесь некорректно, и придумайте пример для иллюстрации проблемы. (*Указание.* Если один процесс пропустит сигнал, то другой ошибочно получит два сигнала; это может привести к взаимоблокировке, которой быть не должно.)

Часть 2

Распределенное программирование

Конструкции для синхронизации, которые рассматривались до сих пор, основаны на чтении и записи разделяемых переменных. Следовательно, чаще всего они используются в параллельных программах, выполняемых на машинах, в которых процессоры разделяют память.

В настоящее время стали обычными *архитектуры с распределенной памятью*. Как описано в разделе 1.2, к ним относятся многомашинные системы и сети машин. Кроме того, иногда применяются такие гибридные комбинации архитектуры с разделяемой памятью и сетевой архитектуры, как сеть из рабочих станций и мультипроцессоров. При распределенной архитектуре процессоры имеют собственную локальную память и общаются с помощью сети связи, а не разделяемой памяти. Следовательно, процессы не могут взаимодействовать напрямую, разделяя переменные. Вместо этого они должны обмениваться *сообщениями*.

Чтобы писать программы для архитектуры с распределенной памятью, сначала необходимо определить их интерфейсы с сетью связи. Это могут быть просто операции чтения и записи, аналогичные операциям чтения и записи разделяемых переменных, т.е. в таких программах понадобится синхронизация с активным ожиданием. Лучше определить специальные сетевые операции, которые включают синхронизацию. Они называются *примитивами передачи сообщений*. Передачу сообщений можно рассматривать как обобщение семафоров для перенаправления данных и обеспечения синхронизации. При передаче сообщений процессы разделяют *каналы*. Каждый канал обеспечивает путь взаимодействия между процессами и, следовательно, является абстракцией сети связи, обеспечивающей физический путь между процессорами.

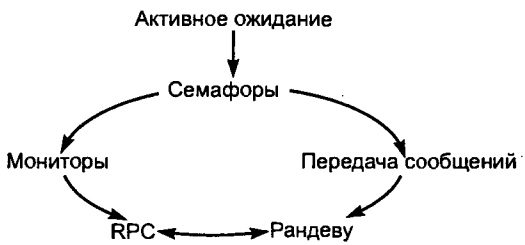
Параллельные программы, которые используют передачу сообщений, называются *распределенными программами*, поскольку их процессы могут выполняться на разных процессорах с распределенной памятью. Распределенная программа может работать и на мультипроцессоре с разделяемой памятью (как любая параллельная программа может выполняться на одном процессоре). В этой ситуации каналы реализуются с помощью разделяемой памяти вместо сети связи.

В распределенных программах каналы обычно являются единственными объектами, которые разделяются процессами. Таким образом, каждая переменная локальна по отношению к одному процессу, который является *владельцем* (caretaker). Это значит, что переменная никогда не становится объектом параллельного доступа, и поэтому нет необходимости в специальном механизме для взаимного исключения. Чтобы взаимодействовать, процессы должны общаться. Итак, основной темой части 2 является синхронизация межпроцессного взаимодействия. Каким образом оно происходит, зависит от схемы взаимодействия процессов: производители и потребители, клиенты и серверы или взаимодействующие равные (см. раздел 1.3).

Предлагалось несколько различных механизмов распределенного программирования. Они отличаются способами использования каналов и синхронизации взаимодействия. Например, каналы могут обеспечивать однонаправленный или двунаправленный поток инфор-

мации, а взаимодействие — быть асинхронным (неблокирующим) или синхронным (блокирующим). В части 2 описаны четыре комбинации: асинхронная передача сообщений, синхронная передача сообщений, удаленный вызов процедур (remote procedure call — RPC) и рандеву. Все четыре механизма эквивалентны, т.е. программа, написанная с использованием одного набора примитивов, может быть переписана с помощью любого из трех других наборов. Однако, как будет видно, с помощью передачи сообщений лучше всего программируются производители и потребители, а также взаимодействующие равные, тогда как RPC и рандеву лучше подходят для программирования клиентов и серверов.

Приведенный ниже рисунок иллюстрирует, как четыре механизма распределенного программирования соотносятся друг с другом и механизмами, использующими разделяемые переменные (см. часть 1). Например, семафоры являются развитием активного ожидания; мониторы сочетают неявное исключение с явной передачей сигналов семафорам; передача сообщений расширяет семафоры данными; протокол RPC и рандеву объединяют процедурный интерфейс мониторов с неявной передачей сообщений.



В главе 7 рассматривается *передача сообщений*, в которой каналы связи обеспечивают однонаправленную посылку от передающего процесса получающему. Каналы являются очередями сообщений типа FIFO. Доступ к каналу производится с помощью обращения к двум примитивам: `send` и `recieve`. Для инициализации взаимодействия один процесс посылает в канал сообщение; другой забирает его оттуда. Отправка сообщения может быть асинхронной (неблокирующей) или синхронной (блокирующей), но получение сообщения всегда является блокирующим, что облегчает программирование и повышает его эффективность. В главе 7 дано определение примитивов асинхронной передачи сообщений и представлены примеры их использования. Также описана дуальность мониторов и передачи сообщений: они эквивалентны и взаимно преобразуемы. В разделе 7.5 объясняется синхронная передача сообщений. В последних четырех разделах приводятся примеры программной нотации CSP, примитивы Linda, библиотека MPI и сетевой модуль языка Java.

В главе 8 рассматриваются *удаленные операции* и два возможных способа их реализации: *удаленный вызов процедур* (RPC) и *рандеву*. Удаленные операции сочетают основные черты мониторов и синхронной передачи сообщений. Как и в мониторах, модуль или процесс экспортирует операции, которые вызываются оператором `call`. Как и при синхронной передаче сообщений, выполнение оператора `call` происходит синхронно — вызвавший его процесс ждет обслуживания вызова и получения результатов. Таким образом, эта операция является двунаправленным каналом связи, от процесса, вызвавшего оператор, к процессу, который обслуживает этот вызов, и обратно. Вызов обслуживается одним из двух способов. Один состоит в создании нового процесса. Он называется *удаленным вызовом процедуры* (*RPC*), поскольку обслуживающий процесс объявлен как процедура и может выполняться не на том процессоре, на котором выполняется вызывающий процесс. Второй способ состоит в инициализации встречи (*рандеву*) с существующим процессом. Рандеву обслуживается с помощью оператора ввода, который ожидает вызов, обрабатывает его и возвращает результаты. Эти механизмы иллюстрируются тремя примерами: модулем удаленного вызова методов языка Java (RPC), рандеву в языке программирования Ada и примитивами языка программирования SR (включая RPC и рандеву).

В главе 9 описано несколько схем взаимодействия процессов в распределенных программах, которые применяются в многочисленных приложениях. Первые три схемы обычно используются для реализации параллельных вычислений на машинах с распределенной памятью: управляющий и рабочие (распределенный портфель задач), пульсация и конвейер. Остальные четыре схемы используются для координации действий распределенного набора программ: зонд-эхо, оповещение (рассылка), передача маркера и децентрализованные серверы.

В главе 10 описывается реализация примитивов передачи сообщений. Вначале показано, как реализуются каналы и передача сообщений, затем описана реализация RPC и рандеву. При написании программ для машины с разделяемой памятью обычно применяются методы, описанные в части 1 этой книги, а для машины с распределенной памятью — передача сообщений, RPC и рандеву. Но можно смешивать и объединять эти методы: разделяемые переменные использовать на машинах с распределенной памятью, а передачу сообщений — с разделяемой памятью. В разделе 10.4 описано, как реализовать *распределенную разделяемую память* (РРП), которая обеспечивает модель программирования с разделяемой памятью на машине с распределенной памятью.

Глава 7

Передача сообщений

Как уже отмечалось, передача сообщений может быть асинхронной или синхронной. Асинхронная передача сообщений используется чаще и является основной темой данной главы. При асинхронной передаче сообщений каналы подобны семафорам, переносящим данные, а примитивы `send` и `receive` аналогичны операциям `V` и `P` над семафорами (соответственно). В действительности, если канал содержит только пустые сообщения (без каких-либо данных), то примитивы `send` и `receive` аналогичны операциям `V` и `P`, причем число "сообщений" в очереди соответствует значению семафора.

В разделе 7.1 определены примитивы для асинхронной передачи сообщений, а в разделе 7.2 показано, как их использовать для программирования фильтров. В разделе 7.3 рассмотрены различные приложения типа "клиент-сервер", включая выделение ресурсов, планирование доступа к диску и файловые серверы. Эти приложения демонстрируют дуальность мониторов и передачи сообщений, иллюстрируют программирование так называемой *непрерывности диалога*, при которой клиент продолжает взаимодействие с сервером (как в Web-приложениях). В разделе 7.4 приводится пример взаимодействующих равных и иллюстрируются три общие схемы взаимодействия: централизованная, симметричная и кольцевая. В разделе 7.5 описана синхронная передача сообщений и ее отличие от асинхронной.

В последних четырех разделах приводятся учебные примеры для трех языков программирования и библиотеки подпрограмм: 1) язык CSP (Communicating Sequential Processes — взаимодействующие последовательные процессы), который представляет синхронную передачу сообщений и так называемое защищенное взаимодействие; 2) язык Linda, обеспечивающий уникальное сочетание разделяемой и ассоциативной памяти (область кортежей), а также шесть примитивов, подобных сообщениям; 3) библиотека MPI, которая широко используется и предоставляет множество примитивов для глобального и межпроцессного взаимодействия; 4) сетевой пакет языка Java, демонстрирующий использование сокетов для программирования простого файлового сервера.

7.1. Асинхронная передача сообщений

Для асинхронной передачи сообщений предлагалось много различных нотаций. Используем одну из них, достаточно представительную и простую.

При асинхронной передаче сообщений канал является очередью сообщений, которые уже отправлены, но еще не получены. Объявление канала имеет вид:

```
chan ch(type1 id1, ..., typen idn);
```

Идентификатор `ch` — это имя канала, $\texttt{type}_i$ и $\texttt{id}_i$ — типы и имена полей данных в передаваемых по каналу сообщениях. Типы необходимы, а имена полей — нет, но они будут использоваться, когда полезно явно указать, что представляет каждое поле. В следующем примере объявлены два канала:

```
chan input(char);
chan disk_access(int cylinder, int block,
                 int count, char* buffer;)
```

Первый канал, input, используется для передачи односимвольных сообщений. Второй, disk_access, содержит сообщения с четырьмя полями; имена полей отражают их назначение. Во многих примерах используются массивы каналов, например:

```
chan result[n](int);
```

Индексы могут иметь значения от 0 до n-1, если только не указан другой промежуток.

Процесс отправляет сообщение каналу ch, выполняя операцию send:

```
send ch(expr1, ..., exprn);
```

Выражения $expr_i$ должны иметь тот же тип, что и соответствующие поля в декларации канала ch. При выполнении операции send вычисляются выражения, затем сообщение с полученными значениями присоединяется к концу очереди, связанной с каналом ch. Поскольку эта очередь не ограничена (по крайней мере теоретически), выполнение операции send никогда не вызывает задержку, поэтому операция send является *неблокирующим* примитивом.

Процесс получает сообщение из канала ch, выполняя операцию:

```
receive ch(var1, ..., varn);
```

Переменные var_i должны иметь тот же тип, что и соответствующие поля в декларации канала ch. При выполнении операции receive принимающий процесс приостанавливается до тех пор, пока в очереди канала не будет хотя бы одного сообщения. Затем из очереди удаляется первое сообщение и значения его полей присваиваются переменным var_i. Таким образом, выполнение операции receive может вызывать задержку, поэтому она является *блокирующим* примитивом, и процесс, принимающий сообщение, не обязан использовать активное ожидание для опроса канала, когда ему нечего делать до прибытия сообщения.

Предполагается, что доступ к содержимому каждого канала является неделимым, а сообщения передаются надежно и без ошибок. Таким образом, каждое переданное в канал сообщение в конце концов будет принято, причем без искажений. Поскольку канал является очередью типа FIFO, сообщения будут приняты в том же порядке, в котором они добавлялись в канал.

Рассмотрим простой пример. В листинге 7.1 представлен процесс, который принимает поток символов из канала input, собирает символы в строки и отправляет строки в канал output. Конец строки задается символом возврата каретки CR, строка содержит не больше, чем MAXLINE символов; для задания конца выходной строки к ней добавляется специальное значение EOL. (CR, MAXLINE и EOL являются символьными константами.)

Листинг 7.1. Процесс-фильтр для составления строк символов

```
chan input(char), output(char [MAXLINE]);

process Char_to_Line {
  char line[MAXLINE]; int i = 0;
  while (true) {
    receive input(line[i]);
    while (line[i] != CR and i < MAXLINE) {
      # line[0:i-1] содержит последние i входных символов
      i = i+1;
      receive input(line[i]);
    }
    line[i] = EOL;
    send output(line);
    i = 0;
  }
}
```

Каналы используются процессами совместно, поэтому объявляются глобальными относительно процессов (как в листинге 7.1). Любой процесс может отправлять данные в любой

канал и принимать их из любого канала. Когда каналы используются таким образом, их иногда называют *почтовыми ящиками*. Однако во многих примерах, которые будут рассматриваться, у канала будет только один получатель, хотя и несколько отправителей. Такой канал часто называется *входным портом*, поскольку он обеспечивает окно (вход) в процесс-получатель. Если у канала только один отправитель и один получатель, то его обычно называют *каналом связи*, поскольку он обеспечивает прямой путь от передающего процесса к принимающему.

Обычно при выполнении примитива `receive` процесс вынужден ждать, но не всегда. Например, если процессу не доступны новые сообщения, он может выполнять другую полезную работу. Или такой процесс, как диспетчер, должен просмотреть всю очередь сообщений, чтобы выбрать наиболее подходящее (по некоторым критериям) для последующей обработки (как диспетчер доступа к диску из раздела 7.3). Процесс может определить, пуста ли в данный момент очередь канала, с помощью вызова функции

```
empty(ch)
```

Эта функция возвращает логическое значение "истина", если в канале `ch` сообщений нет, иначе — "ложь". В отличие от того, что происходит при выполнении соответствующего примитива для условной переменной монитора, процесс может вызвать примитив `empty` и получить значение "истина", но в действительности к моменту продолжения работы процесса в очереди уже могут быть сообщения. Кроме того, если процесс вызывает примитив `empty` и получает результат `false`, то при попытке получить сообщение из канала очередь может оказаться пустой. (Вторая ситуация невозможна, если сообщения из канала получает только один процесс.) Хотя примитив `empty` полезен, использовать его нужно очень осторожно.

7.2. Фильтры: сортирующая сеть

Фильтр — это процесс, который получает сообщения из одного или нескольких входных каналов и отправляет сообщения в один или несколько выходных каналов. Выход фильтра является функцией от его входа и начального состояния. Следовательно, фильтр описывается предикатом, который связывает значения сообщений, отправляемых в выходные каналы, со значениями сообщений, полученных из таких каналов. Действия, выполняемые фильтром в ответ на получение входа, должны обеспечивать описанное предикатом отношение каждый раз, когда фильтр отправляет выход.

Чтобы проиллюстрировать разработку и программирование фильтров, рассмотрим задачу сортировки списка из `n` чисел в порядке возрастания. Простейший способ решения этой задачи — написать процесс-фильтр `Sort`, который получает вход из одного канала, использует один из стандартных алгоритмов сортировки и записывает результат в другой канал. Пусть `input` — входной канал, `output` — выходной. Предположим, что `n` чисел для сортировки отправляются в канал `input` некоторым (неважно, каким именно) процессом. Цель сортирующего процесса — обеспечить, чтобы числа, отправляемые в `output`, были упорядочены и являлись перестановкой значений, полученных из `input`. Пусть `sent[i]` обозначает `i`-е значение, переданное в `output`. Итак, цель описывается следующим предикатом.

SORT: `(∀ i: 1 <= i < n: sent[i] <= sent[i+1]) ∧`
значения, отправленные в `output`, *являются перестановкой значений, полученных из* `input`

Схема процесса `Sort` выглядит таким образом.

```
process Sort {
  получить все числа из канала input;
  отсортировать числа;
  передать отсортированные числа в канал output;
}
```

Поскольку примитив `receive` является блокирующим, процессу `Sort` важно определить момент, когда он получил все числа. Одно из решений — процесс `Sort` заранее знает значение `n`. Более общее решение — число `n` передается первым, а за ним на вход поступают `n` чисел для сортировки. Еще более общее решение — заканчивать входной поток особым *сигнальным* значением (маркером), которое определяет, что все числа были получены. Последнее решение является наиболее общим, поскольку процесс, порождающий вход, может и сам заранее не знать, сколько чисел он передаст.

Если процессы являются "тяжеловесными" объектами (как в большинстве операционных систем), то использованный в процессе `Sort` подход будет наиболее эффективным способом решения задачи сортировки. Но есть и другой способ, подходящий для аппаратной реализации. Используется сеть небольших процессов, которые выполняются параллельно и взаимодействуют между собой. (Смешанный подход — использовать сеть процессов среднего размера.) Существует много типов сортирующих сетей и различных алгоритмов внутренней сортировки. Рассмотрим *сеть слияния*.

Идея сети слияния — циклически и параллельно сливать два отсортированных списка в один более длинный отсортированный список. Сеть состоит из процессов-фильтров `Merge`. Каждый фильтр `Merge` получает значения из двух упорядоченных входных потоков `in1` и `in2` и производит один упорядоченный выходной поток `out`. Предположим, что окончание входных потоков обозначается маркером `EOS`, как было описано выше. Фильтр `Merge` добавляет `EOS` в конец выходного потока. Если есть `n` входных чисел (без маркеров), то после завершения процесса `Merge` должен выполняться следующий предикат.

MERGE: `in1` *и* `in2` *пусты* ∧ `sent[n+1] == EOS` ∧
`(∀ i: 1 <= i < n: sent[i] <= sent[i+1])` ∧
значения, отправленные в канал `out`, *являются перестановкой значений, полученных из каналов* `in1` *и* `in2`

Первая строка предиката *MERGE* говорит о том, что был получен весь вход, а в конец потока `out` был добавлен маркер `EOS`. Вторая — что выход упорядочен. Последние две строки определяют, что выход является перестановкой входа.

Один способ реализовать процесс `Merge` — получить все входные числа, объединить их и отправить построенный список в канал `out`. Однако для этого придется запомнить все входные числа. Поскольку входные потоки упорядочены, для реализации процесса `Merge` лучше циклически сравнивать числа, получаемые из потоков `in1` и `in2`, и отправлять в `out` меньшее из них. Пусть это будут числа `v1` и `v2`. Получим процесс-фильтр, представленный в листинге 7.2.

Для построения сортирующей сети можно использовать набор процессов `Merge` и массивы входных и выходных каналов. При условии, что количество входных чисел `n` является степенью числа 2, процессы и каналы соединяются так, чтобы полученная схема взаимодействия образовывала дерево (рис. 7.1). Информация в сортирующей сети идет слева направо. Каждый узел слева получает и сравнивает два числа, формируя поток из двух отсортированных чисел, и так далее. Крайний справа узел производит окончательный отсортированный поток. Сортирующая сеть содержит `n-1` процессов, ширина сети — $log_2 n$.

В сортирующей сети на рис. 7.1 входные и выходные каналы должны быть разделяемыми. В частности, выходной канал, используемый экземпляром процесса `Merge`, должен совпадать со входным каналом, который используется следующим в графе экземпляром процесса `Merge`. Это можно запрограммировать двумя способами. Первый способ — использовать *статическое именование*: все каналы объявляются как глобальный массив, а каждый экземпляр процесса `Merge` получает данные из двух элементов массива и отправляет в другой элемент. Для этого нужно "встроить" дерево в массив, чтобы каналы, используемые процессом $Merge_i$, были функцией числа `i`. Второй способ — использовать *динамическое именование*: как и раньше, все каналы объявляются глобальными, но процессы при создании в качестве

параметров получают по три канала. Это упрощает программирование процессов Merge, поскольку их текст идентичен. Но для этого способа нужен главный процесс, который динамически создает каналы и передает их в качестве параметров различным процессам Merge.

Листинг 7.2. Процесс-фильтр для слияния двух входных потоков

```
chan in1(int), in2(int), out(int);

process Merge {
  int v1, v2;
  receive in1(v1); # получить первые два входных числа
  receive in2(v2);
  # меньшее из чисел отправить в выходной канал и повторить
  while (v1 != EOS and v2 != EOS) {
    if (v1 <= v2)
      { send out(v1); receive in1(v1); }
    else # (v2 < v1)
      { send out(v2); receive in2(v2); }
  }
# получить остаток непустого входного канала
  if (v1 == EOS)
    while (v2 != EOS)
      { send out(v2); receive in2(v2); }
  else # (v2 == EOS)
    while (v1 != EOS)
      { send out(v1); receive in1(v1); }
  # добавить маркер в выходной канал
  send out(EOS);
}
```

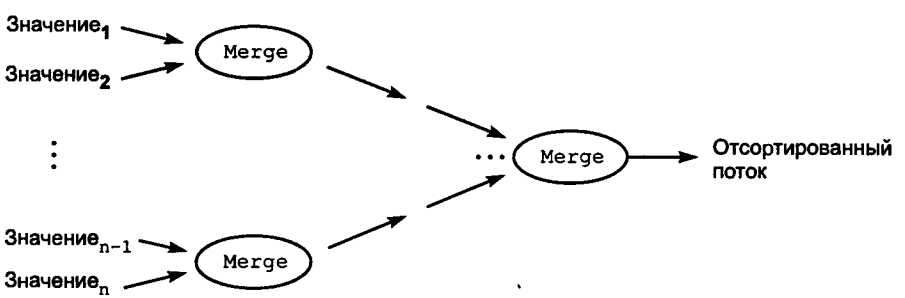


Рис. 7.1. Сортирующая сеть из процессов Merge

Главное свойство таких фильтров, как Merge, — это возможность их соединения самыми разными способами. Нужно только, чтобы выход, производимый одним фильтром, удовлетворял требованиям ко входу другого фильтра. Важное следствие из этого свойства — если внешнее поведение входа и выхода одинаково, то можно заменить один процесс-фильтр (или сеть фильтров) другим процессом (сетью). Например, процесс Sort, описанный выше, можно заменить сетью процессов Merge вместе с процессом (или сетью), который распределяет числа по сети слияния.

Сети фильтров можно использовать при решении многих задач параллельного программирования. Например, в разделе 7.6 рассматривается генерация простых чисел, а в разделе 9.3 — распределенное умножение матриц. В упражнениях описаны дополнительные приложения.

7.3. Клиенты и серверы

Напомним, что сервер — это процесс, который постоянно обрабатывает запросы от клиентских процессов. В этом разделе показано, как использовать асинхронную передачу сообщений для программирования серверов и клиентов. В первых примерах демонстрируется, как превратить мониторы в серверы и реализовать диспетчеры ресурсов. Эти примеры также указывают на *дуальность* мониторов и передачи сообщений: каждый из них может моделировать поведение другого.

Далее показано, как реализовать планирующий драйвер диска и файловый сервер. Планирующий драйвер диска иллюстрирует третий способ решения задачи планирования доступа к диску (см. раздел 5.3), а файловый сервер — важный метод программирования, который называется *непрерывностью диалога*. В обоих примерах демонстрируются программные структуры, которые непосредственно поддерживаются передачей сообщений, т.е. применение этих структур приводит к более компактным решениям, чем при использовании любого механизма синхронизации, основанного на разделяемых переменных.

7.3.1. Активные мониторы

Монитор управляет ресурсом. Он инкапсулирует постоянные переменные, запоминающие состояние ресурса, и предоставляет набор процедур для доступа к ресурсу. Кроме того, процедуры выполняются со взаимным исключением и используют для условной синхронизации переменные условий. Здесь показано, как моделировать эти свойства мониторов, используя серверные процессы и передачу сообщений, т.е. как программировать мониторы в виде активных процессов, а не пассивных наборов процедур.

Предположим пока, что в мониторе есть только одна операция `op` и она не использует условных переменных. Структура монитора такова.

```
monitor Mname {
  объявления постоянных переменных;
  код инициализации;
  procedure op(формальные параметры) {
    тело процедуры;
  }
}
```

Предположим, что *MI* — это инвариант монитора, т.е. предикат, определяющий состояние постоянных переменных в моменты, когда нет активных вызовов.

Для моделирования монитора `Mname` с помощью передачи сообщений используем один серверный процесс `Server`. Постоянные переменные монитора `Mname` становятся локальными переменными этого процесса, т.е. он становится их владельцем (caretaker). Сервер сначала инициализирует переменные, затем выполняет один и тот же цикл, в котором он обслуживает "вызовы" процедуры `op`. Вызов имитируется следующим образом: сначала клиентский процесс отправляет сообщение в канал запроса, а потом получает результат из канала ответа. Таким образом, сервер постоянно получает запросы из канала запроса и отправляет результаты в каналы ответа (результата). Формальные параметры монитора `Mname` становятся дополнительными переменными, локальными в сервере. Чтобы избежать перехвата одним клиентом результатов, предназначенных другому, каждому клиенту нужен собственный канал ответа. Если каналы ответа объявлены как глобальный массив, то клиент в сообщении запроса должен передавать индекс элемента массива для своего канала ответа. (Некоторые примитивы передачи сообщений позволяют получающему процессу определить идентификатор отправителя.)

В листинге 7.3 приведены схемы сервера и его клиентов. Инвариант монитора *MI* теперь стал инвариантом цикла в процессе `Server`. Он становится истинным после выполнения кода инициализации, а также перед обслуживанием каждого запроса и после него. Точнее, он

выполняется во всех точках программы, где Server взаимодействует с клиентскими процессами. Как показано в листинге 7.3, после отправки запроса клиент немедленно переходит в состояние ожидания ответа. Однако клиент мог бы выполнять другую работу до перехода в состояние ожидания, если она есть. Это случается не часто, но возможно, поскольку вызов моделируется двумя отдельными примитивами — send и receive.

Листинг 7.3. Клиенты и сервер с одной операцией

```
chan request(int clientID, типы входных данных);
chan reply[n](типы результатов);

process Server {
  int clientID;
  объявления других постоянных переменных;
  код инициализации;
  while (true) {       ## инвариант цикла MI
    receive request(clientID, входные значения);
    код из тела операции op;
    send reply[clientID](результаты);
  }
}

process Client[i = 0 to n-1] {
  send request(i, аргументы-значения); # "вызов" op
  receive reply[i](аргументы для результатов); # ждать ответа
}
```

В программе в листинге 7.3 используется статическое именование, поскольку каналы глобальны по отношению к процессам, т.е. к ним можно обращаться напрямую. Следовательно, при кодировании процессов необходимо внимательно следить, чтобы они использовали правильные каналы. Например, процесс Client[i] не должен использовать канал ответа другого процесса Client[j]. Можно воспользоваться динамическим именованием, когда каждый клиент создает собственный канал ответа и передает его серверу Server в первом поле запроса request (вместо целочисленного индекса). Тогда клиенты не смогут получить доступ к "чужим" каналам ответа, и появится возможность динамически изменять количество клиентов. (В листинге 7.3 число клиентов n фиксировано.)

Обычно монитор имеет несколько процедур. Дополним программу в листинге 7.3 так, чтобы клиент сообщал, какую операцию он вызывает. Для этого в сообщения запроса включается дополнительный аргумент, имеющий перечислимый тип с одной константой для каждого вида операции. У различных операций будут разные наборы аргументов и типы результатов, поэтому для программирования понадобятся записи с вариантами и объединения. (Можно также передавать аргументные части сообщений запроса и ответа в виде строк байтов; клиенты и сервер должны будут кодировать и декодировать эти строки.)

В листинге 7.4 приведены схемы клиентов и сервера с поддержкой нескольких операций. Оператор if в сервере работает как оператор case с одной веткой для каждого вида операции. Тело операции получает аргументы из переменной args и помещает результаты в переменную results. После выполнения оператора if процесс Server отправляет результаты соответствующему клиенту.

Листинг 7.4. Клиенты и сервер с несколькими операциями

```
type op_kind = enum(op1, ..., opn);
type arg_type = union(arg1, ..., argn);
type result_type = union(res1, ..., resn);
```

```
chan request(int clientID, op_kind, arg_type);
chan reply[n](res_type);

process Server {
  int clientID; op_kind kind; arg_type args;
  res_type results; объявления других переменных;
  код инициализации;
  while (true) {       ## инвариант цикла MI
    receive request(clientID, kind, args);
    if (kind == op1)
      { тело op1; }
    ...
    else if (kind == opn)
      { тело opn; }
    send reply[clientID](results);
  }
}

process Client[i = 0 to n-1] {
  arg_type myargs; result_type myresults;
  поместить аргументы-значения в myargs;
  send request(i, opj, myargs);  # "вызов" opj
  receive reply[i](myresults);  # ждать ответа
}
```

До сих пор предполагалось, что монитор Mname не использует условных переменных. Следовательно, процессу Server не приходится задерживаться, обслуживая запрос, поскольку тело каждой операции содержит только последовательные операторы. Теперь покажем, как работать с более общим вариантом монитора, который поддерживает несколько операций и использует условную синхронизацию. (Клиенты остаются без изменений, поскольку они по-прежнему просто вызывают операции, отправляя запрос, а затем получают ответ; очевидно, что запрос может быть обслужен с задержкой.)

Чтобы показать, как монитор с условными переменными преобразуется в процесс-сервер, начнем с частного примера, а затем опишем, как его обобщить. Рассмотрим задачу управления многоэлементным ресурсом, таким как память или блоки файла. Клиенты получают элементы ресурса, используют их, а затем освобождают и возвращают диспетчеру. Для простоты предположим, что клиенты за один раз получают и освобождают по одному элементу. В листинге 7.5 показана реализация монитора для такого распределителя ресурсов. В программе использован метод передачи условия (см. раздел 5.1), поскольку структура этой программы больше всего подходит для преобразования в серверный процесс. Свободные элементы ресурса хранятся в наборе, доступном с помощью операций вставки и удаления.

Монитор для распределения ресурса имеет две операции, поэтому у эквивалентного ему серверного процесса будет общая структура, как в листинге 7.4. Главное отличие состоит в том, что, когда нет доступных элементов ресурса, сервер не может ожидать, обслуживая запрос. Он должен запомнить запрос и отложить посылку ответа. Позже, когда элемент ресурса освободится, сервер должен будет вернуться к сохраненному запросу (если он есть) и передать освободившийся элемент запрашивающему процессу.

В листинге 7.6 приведены схемы сервера для распределения ресурсов и его клиентов. Теперь в сервере есть вложенные операторы `if`. Внешние операторы имеют ветви для каждого вида операций, а внутренние соответствуют операторам `if` в процедурах монитора. После отправки сообщения запроса клиент ждет, чтобы получить элемент ресурса. Однако, отправив сообщение об освобождении ресурса, клиент не ждет, пока это сообщение будет обработано, поскольку в этом нет необходимости.

Листинг 7.5. Монитор распределения ресурсов

```
monitor Resource_Allocator {
  int avail = MAXUNITS;
  set units = начальные значения;
  cond free; # получает сигнал,
             # когда процессу нужен элемент ресурса

  procedure acquire(int &id) {
    if (avail == 0)
      wait(free);
    else
      avail = avail-1;
    remove(units, id);
  }

  procedure release(int id) {
    insert(units, id);
    if (empty(free))
      avail = avail+1;
    else
      signal(free);
  }
}
```

Листинг 7.6. Сервер распределения ресурса и клиенты

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitid);
chan reply[n](int unitID);

process Allocator {
  int avail = MAXUNITS; set units = начальные значения;
  queue pending; # в начальном состоянии пуста
  int clientID, unitID; op_kind kind;
  объявления других локальных переменных;
  while (true) {
    receive request(clientID, kind, unitID);
    if (kind == ACQUIRE) {
      if (avail > 0) { # удовлетворить запрос
          avail--; remove(units, unitID);
          send reply[clientID](unitID);
      } else # запомнить запрос
          insert(pending, clientID);
    } else {    # kind == RELEASE
      if empty(pending) {
                # возвратить unitID в множество элементов
         avail++; insert(units, unitid);
      } else { # зарезервировать unitID для ожидающего клиента
          remove(pending, clientID);
          send reply[clientID](unitID);
      }
    }
  }
}

process Client[i = 0 to n-1] {
  int unitID;
```

```
  send request(i, ACQUIRE, 0)        # "вызов" запроса
  receive reply[i](unitID);
  # использовать ресурс unitID, затем освободить его
  send request(i, RELEASE, unitID);
   ...
}
```

Этот пример показывает, как имитировать частный случай монитора с помощью процесса-сервера. Такую же базовую схему можно использовать для имитации любого монитора, запрограммированного с помощью метода передачи условия. Но во многих мониторах (см. главу 5) есть операторы `wait`, записанные в циклах, или операторы `signal`, выполняемые безусловно. Чтобы моделировать такие операторы `wait`, серверу придется сохранять ожидающий запрос (как в листинге 7.6) и записывать, какие действия нужно будет совершить для обслуживания запроса. Для моделирования безусловного оператора `signal` сервер должен проверить очередь ожидающих запросов. Если она пуста, сервер ничего не делает, а если ожидающий запрос есть, сервер удаляет его из очереди и обрабатывает *после* обработки операции, содержащей `signal`. Конкретные детали работы зависят от того, какой монитор моделируется; несколько частных случаев рассмотрены в упражнениях.

Монитор `Resource_Allocator` в листинге 7.5 и сервер `Allocator` в листинге 7.6 демонстрируют *дуальность* мониторов и передачи сообщений, т.е. соответствие механизмов, используемых в программах с мониторами и в программах с передачей сообщений. Эти соответствия собраны в табл. 7.1.

Таблица 7.1. Дуальность мониторов и передачи сообщений

Программы с мониторами	Программы с передачей сообщений
Постоянные переменные	Локальные переменные сервера
Идентификаторы процедур	Канал запроса и виды операций
Вызов процедуры	`send request(); receive reply`
Вход в монитор	`receive request()`
Возврат в процедуру	`send reply()`
Оператор `wait`	Сохранение ожидающего запроса
Оператор `signal`	Получение и обработка ожидающего запроса
Тела процедур	Ветви оператора выбора по видам операций

Поскольку тела процедур монитора аналогичны ветвям оператора выбора в сервере, относительная производительность программ с мониторами по сравнению с программами на основе передачи сообщений зависит только от относительной эффективности реализации различных механизмов. На машинах с разделяемой памятью вызовы процедур и действия с условными переменными более эффективны, чем примитивы передачи сообщений. Поэтому большинство операционных систем для таких машин реализуются на основе мониторов. С другой стороны, большинство распределенных систем основаны на передаче сообщений, поскольку она является и эффективной, и подходящей абстракцией. Возможно также сочетание обоих стилей и реализаций; это будет продемонстрировано в главе 8 при обсуждении удаленных вызовов процедур и рандеву. Фактически это усиливает аналогичность мониторов и передачи сообщений.

7.3.2. Планирующий сервер диска

В разделе 5.3 рассматривалась задача планирования доступа к диску и приводились две основные структуры ее решения. В первом решении (см. рис. 5.4) диспетчер доступа к диску

был отдельным от диска монитором, поэтому клиенты сначала вызывали диспетчер, чтобы запросить доступ, затем пользовались диском, а в конце вызывали диспетчер для освобождения доступа. Во втором решении (см. рис. 5.5) диспетчер был посредником между клиентами и процессом обслуживания диска. Клиенты должны были вызывать только одну операцию монитора. (Была рассмотрена и третья структура решения, в которой использовались вложенные мониторы. При программировании с передачей сообщений это решение, по существу, эквивалентно решению с монитором-посредником.)

Описанные структуры легко имитируются в программе, основанной на передаче сообщений, если реализовать мониторы в виде процессов-серверов, используя методы из предыдущего раздела. Но с использованием передачи сообщений можно еще упростить структуру программы, объединив посредник и драйвер диска (см. рис. 5.5) в единый самостоятельно планирующий процесс-сервер. (С мониторами так поступить нельзя, поскольку монитор используется для реализации взаимодействия между клиентами и драйвером диска.)

Структуры для решения задачи планирования доступа к диску представлены на рис. 7.2. Во всех вариантах предполагается, что диск управляется процессом-сервером, выполняющим всю работу по доступу к диску. Принципиальные различия между этими тремя структурами состоят в клиентском интерфейсе, как было сказано выше, и в числе сообщений, которыми процессы должны обменяться для доступа к диску (это описано далее).

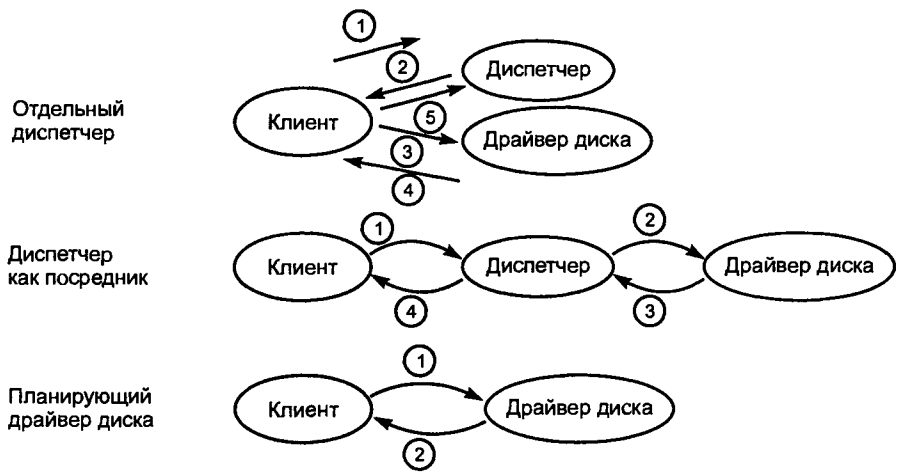


Рис. 7.2. Структуры планирования доступа к диску с передачей сообщений

Если диспетчер отделен от сервера диска, для одного доступа к диску нужны пять сообщений: два для запроса на планирование и получения ответа, два для запроса на доступ к диску и получения ответа и одно для освобождения диска. Клиент участвует во всех пяти взаимодействиях. Если диспетчер является посредником, нужно обменяться четырьмя видами сообщений: клиент отправляет запрос и дожидается получения ответа, драйвер диска запрашивает диспетчер и получает ответ. (Процесс драйвера может возвращать результат для одного запроса на доступ к диску, выполнив следующий запрос к диспетчеру.) Как показано на рис. 7.2, наиболее привлекательна структура, в которой используется планирующий драйвер диска. В ней нужен обмен всего двумя сообщениями. В оставшейся части этого раздела показано, как запрограммировать такой драйвер.

Если бы драйвер диска не выполнял планирования, т.е. доступ к диску осуществлялся по принципу "первым пришел — первым обслужен", то он имел бы такую же структуру, как и сервер `Server` (см. листинг 7.3). Чтобы осуществлять планирование, драйвер должен просматривать все ожидающие запросы, т.е. ему нужно получить все сообщения, ожидающие в канале `request`. Драйвер получает сообщения, выполняя цикл, который завершается, когда канал `request` пуст и есть хотя бы один сохраненный запрос. Затем драйвер выбирает наи-

более подходящий запрос, обращается к диску и отправляет ответ клиенту, приславшему запрос. Драйвер может использовать любую из стратегий планирования диска, описанных в разделе 5.3.

В листинге 7.7 приведена схема драйвера, использующего стратегию планирования кратчайшего времени поиска (shortest-seek-time — SST). Драйвер записывает ожидающие запросы в одну из двух упорядоченных очередей, `left` или `right`, в зависимости от направления, в котором для обработки запроса необходимо перемещать головки диска из их текущего положения. Запросы в очереди `left` упорядочены по уменьшению номера цилиндра, а в очереди `right` — по возрастанию. Переменная `headpos` запоминает текущую позицию головок, а `nsaved` — число сохраненных запросов. Инвариант для внешнего цикла драйвера будет таким.

SST: `left` — *это упорядоченная очередь от наибольшего к наименьшему* `cyl` ∧
все значения `cyl` *в* `left` *не больше* `headpos` ∧
`right` — *это упорядоченная очередь от наименьшего к наибольшему* `cyl` ∧
все значения `cyl` *в* `right` *не меньше* `headpos` ∧
`(nsaved == 0)` ⇒ *и* `left`, *и* `right` *пусты*

Примитив `empty` в условии внутреннего цикла `while` (листинг 7.7) позволяет определить, есть ли еще сообщения в очереди канала `request`. Это пример метода программирования, который называется *опросом*. Процесс драйвера диска постоянно опрашивает канал `request`, чтобы определить, есть ли ожидающие запросы. Если они есть, драйвер получает еще один запрос, так что теперь у него больше запросов для выбора наиболее подходящего. Если ожидающих запросов нет, то драйвер обслуживает наиболее подходящий из сохраненных запросов. Опрос бывает полезен во многих ситуациях и часто реализуется аппаратно, например, для распределения доступа к шине взаимодействия.

Листинг 7.7. Планирующий драйвер диска

```
chan request(int clientID, int cyl, типы других аргументов);
chan reply[n](типы результатов);

process Disk_Driver {
  queue left, right; # упорядоченные очереди сохраненных запросов
  int clientID, cyl, headpos = 1, nsaved = 0;
  переменные для запоминания других аргументов запроса;
  while (true) {   ## инвариант цикла SST
    while (!empty(request) or nsaved == 0) {
      # ждать первого запроса или получить еще один
      receive request(clientID, cyl, ...);
      if (cyl <= headpos)
        insert(left, clientID, cyl, ...);
      else
        insert(right, clientID, cyl, ...);
      nsaved++;
    }
    # выбрать наилучший запрос из очередей left и right
    if (size(left) == 0)
      remove(right, clientID, cyl, args);
    else if (size(right) == 0)
      remove(left, clientID, cyl, args);
    else
      удалить запрос, который в left или right ближе всего к headpos;
    headpos = cyl; nsaved--;
    получить доступ к диску;
    send reply[clientID](results);
  }
}
```

7.3.3. Файловые серверы: непрерывность диалога

В качестве последнего примера взаимодействия типа клиент-сервер представим один из способов реализации файловых серверов — процессов, обеспечивающих доступ к файлам на диске. Для работы с файлом клиент сначала должен его открыть. Если файл можно открыть, то он существует, и клиент имеет право работать с ним, выполняя серию запросов на чтение или запись. В конце концов клиент закрывает файл.

Предположим, что одновременно могут быть открыты не более n файлов, а доступ к каждому открытому файлу обеспечивается отдельным процессом файлового сервера; следовательно, может быть не более n активных файловых серверов. Когда клиент хочет открыть файл, ему сначала нужно получить свободный файловый сервер. Если все файловые серверы идентичны, то подойдет любой свободный.

Можно было бы выделять клиентам файловые серверы с помощью отдельного процесса-распределителя. Но поскольку все они идентичны, а каналы связи разделяемы, можно поступить намного проще. Допустим, есть глобальный канал open. Для получения файлового сервера клиент отправляет запрос в канал open. Бездействуя, каждый файловый сервер опрашивает канал open, поэтому запрос от клиента будет получен одним из не занятых работой файловых серверов. Этот сервер отправляет клиенту ответ и ожидает запросов на доступ к диску. Клиент отправляет их в другой канал, access[i], где i — индекс файлового сервера, ответившего на запрос клиента об открытии файла. Таким образом, access является массивом из n каналов. Когда клиент закрывает файл, сервер вновь освобождается и ждет запросов на открытие файлов.

В листинге 7.8 приведена схема файловых серверов и их клиентов. Клиент отправляет запросы READ и WRITE в один и тот же серверный канал. Это необходимо, поскольку файловый сервер не может знать порядка, в котором совершаются запросы, и, следовательно, не может использовать для них отдельные каналы. По этой же причине для закрытия канала клиент должен послать запрос CLOSE именно в него.

Листинг 7.8. Файловые серверы и клиенты

```
type kind = enum(READ, WRITE, CLOSE);
chan open(string fname; int clientID);
chan access[n](int kind, типы других аргументов);
chan open_reply[m](int serverID); # id сервера или ошибка
chan access_reply[m](типы результатов); # данные, ошибка, ...

process File_Server[i = 0 to n-1] {
  string fname; int clientID;
  kind k; переменные для других аргументов;
  bool more = false;
  переменные для локального буфера, кэш-памяти и т.д.;
  while (true) {
    receive open(fname, clientID);
    открыть файл fname; если успешно, то:
    send open_reply[clientID](i); more = true;
    while (more) {
      receive access[i](k, другие аргументы);
      if (k == READ)
        обработать запрос на чтение;
      else if (k == WRITE)
        обработать запрос на запись;
      else   # k == CLOSE
        { закрыть файл; more = false; }
      send access_reply[clientID](results);
```

```
    }
  }
}

process Client[j = 0 to m-1] {
  int serverID; объявления других переменных;
  send open("foo", j);       # открыть файл "foo"
  receive open_reply[j](serverID); # получить id сервера
  # использовать файл, затем закрыть его:
  send access[serverID](аргументы доступа);
  receive access_reply[j](результаты);
  ...
}
```

Взаимодействие между клиентом и сервером в листинге 7.8 является примером *непрерывности диалога*. Клиент начинает диалог с файловым сервером, когда тот получает запрос клиента open. Затем клиент продолжает общаться с этим же сервером. Для этого сервер программируется так, что сначала получает запрос из канала open, а затем — запросы из "своего" элемента массива access.

Программа в листинге 7.8 иллюстрирует один из способов реализации файловых серверов. Предполагается, что канал open совместно используется файловыми серверами, и каждый из них может получить из канала сообщение. Если у каждого канала может быть только один получатель, то нужен отдельный процесс-распределитель. Этот процесс получает запросы на открытие файла и выделяет клиенту файловый сервер. Файловые серверы, в свою очередь, должны сообщать процессу-распределителю, когда они свободны.

В решении в листинге 7.8 используется фиксированное число n файловых серверов. В языках программирования, поддерживающих динамическое создание процессов и каналов, лучше динамически создавать файловые серверы и каналы для доступа к ним по мере необходимости. Тогда в любой момент времени в системе будет ровно столько серверов, сколько в действительности используется; важнее то, что число файловых серверов не будет ограничено. Другая крайность — иметь по одному файловому серверу на каждый диск. Но тогда или файловый сервер, или интерфейс клиента будет намного сложнее, чем показанный в листинге 7.8, поскольку либо сервер должен отслеживать информацию, связанную со всеми клиентами, открывшими файлы, либо клиенты должны передавать информацию о состоянии файла в каждом запросе.

Еще один подход, используемый в файловой системе Sun Network File System (NFS), состоит в том, что доступ к файлам реализован исключительно с помощью удаленных процедур. "Открытие" файла состоит из получения файлового дескриптора и набора атрибутов файла, которые затем передаются в каждом вызове процедуры доступа к файлу. В отличие от процессов File_Server (см. листинг 7.8), процедуры доступа к файлам в NFS сами по себе не имеют состояний — вся необходимая для доступа к файлу информация передается в аргументах каждого вызова этих процедур. В результате возрастают затраты на передачу аргументов, но существенно упрощается обработка сбоев как сервера, так и клиентов. Например, если происходит сбой в работе файлового сервера, то клиент просто повторяет свои запросы, пока не получит ответ. Если же происходит сбой в работе клиента, серверу не нужно делать ничего, поскольку он не содержит информацию о состоянии.

7.4. Взаимодействующие равные: обмен значений

В предыдущих примерах данной главы было показано, как использовать передачу сообщений для программирования фильтров, клиентов и серверов. Здесь рассматривается простой пример взаимодействующих равных, в котором сравниваются три полезные схемы взаи-

модействия: централизованная, симметричная и кольцевая. В главах 9 и 11 приведены примеры использования этих схем взаимодействия. Они часто встречаются в распределенных параллельных вычислениях и в децентрализованных распределенных системах.

Пусть есть n процессов, каждый из которых имеет локальную целочисленную переменную v. Задача процессов — определить наименьшее и набольшее значения среди этих n переменных. Эту задачу можно решить так: один из процессов, например P[0], собирает все n чисел, находит минимальное и максимальное из них и рассылает результаты остальным процессам. Централизованное решение, в котором используется этот способ, показано в листинге 7.9. В нем используется 2(n-1) сообщений. (Процесс P[0] может использовать примитив broadcast, передающий копии сообщения всем остальным процессам; тогда общее число сообщений будет n.)

Листинг 7.9. Обмен значений: централизованное решение

```
chan values(int), results[n](int smallest, int largest);

process P[0] {    # управляющий процесс
  int v;  # считается, что v инициализирована
  int new, smallest = v, largest = v; # начальное состояние
  # собрать числа, запомнить максимальное и минимальное
  for [i = 1 to n-1] {
    receive values(new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
  # разослать результаты остальным процессам
  for [i = 1 to n-1]
    send results[i](smallest, largest)
}

process P[i = 1 to n-1] {
  int v;  # считается, что v инициализирована
  int smallest, largest;
  send values(v);
  receive results[i](smallest, largest);
}
```

В программе в листинге 7.9 всю "работу" по определению максимального и минимального значений выполняет один процесс; остальные просто отправляют свои значения и ждут результата. Второй способ решения задачи — использовать симметричный подход, в котором все процессы выполняют один и тот же алгоритм. Каждый процесс сначала отправляет свое значение всем остальным, затем все процессы параллельно вычисляют минимум и максимум из n значений (листинг 7.10). Это решение имеет структуру типа "одна программа — много данных" (single program, multiple data — SPMD). Все процессы выполняют одну и ту же программу, но работают с разными данными. Всего используется n(n-1) сообщений. (Если доступен примитив broadcast, число сообщений сократится до n.)

Листинг 7.10. Обмен значений: симметричное решение

```
chan values[n](int);

process P[i = 0 to n-1] {
  int v;  # считается, что v инициализирована
  int new, smallest = v, largest = v; # начальное состояние
```

```
  # отправить мое значение всем остальным процессам
  for [j = 0 to n-1 st j != i]
    send values[j](v);
  # собрать значения, запомнить минимум и максимум
  for [j = 1 to n-1] {
    receive values[i](new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
}
```

Третий способ решения заключается в организации процессов в логическое кольцо, в котором каждый процесс `P[i]` получает сообщения от своего предшественника и отправляет сообщения преемнику. Например, процесс `P[0]` отправляет сообщения процессу `P[1]`, тот — процессу `P[2]` и т.д., `P[n-1]` — `P[0]`. Каждый процесс проходит две стадии. На первой стадии процесс получает два числа, определяет минимальное и максимальное из них и своего значения, затем отправляет результаты своему преемнику. На второй стадии процесс получает значения глобальных минимума и максимума и передает их преемнику. Один процесс, скажем `P[0]`, действует как инициатор вычислений (листинг 7.11). Это решение почти симметрично (немного отличается лишь процесс `P[0]`), и в нем используется `2(n-1)` сообщений.

Листинг 7.11. Обмен значений: кольцевое решение

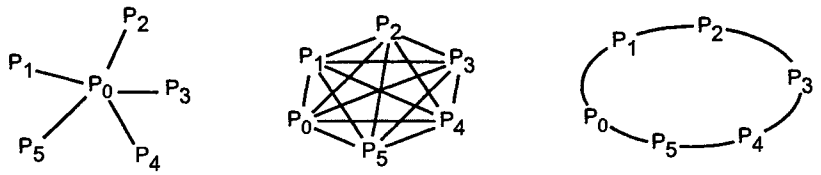
```
chan values[n](int smallest, int largest);

process P[0] {     # процесс-инициализатор
  int v;  # считается, что v инициализирована
  int smallest = v, largest = v; # начальное состояние
  # послать v следующему процессу, P[1]
  send values[1](smallest, largest);
  # получить глобальные максимум и минимум от P[n-1] и
  # отправить их процессу P[1]
  receive values[0](smallest, largest);
  send values[1](smallest, largest);
}

process P[i = 1 to n-1] {
  int v;  # считается, что v инициализирована
  int smallest, largest;
  # получить текущие минимум и максимум и
  # обновить их, сравнивая с v
  receive values[i](smallest, largest);
  if (v < smallest)
      smallest = v;
  if (v > largest)
      largest = v;
  # отправить результат следующему процессу, затем ожидать
  # получения глобальных результатов
  send values[(i+1) mod n](smallest, largest);
  receive values[i](smallest, largest);.
}
```

На рис. 7.3 показаны структуры взаимодействия рассмотренных программ для 6 процессов. Процессы представлены вершинами графов, а пары каналов взаимодействия — ребрами. Как вид-

но, граф централизованного решения имеет форму звезды, в центре которой находится управляющий процесс. Структура симметричного решения — полный граф, у которого каждая вершина соединена со всеми остальными, а кольцевого — замкнутое кольцо, или круговой конвейер.



а) централизованное решение б) симметричное решение в) кольцевое решение

Рис. 7.3. Структуры взаимодействия в трех программах

Симметричное решение является самым коротким и его легко программировать, поскольку все процессы выполняют одни и те же действия. С другой стороны, в нем используется самое большое число сообщений (если только не доступен примитив `broadcast`). Сообщения передаются и принимаются практически одновременно, поэтому, если сеть связи поддерживает одновременные передачи, их можно передавать параллельно. В целом, однако, чем больше сообщений программа отправляет, тем медленнее она выполняется. Иначе говоря, затраты на взаимодействие существенно уменьшают выигрыш в производительности, который можно получить за счет параллельного выполнения. (Подробно эти вопросы обсуждаются во введении к части 3.)

Централизованное и кольцевое решения используют линейное число сообщений, но имеют разные схемы взаимодействия, что приводит к разным характеристикам производительности. В централизованном решении все сообщения передаются управляющему процессу приблизительно одновременно, поэтому существенную задержку выполнения управляющего процесса может вызвать только первый оператор `receive`. Аналогично результаты от управляющего процесса отправляются без задержек, поэтому другие процессы возобновляются достаточно быстро.

В кольцевом решении все процессы одновременно являются и потребителями, и производителями, соединенными в круговой конвейер. Таким образом, последний процесс конвейера (`P[n-1]`) должен дожидаться, пока каждый из остальных процессов (по очереди) не получит сообщение, проведет небольшие вычисления и отошлет результаты следующему процессу в конвейере. Сообщения должны пройти по конвейеру два полных цикла до того, как каждый процесс получит общий результат. Это решение линейно по своей природе, и здесь невозможно совместить сообщения или избавиться от задержек в операторах `receive`. Следовательно, кольцевое решение не годится для данной простой задачи. С другой стороны, эта структура взаимодействия очень эффективна, если получаемые сообщения можно быстро передавать дальше, или если каждый процесс должен выполнить достаточно большой объем вычислений до получения следующего сообщения. Примеры таких ситуаций приводятся в главах 9 и 11.

7.5. Синхронная передача сообщений

Примитив `send` не приводит к блокировке программы, поэтому процесс, отправляющий сообщение, может работать асинхронно по отношению к процессу, который должен получить сообщение. При синхронной передаче отправка сообщения приводит к блокировке процесса-отправителя до тех пор, пока сообщение не будет получено. Чтобы отличать блокирующую передачу, используем для нее отдельный примитив `synch_send`. Аргументы этих двух примитивов одинаковы — имя канала и передаваемое сообщение, но их семантика различна.

Преимущество синхронной передачи сообщений состоит в том, что размер каналов связи, а следовательно, и объем буфера, ограничены. Причина в том, что процесс может поставить в очередь любого канала не более одного сообщения. Пока это сообщение не будет получено, процесс-отправитель не сможет продолжить работу и отправить новое сообщение. Фактически в реализации синхронной передачи сообщение может оставаться в адресном пространстве отправителя до тех пор, пока адресат не будет готов принять его, а затем сообщение может просто копироваться прямо в адресное пространство процесса-получателя. При такой реализации содержимым канала является просто очередь адресов сообщений, ожидающих отправки.

Однако синхронная передача сообщений имеет два недостатка по сравнению с асинхронной. Во-первых, снижается степень параллельности работы. Когда взаимодействуют два процесса, по меньшей мере один из них блокируется, в зависимости от того, кто первым пытается установить связь — передающий процесс или принимающий. Рассмотрим следующий пример программы типа "производитель-потребитель".

```
channel values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    выполнить некоторые вычисления;
    synch_send values(data[i]);
  }
}

process Consumer {
  int results[n];
  for [i = 0 to n-1] {
    receive values(results[i]);
    выполнить некоторые вычисления;
  }
}
```

Предположим, вычислительные части процессов таковы, что `Producer` иногда работает быстрее, чем `Consumer`, а иногда медленнее. Следовательно, процессы будут подходить к операторам взаимодействия в разное время. Тогда каждая пара операторов отправки и приема сообщений будет задерживаться, и общее время работы увеличится на сумму длительностей всех задержек. При асинхронной передаче сообщений пауз в выполнении вообще может не быть — если быстрее работает `Producer`, то его сообщения накапливаются в очереди канала, а если `Consumer` — в канале может скапливаться достаточное количество сообщений, чтобы примитив `receive` не вызывал блокировку. Чтобы достичь тех же результатов при синхронной передаче сообщений, необходимо поместить между процессами `Producer` и `Consumer` дополнительный буферный процесс. (Как это сделать, показано в следующем разделе.)

Степень параллелизма снижается и при некоторых видах взаимодействия клиент-сервер. Когда клиент освобождает ресурс (как в листинге 7.6), ему обычно не нужно ожидать, пока сервер не получит сообщение об освобождении. Но при синхронной передаче сообщений клиент должен приостановиться. Еще один пример: клиентский процесс хочет вывести данные на дисплей, в файл или какое-нибудь другое устройство вывода, управляемое серверным процессом. Часто клиенту необходимо продолжить работу сразу после передачи запроса на запись; ему не важно, будут данные выведены сразу или в ближайшем будущем — лишь бы это в конце концов было сделано. При синхронной передаче сообщений, однако, клиент *должен* ожидать, пока запрос на запись не будет получен сервером.

Второй недостаток синхронной передачи сообщений состоит в том, что использующие ее программы более предрасположены ко взаимным блокировкам. Например, программист должен следить за тем, чтобы соблюдалось соответствие между операторами передачи

и приема, т.е. если существует процесс, вызывающий оператор передачи (приема), то другой процесс должен в конце концов вызвать оператор приема (передачи) для того же канала. Это типично для взаимодействий "производитель-потребитель", поскольку производитель отправляет, а потребитель получает данные из канала, который их соединяет. Это же типично и для взаимодействий типа клиент-сервер: клиент "вызывает" сервер, выполняя оператор отправки с последующим оператором приема; сервер выполняет оператор приема для получения запроса и оператор отправки, отвечая на запрос.

Однако при программировании взаимодействующих равных нужно быть осторожным. Рассмотрим пример, в котором два процесса обмениваются числовыми значениями.

```
channel in1(int), in2(int);

process P1 {
  int value1 = 1, value2;
  synch_send in2(value1);
  receive in1(value2);
}

process P2 {
  int value1, value2 = 2;
  synch_send in1(value2);
  receive in2(value1);
}
```

Эта программа зависнет, поскольку оба процесса заблокируются в операторах `synch_send`. В одном из процессов (но не в обоих) нужно сначала выполнить оператор `receive`. При асинхронной же передаче сообщений в такой программе прекрасно применяется симметричный подход. Кроме того, при симметричном подходе намного проще изменить программу для работы с большим числом процессов.

Итак, операторы `send` и `synch_send` часто взаимозаменяемы. Все примеры, рассмотренные в этой главе ранее, кроме одного, будут правильно работать, если оператор `send` заменить оператором `synch_send`. Единственное исключение — программа в листинге 7.10, в которой для обмена значениями использован симметричный подход. Таким образом, основное различие между асинхронной и синхронной передачей сообщений связано с противоречием между как можно большей степенью параллелизма и ограниченностью буферов взаимодействия. Поскольку памяти обычно хватает и асинхронные сообщения менее подвержены опасности зависания программы, программисты чаще предпочитают именно их.

7.6. Учебные примеры: CSP

Язык CSP (Communicating Sequential Processes — взаимодействующие последовательные процессы) — это одна из важнейших разработок в истории параллельного программирования. CSP впервые был описан в статье Тони Хоара (Tony Hoare) в 1978 г. В этой работе были представлены синхронная передача сообщений и так называемое *защищенное взаимодействие*. Язык CSP существенно повлиял на разработку языка программирования Occam и нескольких других языков, включая Ada и SR (они описаны в главе 8). Кроме того, язык CSP вызвал многочисленные исследования и публикации по формальным моделям и семантике передачи сообщений.

В данном разделе описан язык CSP. В примерах представлены производители и потребители, клиенты и серверы, а также интересная программа генерации простых чисел с помощью конвейера. Здесь используется синтаксис, подобный описанному в статье Хоара. В конце раздела дается краткое описание языка Occam и последней версии CSP, который стал формальным языком для моделирования и анализа поведения параллельных взаимодействующих систем.

7.6.1. Операторы взаимодействия

Предположим, что процесс A должен передать значение выражения e процессу B. В языке CSP это достигается с помощью следующих фрагментов кода:

```
process A { ... B!e; ... }
process B { ... A?x; ... }
```

Оператор `B!e` называется *оператором вывода*. Он задает процесс назначения B и выражение e, значение которого нужно отправить этому процессу. Оператор `A?x` называется *оператором ввода*. Он указывает процесс-источник A и переменную x, в которую должно быть записано принятое сообщение. Эти операторы называются операторами ввода и вывода, а не отправки и приема, поскольку используются как для внешней связи, так и для межпроцессного взаимодействия. (Оператор вывода ! читается "выдать",? — "запросить".)

При условии, что типы переменных e и x совпадают, указанные выше два оператора называются *согласованными*. Операторы ввода и вывода приостанавливают процесс до тех пор, пока другой процесс не выполнит соответствующий согласованный оператор, после чего оба оператора выполняются одновременно. В результате значение выражения в операторе вывода присваивается переменной в операторе ввода. Выполнение согласованных операторов взаимодействия можно рассматривать как *распределенное присваивание*, при котором значение из одного процесса присваивается переменной в другом. Пока проходит взаимодействие, эти процессы синхронизируются, а затем продолжают работу независимо.

В приведенном выше примере использованы простейшие формы операторов ввода и вывода, имеющих следующий общий вид.

```
Destination!port(e1, ..., en);
Source?port(x1, ..., xn);
```

`Destination` и `Source` обозначают отдельный процесс, как в первом примере, или элемент массива процессов. Этот вид *прямого именования* каналов взаимодействия используется в документации по языку CSP по причине своей простоты. Однако модульные программы легче писать, когда каналы имеют имена, глобальные относительно процессов (как в разделе 7.1). Такой подход использован в описанном ниже языке Occam.

`Source` также может ссылаться на *любой* элемент массива процессов, что записывается как `Source[*]`. Выражение `port` задает канал связи в процессе назначения. Значения выражений e_i оператора вывода отправляются в указанный порт процесса назначения. Оператор ввода получает сообщение в указанный порт от процесса-источника и присваивает значения локальным переменным x_i. Порты используются, чтобы различать виды сообщений, которые должен получать процесс. Если есть только один вид сообщений, порт не указывается. Также, если записано только одно e_i или x_i, можно опустить скобки. В первом примере использованы оба сокращения.

В качестве простого примера рассмотрим процесс-фильтр, который постоянно вводит символ из процесса `West`, а затем выводит его в процесс `East`.

```
process Copy {
  char c;
  do true ->
    West?c;    # ввести символ из процесса West
    East!c;    # вывести символ в процесс East
  od
}
```

Оператор ввода ждет, когда процесс `West` будет готов отправить символ, а оператор вывода — когда процесс `East` будет готов получить его.

Операторы `do` и `if` языка CSP используют представленную Дейкстрой (Dijkstra) нотацию *защищенных команд*. Защищенная команда имеет вид `B -> S`, где B — логическое выражение (защита), а S — список операторов (команда). Оператор `do` задает бесконечный цикл. Пояснения к другим защищенным командам будут даваться по мере их использования.

Еще один пример — процесс-сервер, использующий алгоритм Евклида[12] для вычисления наибольшего общего делителя двух положительных целых чисел x и y.

```
process GCD {
  int id, x, y;
  do true ->
    Client[*]?args(id, x, y);  # ввести "запрос"
    # повторять, пока не x == y:
    do x > y -> x = x - y;
    [] x < y -> y = y - x;
    od
    Client[id]!result(x);  # возвратить результат
  od
}
```

Процесс GCD ждет получения входных данных в порт args от любого элемента массива клиентов. Клиент, достигающий согласованного оператора вывода, также отправляет свой идентификатор. Процесс GCD вычисляет ответ и отправляет его в порт result клиента. Во внутреннем цикле do, если истинно x > y, то из значения x вычитается значение y; если же истинно y > x, то из y вычитается x. Цикл заканчивается, когда не выполняется ни одно из этих условий, т.е. x == y.

Клиент Client[i] взаимодействует с процессом GCD, выполняя такой код.

```
... GCD!args(i,v1,v2); GCD?result(r); ...
```

Имена портов в действительности здесь не нужны, но они помогают понять роль каждого канала.

7.6.2. Защищенное взаимодействие

Операторы ввода и вывода позволяют процессам взаимодействовать. Сами по себе они несколько ограничены, поскольку оба являются блокирующими операторами. Процессу часто нужно взаимодействовать с другими процессами, возможно, через разные порты, даже не зная порядка, в котором они будут связываться с данным процессом. Например, рассмотрим расширенную версию процесса Copy, который может хранить в буфере, скажем, не более 10 символов. Если в буфере находятся больше одного, но меньше десяти символов, то процесс Copy может либо ввести еще один символ из процесса West, либо вывести символ в процесс East. Но процесс Copy не знает, какой из процессов, East или West, первым достигнет согласованного оператора.

Защищенные операторы взаимодействия поддерживают недетерминированное взаимодействие. Они объединяют свойства условных операторов и операторов взаимодействия и имеют следующий вид.

```
B; C -> S;
```

Здесь B — это логическое выражение, C — оператор взаимодействия, S — список операторов. Логическое выражение можно опустить; тогда неявно оно имеет значение true.

Вместе B и C образуют так называемую *защиту*. Защита *пропускает*, если выражение B истинно и выполнение оператора C не приводит к задержке, т.е. если другой процесс уже приостановлен на согласованном операторе взаимодействия. Защита *не пропускает*, если B ложно. Защита *блокирует*, если B истинно, но оператор C еще не может быть выполнен.

Защищенные операторы взаимодействия используются в операторах if и do. Рассмотрим следующий оператор.

```
if B1; C1 -> S1;
[] B2; C2 -> S2
fi
```

[12] Здесь для простоты изложения использован далеко не самый эффективный вариант алгоритма Евклида. — *Прим. ред.*

Этот оператор имеет две ветви, каждая из которых является защищенным оператором взаимодействия. Он выполняется следующим образом. Сначала вычисляются логические выражения в защитах. Если обе защиты не пропускают, то выполнение оператора заканчивается. Если пропускает хотя бы одна, то выбирается одна из них (недетерминированно); если обе защиты блокируют, то начинается ожидание, пока одна из них не пропустит. После того как защита будет выбрана и пропустит, выполняется оператор взаимодействия C в выбранной защите. Затем выполняется соответствующий оператор S и оператор if завершается.

Оператор do выполняется аналогично. Отличие в том, что процесс выбора ветвей повторяется, пока все защиты не перестанут пропускать. Поясним это на примерах.

Вначале перепрограммируем предыдущую версию процесса Copy следующим образом.

```
process Copy {
  char c;
  do West?c -> East!c; od
}
```

Здесь оператор ввода перемещен в защиту оператора do. В защите нет логического выражения, поэтому она всегда пропускает, и, следовательно, оператор do выполняется бесконечно.

В качестве еще одного примера расширим последний процесс так, чтобы он буферизовал один или два символа, а не только один. В начальном состоянии процесс Copy должен ввести один символ из процесса West. Но, сохраняя один символ, он может либо вывести его в процесс East, либо ввести еще один из процесса West. У него нет способа узнать, какое из этих действий выбрать, поэтому он использует следующее защищенное взаимодействие.

```
process Copy {
  char c1, c2;
  West?c1;
  do West?c2 -> East!c1; c1 = c2;
  [] East!c1 -> West?c1;
  od
}
```

В начале каждой итерации цикла do в переменную c1 буферизуется один символ. Первая ветвь оператора do ожидает второй символ из процесса West, выводит первый символ в процесс East, после чего присваивает переменной c1 значение c2. Таким образом, процесс возвращается к начальному состоянию перед циклом. Вторая ветвь оператора do выводит первый символ в процесс East, после чего вводит следующий символ из процесса West. Если пропускают обе защиты оператора do, т.е. процесс West готов отправить еще один символ, а East — принять один, то может быть выбрана любая ветвь. Поскольку обе защиты всегда пропускают, цикл do никогда не завершается.

Эту программу легко обобщить до кольцевого буфера, реализовав очередь внутри процесса Copy. Например, в следующей программе может буферизоваться до 10 символов.

```
process Copy {
  char buffer[10];
  int front = 0, rear = 0, count = 0;
  do count < 10; West?buffer[rear] ->
        count = count+1; rear = (rear+1) mod 10;
  [] count > 0; East!buffer[front] ->
        count = count-1; front = (front+1) mod 10;
  od
}
```

В данной версии процесса Copy используется оператор do с двумя ветвями, но теперь в защитах есть и операторы взаимодействия, и логические выражения. Защита в первой ветви пропускает, когда в буфере есть свободное место и процесс West готов вывести символ; защита второй ветви пропускает, когда в буфере есть символ и процесс East готов ввести его. Здесь цикл do также никогда не завершается, поскольку всегда хотя бы одно из логических выражений истинно.

Еще один пример иллюстрирует использование нескольких портов в серверном процессе. Рассмотрим распределитель ресурсов (см. листинг 7.6), использующий асинхронную передачу сообщений. Процесс `Allocator` в языке CSP программируется следующим образом.

```
process Allocator {
  int avail = MAXUNITS;
  set units = начальные значения;
  int index, unitid;
  do avail > 0; Client[*]?acquire(index) ->
        avail--; remove(units, unitid);
        Client[index]!reply(unitid);
  [] Client[*]?release(index, unitid) ->
        avail++; insert(units,unitid);
  od
}
```

Этот процесс получился *намного* короче, чем процесс `Allocator` в листинге 7.6. В частности, нет необходимости выполнять слияние сообщений `acquire` и `release` в один и тот же канал; вместо этого используются разные порты и оператор `do` с отдельной ветвью для каждого порта. Кроме того, получение сообщения `acquire` можно отложить до тех пор, пока не появятся доступные элементы; тогда нет необходимости сохранять запросы. Но эти отличия возникают за счет использования защищенного взаимодействия, а не отличий между асинхронной и синхронной передачей сообщений. В действительности, как будет показано в разделе 8.3, защищенное взаимодействие можно использовать вместе с асинхронной передачей сообщений.

В качестве последнего примера возьмем два процесса, которые должны обмениваться значениями двух локальных переменных. Эта задача уже рассматривалась в конце раздела 7.5; чтобы избежать зависаний, там использовалось асимметричное решение. Теперь запрограммируем симметричный обмен с помощью защищенных операторов взаимодействия.

```
process P1 {
  int value1 = 1, value2;
  if P2!value1 -> P2?value2;
  [] P2?value2 -> P2!value1;
  fi
}

process P2 {
  int value1, value2 = 2;
  if P1!value2 -> P1?value1;
  [] P1?value1 -> P1!value2;
  fi
}
```

Здесь используется недетерминированный выбор, который возникает, когда существует несколько пар согласованных защит. Оператор взаимодействия в первой защите процесса `P1` согласуется с оператором взаимодействия второй защиты в процессе `P2`; другая пара операторов взаимодействия также согласована. Следовательно, любая пара может быть выбрана. Полученное решение является симметричным, но не настолько простым, как симметричное решение с асинхронной передачей сообщений.

7.6.3. Пример: решето Эратосфена

В упомянутой выше статье Хоара по языку CSP есть интересный пример параллельной программы для генерации простых чисел. Решение этой задачи приводится здесь и как законченный пример программы на языке CSP, и как еще один пример использования конвейера процессов для распараллеливания последовательного алгоритма.

Решето Эратосфена носит имя греческого математика, который его создал. Это классический алгоритм вычисления простых чисел в заданном промежутке. Предположим, что нам нужно породить простые числа между 2 и n. Сначала создадим список всех чисел:

```
2  3  4  5  6  7  ...  n
```

Начиная с первого незачеркнутого числа в списке, 2, просмотрим список, вычеркивая все числа, кратные 2. Если n нечетно, получим такой список:

```
2  3  ~~4~~  5  ~~6~~  7  ...  n
```

В этот момент все вычеркнутые числа не являются простыми; невычеркнутые остаются кандидатами в простые. Перейдем к следующему незачеркнутому числу в списке, 3, и повторим процесс, вычеркивая все числа, кратные трем. Если продолжать до тех пор, пока не рассмотрим все незачеркнутые числа, в списке останутся только простые числа от 2 до n. По сути, решето "вылавливает" простые числа, отсеивая числа, кратные простым.

Рассмотрим, как распараллелить этот алгоритм. Можно назначить каждому числу в списке отдельный процесс и вычеркивать числа параллельно. Но тогда возникнет две проблемы. Во-первых, процессы могут взаимодействовать только с помощью передачи сообщений, поэтому каждому процессу придется дать собственную копию всего списка чисел, а для объединения результатов понадобится отдельный процесс. Во-вторых, придется использовать слишком много процессов. В действительности достаточно столько процессов, сколько есть простых чисел, но сами они заранее неизвестны!

Можно преодолеть обе проблемы, распараллелив решето Эратосфена с помощью конвейера процессов-фильтров. Каждый фильтр получает поток чисел от своего предшественника и отправляет поток чисел преемнику. Первое число, получаемое фильтром, — это следующее по величине простое число; фильтр передает своему преемнику все числа, не кратные первому.

Программа показана в листинге 7.12. Первый процесс, `Sieve[1]`, отправляет все нечетные числа процессу `Sieve[2]`. Каждый следующий процесс получает поток чисел от своего предшественника. Первое число `p`, получаемое процессом `Sieve[i]`, является *i*-м простым. Каждый процесс `Sieve[i]` последовательно передает все получаемые числа, которые не кратны `p`. Общее количество `L` процессов `Sieve` должно быть достаточно большим, чтобы гарантировать порождение всех простых чисел от 2 до n. Например, есть 25 простых чисел, которые меньше 100; их относительное количество уменьшается с ростом значений n.[13]

Листинг 7.12. Решето Эратосфена на языке CSP

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # передать нечетные числа Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                # p является простым
  do Sieve[i-1]?next ->        # получить следующее число-кандидат
    if (next mod p) != 0 ->    # если оно может быть простым,
      Sieve[i+1]!next;         #   передать его дальше
    fi
  od
}
```

Процесс `Sieve[1]` завершается, когда заканчиваются нечетные числа для пересылки процессу `Sieve[2]`. Каждый из остальных процессов блокируется в ожидании ввода от сво-

[13] Точнее, относительное количество простых чисел, не превосходящих *n*, с ростом *n* асимптотически приближается к $1 / \ln n$. — *Прим. ред.*

его предшественника. Когда программа останавливается, значения переменных p в процессах являются простыми числами. С помощью маркера, отмечающего конец списка, несложно изменить программу, чтобы она завершалась нормально и выводила простые числа в порядке возрастания. Когда процесс видит маркер (нуль, например), он выводит свое значение p, отправляет маркер своему преемнику и завершается.

7.6.4. Язык Occam и современная версия CSP

Описанная выше нотация языка CSP широко используется в научных работах, но сам CSP никогда не был реализован. Однако были реализованы и используются несколько языков, основанных на нем. Из них наиболее известен язык Occam. Он расширяет нотацию CSP для использования глобальных каналов вместо прямого именования, обеспечивает использование процедур и функций, а также допускает вложенный параллелизм.

Хоар и другие продолжили разработку формального языка, предназначенного для описания и исследования параллельных взаимодействующих систем. Формальный язык возник под влиянием идей исходной нотации CSP и также был назван CSP, но используется не для написания прикладных программ, а для моделирования их поведения.

Далее дан обзор свойств языков Occam и современного CSP, приведены примеры, иллюстрирующие особенности каждого из них. В исторической справке в конце главы описаны источники подробной информации, включая библиотеку модулей Java для поддержки модели программирования Occam/CSP.

Occam

Язык Occam включает совсем небольшое число механизмов (название языка происходит от выражения "бритва Оккама") и имеет весьма специфический синтаксис. Первая версия языка Occam была разработана в середине 1980-х годов, версия Occam 2 появилась в 1987 г., а сейчас ведется разработка Occam 3. Хотя Occam является самостоятельным языком программирования, он разрабатывался для работы с *транспьютером*, ранней версией мультикомпьютера, и был, по существу, его машинным языком.

Базовыми элементами программы на языке Occam являются декларации и три примитивных "процесса": присваивание, ввод и вывод. Процесс присваивания — это просто оператор присваивания. Процессы ввода и вывода аналогичны командам ввода и вывода для языка CSP, но каналы имеют имена и являются глобальными по отношению к процессам. Кроме того, каждый канал должен иметь только одного отправителя и одного получателя.

Базовые процессы объединяются в обычные процессы с помощью так называемых конструкторов. Есть последовательные конструкторы, параллельный конструктор, аналогичный оператору co, и защищенный оператор взаимодействия. В синтаксисе языка Occam каждый базовый процесс, конструктор и декларация занимают отдельную строку. Декларации заканчиваются двоеточием; в записи используются отступы.

Программа на языке Occam содержит статическое число процессов и статические пути взаимодействия. Модульность (в ее простейшем варианте) обеспечивается процедурами и функциями, которые по существу являются процессами с параметрами и могут разделять только каналы и константы. Рекурсия и другие типы динамического создания и именования объектов не поддерживаются, поэтому многие алгоритмы трудно запрограммировать. Однако это же свойство языка позволяет компилятору определить число процессов в программе и способы их взаимодействия. Эта возможность в сочетании с тем, что различные конструкторы не могут разделять переменные, позволяет компилятору назначать процессы и данные процессорам такой машины с распределенной памятью, как транспьютер.

В большинстве языков программирования операторы по умолчанию выполняются последовательно; программист должен явно определять их параллельное выполнение. В языке Occam использован другой подход. Способа выполнения операторов по умолчанию нет, а вместо этого есть

два базовых конструктора: SEQ для последовательного выполнения и PAR для параллельного. Например, следующая программа последовательно увеличивает значения переменных x и y на 1.

```
INT x, y :
SEQ
  x := x + 1
  y := y + 1
```

Поскольку эти операторы обращаются к разным переменным, их можно выполнять параллельно. Для этого в приведенной программе нужно изменить SEQ на PAR. В языке Occam существует множество других конструкторов, таких как IF, CASE, WHILE и ALT (последний используется для защищенного взаимодействия), а также еще один интересный механизм, который называется *репликатором*. Он похож на квантификатор и используется аналогично.

Процессы создаются с помощью конструкторов PAR. Они взаимодействуют через каналы, доступ к которым обеспечивается базовыми процессами ввода ? и вывода !. Следующая простая последовательная программа дает эхо символов, введенных с клавиатуры, на дисплей.

```
WHILE TRUE
  BYTE ch :
  SEQ
    keyboard ? ch
    screen ! ch
```

Здесь keyboard и screen — каналы, связанные по умолчанию с периферийными устройствами (клавиатурой и экраном). (В языке Occam обеспечиваются механизмы для связывания каналов ввода-вывода с устройствами.)

В данной программе используется односимвольный буфер ch. Ее можно преобразовать в параллельную программу, которая использует двойную буферизацию с помощью двух процессов — одного для чтения с клавиатуры и второго для записи на экран. Процессы сообщаются через дополнительный канал comm; в каждом из этих процессов есть локальная переменная ch.

```
CHAN OF BYTE comm :
PAR
  WHILE TRUE      -- процесс ввода с клавиатуры
    BYTE ch :
    SEQ
      keyboard ? ch
      comm ! ch
  WHILE TRUE      -- процесс вывода на экран
    BYTE ch :
    SEQ
      comm ? ch
      display ! ch
```

Эта программа ясно демонстрирует особенности синтаксиса языка Occam. Требование размещения каждого элемента в отдельной строке приводит к длинным программам, но использование отступов делает ненужными закрывающие ключевые слова.

Конструктор ALT обеспечивает защищенное взаимодействие. Защита состоит или из процесса ввода, или логического выражения и процесса ввода, или логического выражения и конструктора SKIP. В качестве примера рассмотрим следующий процесс, буферизующий один или два символа.

```
PROC Copy(CHAN OF BYTE West, Ask, East)
  BYTE c1, c2, dummy :
  SEQ
    West ? c1
    WHILE TRUE
      ALT
        West ? c2     -- у процесса West есть байт
          SEQ
```

```
      East ! c1
      c1 := c2
  Ask ? dummy  -- процессу East нужен байт
    SEQ
      East ! c1
      West ? c1
```

Процесс Copy определен здесь в виде процедуры для иллюстрации еще одного аспекта языка Occam. Процесс Copy был бы создан с помощью вызова процедуры в конструкторе PAR и передачи ему трех каналов.

Язык Occam не допускает команды вывода в защите конструкторов ALT. Это усложняет программирование процессов, аналогичных приведенному, поскольку процесс, который выполняет ввод из канала East, должен сначала использовать канал Ask, чтобы сообщить, что ему нужен байт. Но это же ограничение существенно упрощает реализацию, которая для транспьютера была особенно важна. Поскольку в языке Occam также запрещены пустые сообщения (сигналы), некоторое значение всегда должно передаваться в канал Ask, даже если оно не используется.

Современная версия CSP

Как уже было отмечено, CSP развился из программной нотации в формальный язык для моделирования параллельных систем, в которых процессы взаимодействуют с помощью сообщений. Кроме того, CSP является набором формальных моделей и средств, которые помогают понять и проанализировать поведение систем, описанных с помощью CSP.

Современная версия языка CSP впервые была описана в 1985 г. в книге Тони Хоара. Основной целью этой книги было представление теории взаимодействующих процессов. При этом Хоар развил свою оригинальную нотацию, сделав ее более абстрактной и применимой для формального анализа. Основная идея CSP осталась той же — процессы, взаимодействующие с помощью синхронной передачи сообщений, — но CSP стал исчислением для изучения систем взаимодействующих процессов, а не языком для написания прикладных программ. Главным результатом первых работ по новой версии CSP стало построение нескольких семантических теорий: операционной, денотационной, трассировочной и алгебраической. Теперь существуют средства автоматического доказательства, позволяющие использовать CSP и его теории для моделирования практических приложений (см. историческую справку). Например, CSP использовался для моделирования протоколов взаимодействия, систем управления реального времени, протоколов безопасности и отказоустойчивых систем.

В современном CSP процесс характеризуется способом, используемым для взаимодействия с окружением (другими процессами или внешней средой). Все взаимодействия происходят в результате событий взаимодействия (*событий*). Набор взаимосвязанных процессов моделируется с помощью определения шаблона взаимодействия каждого процесса. Новая нотация CSP не императивна, а функциональна. Основными являются операторы *присоединения*, или *префиксации* (последовательного выполнения), *рекурсии* (повторения) и *защищенного выбора* (недетерминированного выбора). Существует математическая версия CSP, используемая в книгах и работах наравне с машинно-читаемой версией языка, используемой в инструментальных средствах анализа. Несколько последующих примеров, написанных в машинно-читаемой нотации, демонстрируют особенности современного CSP.

Оператор присоединения -> используется для задания последовательного порядка событий. Если red и green являются событиями, то светофор, который *один раз* включает green, а затем red, может быть задан так:

```
green -> red -> STOP
```

Последний элемент STOP — это простейший процесс CSP, который останавливается без взаимодействия.

Для описания повторения используется рекурсия. Например, следующий код задает светофор, который циклически включает green, а затем red:

```
LIGHT = green -> red -> LIGHT
```

Данный код гласит, что процесс LIGHT сначала передает green, затем red и далее повторяется. Поведение этого процесса можно задать и несколькими другими способами — например, с помощью двух взаимно рекурсивных процессов (один для передачи green, другой — red). Основное здесь — поведение, т.е. последовательность происходящих событий, а не то, как процесс "запрограммирован".

Два указанных выше события взаимодействия передавались внешнему окружению. Однако чаще нужны процессы, взаимодействующие друг с другом. Для этого язык CSP использует каналы, как в следующем примере процесса с односимвольным буфером.

```
COPY1 = West?c:char -> East!c -> COPY1
```

Программу вычисления наибольшего общего делителя (раздел 7.6) в новом CSP можно записать таким образом.

```
GCD = Input?id.x.y -> GCD(id, x, y)
GCD(id, x, y) = if (x = y) then
                  Output!id.x -> GCD
                else if (x > y) then
                  GCD(id, x-y, y)
                else
                  GCD(id, x, y-x)
```

Здесь использованы два взаимно рекурсивных процесса. Первый ждет события ввода и вызывает второй процесс. Второй процесс повторяется до выполнения условия x = y, затем выводит результат и запускает первый процесс для ожидания еще одного события ввода.

Последний пример: следующий код определяет поведение системы, буферизующей один или два символа.

```
COPY = West?c1:char -> COPY2(c1)
COPY2(c1) =   West?c2:char -> East!c1 -> COPY2(c2)
            []
              East!c1 -> West?c1:char -> COPY2(c1)
```

Второй процесс использует оператор защищенного выбора []. Защита в первой ветви ждет ввода из канала West; защита во второй ветви ждет передачи вывода в канал East. Выбор недетерминирован, если возможны оба взаимодействия. Поведение, таким образом, то же, что и у двухсимвольного процесса Copy, в котором использовано защищенное взаимодействие (раздел 7.6.2). Современный CSP также предоставляет элегантный способ определения процесса для n-элементного буфера с помощью так называемого *связанного параллельного оператора*, чтобы связать в цепочку n экземпляров параметризованной версии приведенного выше процесса COPY1.

7.7. Учебные примеры: Linda

Linda является воплощением своеобразного подхода к параллельному программированию, который синтезирует и обобщает аспекты разделяемых переменных и асинхронной передачи сообщений. Linda — это скорее не язык, а набор из шести примитивов, используемых для доступа к так называемому *пространству кортежей* (ПК) — разделяемой ассоциативной памяти, состоящей из множества отмеченных записей данных, которые называются *кортежами*. Любой последовательный язык программирования можно дополнить примитивами набора Linda и получить параллельный вариант этого языка.

Пространство кортежей похоже на единый разделяемый канал связи, за исключением того, что кортежи не упорядочены. Операция помещения кортежа OUT аналогична оператору send, операция извлечения кортежа IN — оператору receive, операция просмотра кортежа RD — оператору receive, оставляющему сообщение в канале. Операция EVAL обеспечивает создание процесса, а операции INP и RDP — неблокирующие ввод и чтение.

Хотя ПК логически разделяется процессами, его можно реализовать, распределяя его части между процессорами мультикомпьютера или сети. Таким образом, ПК можно использовать для хранения распределенных структур данных, и различные процессы могут параллельно получать доступ к разным элементам структуры данных. Как будет видно из последующих примеров, этот механизм непосредственно поддерживает схему взаимодействия процессов "портфель задач". ПК можно сделать постоянным (возобновляемым), сохраняя его содержимое после завершения программы. Это позволяет применять ПК для реализации файлов или систем баз данных.

Набор Linda появился в начале 1980-х годов. (Название имеет необычное происхождение, см. историческую справку.) В первом варианте было три примитива; остальные были добавлены позже. Несколько языков программирования, включая C и Fortran, были дополнены примитивами Linda.

7.7.1. Пространство кортежей и взаимодействие процессов

Пространство кортежей состоит из неупорядоченного множества пассивных кортежей данных и активных кортежей процессов. Кортежи данных являются отмеченными записями, которые содержат разделяемое состояние вычисления. Кортежи процессов — это процедуры, которые выполняются асинхронно. Они взаимодействуют с помощью чтения, записи и создания кортежей данных. Когда кортеж процесса завершается, он превращается в кортеж данных, как описано ниже.

Каждый кортеж данных в ПК имеет вид

```
("tag", value1, ..., valuen)
```

Метка "tag" является строкой и используется для того, чтобы различать кортежи, представляющие разные структуры данных. Значения ($value_1$, ..., $value_n$) — это нуль или несколько значений данных, например, целых чисел, действительных чисел или массивов.

Для обработки кортежей данных определены три базовых неделимых примитива `OUT`, `IN` и `RD`. Процесс помещает кортеж в ПК, выполняя такой код.

```
OUT("tag", expr1, ..., exprn);
```

Выполнение примитива `OUT` заканчивается, когда вычислены выражения и полученный кортеж данных помещен в ПК. Операция `OUT` аналогична, таким образом, оператору `send`, за исключением того, что кортеж помещается в ПК, а не в определенный канал.

Процесс извлекает кортеж данных из ПК, выполняя следующий код.

```
IN("tag", field1, ..., fieldn);
```

Каждое поле $field_i$ может быть выражением или формальным параметром вида `?var`, где `var` — имя переменной выполняемого процесса. Аргументы примитива `IN` называются *шаблоном*. Процесс, который выполняет операцию `IN`, ждет, пока в ПК не появится хотя бы один кортеж, соответствующий шаблону, и удаляет его из ПК.

Кортеж данных `d` соответствует шаблону `t`, если: 1) их метки идентичны; 2) шаблон `t` и кортеж `d` имеют одинаковое число полей и соответствующие поля имеют одинаковые типы; 3) значение каждого выражения в `t` равно соответствующему значению в кортеже данных `d`. После того как подходящий кортеж будет удален из ПК, формальным параметрам шаблона присваиваются соответствующие значения из кортежа. Таким образом, примитив `IN` аналогичен оператору `receive`, причем метка и значения в шаблоне служат для идентификации канала.

Рассмотрим два простых примера. Пусть `sem` — это символьное имя семафора. Тогда операция `V` имеет вид `OUT("sem")`, а `P` — `IN("sem")`. Значение семафора — это количество кортежей `sem` в ПК. Для моделирования массива семафоров используется дополнительное поле, представляющее индекс массива, например:

```
IN("forks", i);      # P(forks[i])
 ...
OUT("forks", i);     # V(forks[i])
```

Шаблону в операторе `IN` соответствует любой кортеж с меткой `"forks"` и таким же значением `i`.

Третий базовый примитив RD используется для просмотра кортежей. Если t — это шаблон, то выполнение оператора RD(t) приостанавливает процесс до тех пор, пока в ПК не появится соответствующий шаблону кортеж данных. Как и при использовании оператора IN, переменным в шаблоне t присваиваются значения соответствующих полей кортежа данных, но сам кортеж остается в ПК.

Существуют и неблокирующие варианты примитивов IN и RD. Операции INP и RDP являются предикатами, возвращающими значение "истина", если в ПК есть соответствующий кортеж, и "ложь", если нет. (Эти примитивы называются *условными*.) Возвращая значение "истина", они выполняют соответственно те же действия, что примитивы IN и RD.

Используя три базовых примитива, можно изменять и просматривать "глобальную" переменную. Например, можно реализовать однократно используемый барьер-счетчик, поместив его в ПК.

```
OUT("barrier", 0);
```

Достигнув точки барьерной синхронизации, процесс сначала увеличивает значение счетчика с помощью следующих операций.

```
IN("barrier", ?counter);     # получить кортеж барьера
OUT("barrier", counter+1);   # поместить новое значение
```

Затем процесс ждет, пока к барьеру не придут все n процессов:

```
RD("barrier", n);
```

Шестой и последний примитив из набора Linda — EVAL, создающий кортежи процессов. Эта операция имеет такой же вид, что и OUT, и так же создает новый кортеж.

```
EVAL("tag", expr1, ..., exprn);
```

Обычно хотя бы одно из выражений является вызовом процедуры или функции. На логическом уровне все поля кортежа вычисляются параллельно отдельными процессами. Кроме того, процесс, выполняющий оператор EVAL, не приостанавливается. На практике реализации примитивов Linda порождают новые процессы только для вычисления полей, состоящих из одного вызова функции. Остальные поля вычисляются последовательно. В любом случае после вычисления всех полей, т.е. после завершения всех порожденных процессов результаты объединяются, и кортеж становится пассивным кортежем данных. Меткой этого кортежа становится метка из EVAL, а значением каждого поля — значение соответствующего выражения.

Примитив EVAL обеспечивает средства введения параллельности в Linda-программу. (Напомним, что примитивы Linda добавляются к стандартным последовательным языкам.) В качестве простого примера рассмотрим следующий параллельный оператор.

```
co [i = 1 to n]
  a[i] = f(i);
```

Этот оператор выполняет n параллельных вызовов функции f и присваивает результаты элементам разделяемого массива a. Этому коду соответствует такой код на языке C, обогащенном примитивами Linda.

```
for (i = 1; i <= n; i++)
  EVAL("a", i, f(i));
```

Он порождает n кортежей процессов, каждый из которых вычисляет один вызов f(i). Завершаясь, кортеж процесса возвращает кортеж данных, содержащий имя массива, индекс и значение соответствующего элемента массива a.

ПК можно использовать и для реализации обычного обмена сообщениями. Например, процесс может послать сообщение в канал ch, выполнив следующий код.

```
OUT("ch", expressions);
```

Имя разделяемого объекта здесь также преобразуется в метку кортежа. Другой процесс может получить сообщение, выполняя такой код.

```
IN("ch", expressions);
```

Однако этот оператор извлекает любой подходящий кортеж. Если важно, чтобы сообщения были получены в том же порядке, в котором они отправляются, программист должен задать его. Это можно сделать, используя счетчики для нумерации сообщений и считывания их в порядке возрастания номеров.

7.7.2. Пример: генерация простых чисел с помощью портфеля задач

В разделе 7.6 было показано, как генерировать простые числа с помощью решета Эратосфена: числа проходят через конвейер процессов-фильтров. Каждый фильтр получает один простой множитель и проверяет, делит ли он остальные числа-кандидаты в простые. Если нет, кандидат может быть простым числом и передается следующему процессу-фильтру в конвейере.

Можно генерировать простые числа, используя парадигму "портфель задач", представленную в разделе 3.6. Как обычно, рабочие процессы разделяют портфель задач. В данном случае он содержит числа-кандидаты и хранится в пространстве кортежей. Управляющий процесс помещает числа-кандидаты в портфель и собирает результаты.

Каждый рабочий процесс удаляет число-кандидат из портфеля и определяет, является ли оно простым, деля его на меньшие простые числа. Но как рабочий процесс может получить все нужные ему простые числа? Выход — проверять кандидатов в строго возрастающем порядке. Тогда простые числа, необходимые для проверки нового кандидата, уже будут сгенерированы. Рабочий процесс может приостановить работу, ожидая, что другой процесс сгенерирует меньшее простое число. Но рабочие процессы не будут взаимно блокироваться, поскольку в конце концов все меньшие простые числа будут сгенерированы.

В листинге 7.13 приведена программа на языке C-Linda, которая генерирует первые `limit` простых чисел с помощью портфеля задач, хранимого в пространстве кортежей. Программа на языке C-Linda имеет ту же структуру, что и C-программа, лишь главная процедура названа `real_main`, а не `main`.

Листинг 7.13. Генерация простых числе на языке C-Linda

```
# include "linda.h"
# define LIMIT 1000      /* верхняя граница предела */

void worker() {
  int primes[LIMIT] = {2,3};  /* таблица простых чисел*/
  int numPrimes = 1, i, candidate, isprime;
  /* циклическое получение кандидатов и их проверка */
  while(true) {
    if (RDP("stop"))  /* проверка завершения */
      return;
    IN("candidate", ?candidate);  /* получить кандидат */
    OUT("candidate", candidate+2);  /* вывод следующего */
    i = 0; isprime = 1;
    while (primes[i]*primes[i] <= candidate) {
      if (candidate%primes[i] == 0) { /* не простое */
        isprime = 0; break;
      }
      i++;
      if (i > numPrimes) {  /* необходимо следующее простое */
        numPrimes++;
        RD("prime", numPrimes, ?primes[numPrimes]);
      }
    }
    /* сообщить результат управляющему процессу */
```

```
    OUT("result", candidate, isprime);
  }
}

real_main(int argc, char *argv[]) {
  int primes[LIMIT] = {2,3};  /* локальная таблица простых чисел */
  int limit, numWorkers, i, isprime;
  int numPrimes = 2, value = 5;
  limit = atoi(argv[1]);  /* считать командную строку */
  numWorkers = atoi(argv[2]);
  /* создать рабочие процессы и поместить в портфель */
  /* число 5 (это первый кандидат) */
  for (i = 1; i <= numWorkers; i++)
    EVAL("worker", worker());
  OUT("candidate", value);
  /* получить результаты от рабочих процессов в порядке возрастания */
  while (numPrimes < limit) {
    IN("result", value, ?isprime);
    if (isprime) {  /* поместить результат в таблицу и ПК */
      primes[numPrimes] = value;
      OUT("prime", numPrimes, value);
      numPrimes++;
    }
    value = value + 2;
  }
  /* завершить рабочие процессы и вывести простые числа */
  OUT("stop");
  for (i = 0; i < limit; i++)
    printf("%d\n", primes[i]);
}
```

Главная процедура сначала выполняет один процесс. Он начинает с чтения аргументов командной строки, чтобы определить, сколько нужно простых чисел и сколько процессов можно использовать. Затем используются примитив EVAL, чтобы создать рабочие процессы, и примитив OUT, чтобы поместить первое число-кандидат 5 в ПК.

Главная процедура выполняет роль управляющего процесса. Для получения результатов от рабочих процессов используется примитив IN. Результаты получаются в порядке вычисления простых чисел. Получив простое число, процедура `real_main` помещает простое число и его номер в ПК, после чего обновляет локальную таблицу. Например, 5 является третьим простым числом, 7 — четвертым и т.д.

Программа завершается, когда процедура `real_main` получает `limit` простых чисел. Управляющий процесс сообщает рабочим о необходимости остановки, помещая в ПК кортеж `stop`, и печатает простые числа из локальной таблицы.

Каждый рабочий процесс многократно получает число-кандидат из ПК, проверяет, простое ли оно, и отправляет управляющему результат, помещая его в ПК. После извлечения из ПК числа-кандидата с помощью примитива IN рабочий процесс помещает следующее нечетное число-кандидат в ПК. Таким образом, в ПК в любой момент времени может быть не более одного кандидата, и кандидаты являются возрастающими нечетными числами.

Каждый рабочий процесс поддерживает локальную таблицу простых множителей. Он дополняет эту таблицу, если для проверки числа-кандидата ему нужно еще одно простое число. Рабочий процесс получает простое число, считывая его из ПК. Чтобы избежать взаимных блокировок, рабочие процессы считывают простые числа в строго возрастающем порядке.

На каждой итерации рабочий процесс проверяет, не нужно ли остановиться, с помощью условного примитива чтения RDP. Этот примитив возвращает значение “истина”, если в ПК есть кортеж `stop`, и значение “ложь”, если нет. Поскольку рабочие процессы не увидят кор-

теж stop, пока управляющий не получит limit простых чисел, будет проверено больше чисел-кандидатов, чем необходимо. Этого трудно избежать без существенного увеличения числа сообщений, передаваемых между процессами.

Программа в листинге 7.13 — не самая эффективная. Каждое число-кандидат помещается в ПК и извлекается оттуда дважды, один раз в кортеже candidate, второй — в кортеже result. Доступ к ПК намного менее эффективен, чем к локальным переменным. Чтобы повысить степень использования локальных переменных, можно заставить управляющий процесс помещать в ПК сразу целую группу чисел-кандидатов, например из 10 или 20 чисел. Рабочие процессы могут вычислять все простые числа из данной группы и отправлять общий результат диспетчеру. Можно также вначале помещать в таблицы простых чисел управляющего и рабочих процессов больше первых простых чисел.

7.8. Учебные примеры: библиотека MPI

Интерфейс передачи сообщений (Message Passing Interface — MPI) — это библиотека процедур для передачи сообщений. При использовании библиотеки MPI процессы распределенной программы записываются на таком последовательном языке, как C или Fortran; их взаимодействие и синхронизация задается с помощью вызовов процедур библиотеки MPI.

Интерфейс программирования MPI-приложений был создан в середине 1990-х годов большой группой разработчиков из университетов, правительственных и промышленных организаций. Он отражает опыт работы с такими библиотеками передачи сообщений, как PVM. Цель этой группы состояла в разработке единой библиотеки, которую можно было бы эффективно реализовать на различных многопроцессорных машинах. Теперь библиотека MPI стала стандартом де-факто; существует несколько ее реализаций.

Программы, использующие библиотеку MPI, имеют так называемый SPMD-стиль (single program, multiple data — одна программа, много данных). Каждый процессор выполняет копию одной и той же программы. (В написанном на языке C приложении каждый процессор начинает выполнять процедуру main.) Каждый экземпляр программы может определить собственный идентификатор и, следовательно, предпринять различные действия. Экземпляры программы взаимодействуют, вызывая функции библиотеки MPI, которые поддерживают взаимодействие процессов с процессами, группами и окружением.

Ниже представлена простая законченная программа, иллюстрирующая межпроцессную передачу сообщений. Далее описаны функции MPI для связи с группами. (Больший пример программы с использованием библиотеки MPI приведен в листинге 12.2.)

7.8.1. Базовые функции

Напомним задачу, представленную в разделе 7.4, в которой два процесса обменивались значениями локальных переменных. В листинге 7.14 приведена законченная C-программа, использующая для решения этой задачи библиотеку MPI.

Листинг 7.14. Программа обмена значениями между двумя процессами с помощью библиотеки MPI

```
# include <mpi.h>
main(int argc, char *argv[]) {
  int myid, otherid, size;
  int length = 1, tag = 1;
  int myvalue, othervalue;
  MPI_Status status;
  /* инициализировать MPI и получить свой id (ранг) */
  MPI_Init(&argc, &argv);
```

```
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid == 0) {
    otherid = 1; myvalue = 14;
  } else {
    otherid = 0; myvalue = 25;
  }
  MPI_Send(&myvalue, length, MPI_INT, otherid,
           tag, MPI_COMM_WORLD);
  MPI_Recv(&othervalue, length, MPI_INT, MPI_ANY_SOURCE,
           tag, MPI_COMM_WORLD, &status);
  printf("процесс номер %d получил %d\n", myid, othervalue);
  MPI_Finalize();
}
```

Первая строка содержит директиву включения заголовочного файла библиотеки MPI, в котором определены типы, константы и прототипы функций. Программа компилируется, компонуется с библиотекой MPI и затем выполняется с помощью команды (например `mpirun`), которую обеспечивает конкретная реализация библиотеки MPI. Эта команда создает определенное число процессов, и каждый из них начинает выполнять процедуру `main`.

В программе есть вызовы шести основных процедур библиотеки MPI; они присутствуют практически во всех программах, использующих эту библиотеку.

`MPI_Init`. Инициализировать библиотеку MPI и возвратить копию аргументов командной строки, передаваемых программе. Копию получают все экземпляры программы. В результате этого вызова инициализируется переменная `MPI_COMM_WORLD`, которая является набором запущенных процессов.

`MPI_Comm_size`. Определить число процессов (`size`), которые нужно запустить. Для данной программы их должно быть два.

`MPI_Comm_rank`. Определить ранг (идентификатор) процесса. Рангами являются числа от `0` до `size-1`.

`MPI_Send`. Отправить сообщение другому процессу. Аргументами этой процедуры являются: буфер, в котором находится сообщение, число элементов для передачи, тип данных в сообщении, идентификатор (ранг) процесса назначения, метка, определяемая программистом для того, чтобы различать типы сообщений, и значение `MPI_COMM_WORLD`.

`MPI_Receive`. Получить сообщение от другого процесса. Аргументы: буфер, в который должно быть помещено сообщение, число элементов в сообщении, тип данных в сообщении, идентификатор передающего процесса или значение "безразлично" (`MPI_ANY_SOURCE`), метка сообщения, группа взаимодействия, статус возврата.

`MPI_Finalize`. "Сбросить" библиотеку MPI и завершить данный процесс.

В приведенной программе процесс `0` отправляет значение `14` процессу `1`, а тот отправляет значение `25` процессу `0`. Обмен кодируется симметричным образом, как отправка с последующим приемом. Если запущено больше двух копий программы, то все процессы с номерами от `1` до `size-1` отошлют число `25` процессу `0`. Процесс `0` получит одну копию значения `25` от одного из остальных процессов, но свое значение отошлет только процессу `1`.

Процедура `MPI_Send` приостанавливает работу программы, пока сообщение не будет буферизовано или получено указанным процессом назначения (с идентификатором `otherid`). Таким образом, процедуры `MPI_Send` и `MPI_Receive` вместе обеспечивают асинхронную передачу сообщений, хотя технически реализация библиотеки MPI не обязана буферизовать сообщения. (Это приводит к проблемам при написании программ, использующих библиотеку MPI, поскольку становятся возможными взаимоблокировки.) Библиотека MPI поддержи-

вает и другие режимы передачи и приема сообщений. Например, программист может выделить память для буфера и тем самым обеспечить неблокирующую отправку. Другие функции включают синхронную передачу и опрос (неблокирующий прием) сообщений.

7.8.2. Глобальное взаимодействие и синхронизация

В библиотеке MPI есть различные функции для глобального взаимодействия и синхронизации. Они дают возможность каждому члену группы процессов непосредственно связываться со всеми членами этой группы. Следующие функции представляют наибольший интерес и используются чаще всего.

`MPI_Barrier`. Сигнализировать прибытие к барьеру. Возврат из нее происходит, когда эту функцию вызовут все процессы группы. (По умолчанию определена группа `MPI_CAMM_WORLD`.)

`MPI_Bcast`. Разослать копии сообщения всем членам группы (включая отправителя).

`MPI_Scatter`. Распределить массив `a`, состоящий из `size` элементов, отправляя каждому процессу `i` в группе сообщение со значением `a[i]`.

`MPI_Gather`. Собрать сообщения от процессов группы и записать их в массив из `size` элементов. Сообщение от процесса `i` записывается в `i`-й элемент массива.

`MPI_Reduce`. Собрать значения в сообщениях от каждого процесса и свести их к одному значению. Операторами сведения являются `MPI_SUM`, `MPI_MAX` и другие ассоциативные и коммутативные бинарные операторы.

`MPI_Allreduce`. То же, что и `MPI_Reduce`, только *каждый* процесс получает копию полученного значения.

Возможности, обеспечиваемые этими процедурами, можно запрограммировать явно, используя межпроцессную передачу сообщений. Например, рассылку копий сообщения можно запрограммировать с помощью цикла, в котором сообщение отправляется каждому процессу. Однако, пользуясь этими процедурами, многие приложения написать намного легче.

Рассмотрим, например, программы в листингах 7.9 и 7.10. В управляющем процессе `P[0]` (см. листинг 7.9) можно было бы использовать функцию `MPI_Gather` для сбора сообщений от всех остальных процессов, а функцию `MPI_Bcast` — для рассылки результатов. Симметричные процессы в листинге 7.10 можно перепрограммировать, чтобы использовать всего два вызова функции `MPI_Allreduce`! Первый вызов функции должен свести все локальные значения к наименьшему, второй — к наибольшему. В другом примере (листинг 12.2) иллюстрируется использование функции `MPI_Reduce`.

Дополнительное преимущество процедур глобального взаимодействия состоит в том, что система MPI может реализовывать их более эффективно, чем программист. Во-первых, будет происходить намного меньше переключений контекста, поскольку прикладной процесс совершает один вызов функции и блокируется до его завершения. Во-вторых, можно уменьшить объем буферного пространства. Наконец, у системы MPI будет больше возможностей перемежать передачи сообщений и внутреннюю обработку, а также использовать параллелизм, доступный в сети взаимодействий.

7.9. Учебные примеры: язык Java

Язык Java поддерживает параллельное программирование с помощью потоков, разделяемых переменных и синхронизированных методов (см. раздел 5.4). Язык Java можно использовать и для написания распределенных программ. В нем нет встроенных примитивов для передачи

сообщений, но есть стандартный модуль `java.net`. Классы этого модуля поддерживают низкоуровневое взаимодействие, используя дейтаграммы, связь более высокого уровня с помощью сокетов и взаимодействие через Internet с помощью адресов URL (uniform resource locator — унифицированный указатель информационного ресурса). Далее дано краткое введение в сети и сокеты, а затем представлен полный пример процесса для удаленного чтения файла и его клиента.

7.9.1. Сети и сокеты

Сеть состоит из набора узлов (ведущих компьютеров — host), связанных с помощью коммуникационного пространства. Например, локальная сеть может состоять из серверов и рабочих станций, связанных с помощью Ethernet. Internet — это пример глобальной сети, содержащей миллионы узлов и тысячи взаимосвязанных сетей. Каждый узел имеет уникальное имя, поэтому другие узлы могут связываться с ним. Фактически узлы сети Internet имеют по два уникальных имени: символьное имя домена Internet и числовой адрес для протокола IP (Internet protocol). Например, данная книга записана на узле `paloverde.cs.arizona.edu`, а его IP-адрес — `192.12.69.16`.

Программы, выполняемые на разных узлах, используют *протоколы взаимодействия*. Существует множество таких протоколов. Они поддерживают различные типы данных (от текста и графики до фильмов), различные уровни абстракции (такие как передача файлов или пакетов данных) и различные соотношения свойств, например, скорости и надежности. Системные программисты обычно используют два протокола — TCP (Transmission Control Protocol — протокол управления передачей) и UDP (User Data Protocol — протокол пользовательских данных). Эти протоколы обычно реализованы на основе протокола IP (Internet Protocol), который, в свою очередь, основан на аппаратных протоколах, таких как драйвер Ethernet. С другой стороны, на основе протокола TCP реализованы высокоуровневые протоколы передачи файлов (file transfer protocol — FTP) и передачи гипертекстов (hypertext transfer protocol — HTTP).

Взаимодействие по протоколу TCP основано на соединениях и потоках. Соединение — это связь между двумя узлами, которая устанавливается перед процессом взаимодействия и существует до его завершения. Соединение можно представить как "программный провод" между узлами. Поток взаимодействия — это последовательность сообщений, передаваемых по соединению. Поток обеспечивает надежное и упорядоченное взаимодействие, т.е. сообщения не теряются и доставляются в порядке их отправки. Таким образом, семантически поток совпадает с каналом (раздел 7.1).

Протокол UDP основан на дейтаграммах. Дейтаграмма состоит из адреса назначения и небольшого "пакета" данных, который является просто массивом байтов. Соединения не нужны, поскольку адрес назначения есть в каждой дейтаграмме. Связь по протоколу UDP не надежна, поскольку дейтаграммы могут теряться или приходить не по порядку. С другой стороны, связь с помощью дейтаграмм намного быстрее, чем связь, основанная на потоках. Таким образом, альтернативность протоколов TCP и UDP связана в основном с выбором между надежностью и скоростью.

Модуль `java.net` содержит несколько классов, которые поддерживают взаимодействие с помощью как дейтаграмм, так и потоков. Далее иллюстрируется использование двух классов, `Socket` и `ServerSocket`, широко используемых в приложениях типа "клиент-сервер".

7.9.2. Пример: удаленное чтение файла

В состав многих вычислительных средств входят рабочие станции, которые используются отдельными людьми и разделяемыми серверами для хранения файлов, доступа к Internet и т.п. Предположим, что клиентские приложения выполняются на рабочих станциях, связанных с машиной, на которой установлен файловый сервер (серверная машина). Он обеспечивает доступ к файловой системе, хранимой на локальных дисках. (Будем предполагать, что серверная машина работает под управлением операционной системы Unix и использует ее

файловую систему.) Когда клиент хочет прочитать файл, который хранится на серверной машине, он связывается с выполняемым на ней серверным процессом. Клиент отправляет серверу имя файла, который он хочет прочитать. При условии, что файл может быть открыт, сервер считывает строки файла и отправляет их клиенту.

В листинге 7.15 приведен код на языке Java для процесса-сервера. Сервер сначала создает серверный сокет и ждет соединения по порту 9999. (Порт — это уникальный номер, используемый клиентами и серверами для того, чтобы различить конкретное соединение.) Затем сервер ждет связи с клиентом. Далее сервер создает входной и выходной потоки, использующие сокет. Получив от клиента имя файла, сервер открывает указанный файл. Если этого файла нет, сервер отправляет клиенту сообщение об ошибке, закрывает потоки и сокет и ждет следующего подключения клиента. Если файл существует, сервер считывает строки файла и отправляет их клиенту (используя метод println). При достижении конца файла сервер закрывает потоки и сокет и ждет следующего подключения клиента. Если возникает исключительная ситуация, сервер выводит сообщение об ошибке и завершается.

Листинг 7.15. Сервер чтения файлов на языке Java

```
// Читать файл и передать его клиенту
import java.io.*; import java.net.*;

public class FileReaderServer {
 public static void main(String args[]) {
  try {
   // создать серверный сокет и
   // ждать подключения по порту 9999
   ServerSocket listen = new ServerSocket(9999);
   while (true) {
     System.out.println("жду соединения");
     Socket socket = listen.accept();  // ждать клиента
    // создать входной и выходной потоки для общения с клиентом
     BufferedReader from_client =
       new BufferedReader(new InputStreamReader
         (socket.getInputStream()));
     PrintWriter to_client = new PrintWriter
         (socket.getOutputStream());
    // получить от клиента имя файла и проверить его существование
     String filename = from_client.readLine();
     File inputFile = new File(filename);
     if (!inputFile.exists()) {
       to_client.println("не могу открыть " + filename);
       to_client.close(); from_client.close();
       socket.close();
       continue;
     }
     // читать строки файла и отправлять клиенту
     System.out.println("reading from file " + filename);
     BufferedReader input =
       new BufferedReader(new FileReader(inputFile));
     String line;
     while ((line = input.readLine()) != null)
       to_client.println(line);
     to_client.close(); from_client.close();
     socket.close();
  }}
  catch (Exception e) // сообщить о любом исключении
    { System.err.println(e); }
}}
```

Листинг 7.16 содержит код клиента на языке Java. После считывания имен узла и файла из командной строки клиент пытается открыть соединение через сокет с узлом по порту 9999 — обусловленному номеру порта сервера. Если сервер может принять соединение, клиент отправляет имя файла filename серверу, получает строки файла и выводит их на дисплей. Если возникает исключительная ситуация, клиент выводит сообщение об ошибке и завершает работу.

Листинг 7.16. Клиент чтения файлов на языке Java

```
// Получить файл от RemoteFileServer и вывести его в stdout
import java.io.*; import java.net.*;

public class Client {
 public static void main(String[] args) {
  try {
   // прочитать аргументы командной строки
   if (args.length != 2) {
     System.out.println("нужны два аргумента");
     System.exit(1);
   }
   String host = args[0];
   String filename = args[1];
   // открыть сокет, а затем входной и выходной потоки для него
   Socket socket = new Socket(host,9999);
   BufferedReader from_server =
     new BufferedReader(new InputStreamReader
       (socket.getInputStream()));
   PrintWriter to_server = new PrintWriter
       (socket.getOutputStream());
   // отправить серверу имя файла, а затем читать и печатать строки
   // до закрытия соединения сервером
   to_server.println(filename); to_server.flush();
   String line;
   while ((line = from_server.readLine()) != null) {
     System.out.println(line);
   }
  }
  catch (Exception e)    // сообщать о любом исключении
    { System.err.println(e); }
}}
```

Эти две программы выполняются следующим образом. Сначала каждая компилируется на своем узле с помощью Java-компилятора, например javac. Затем командой

```
java FileReaderServer
```

запускается сервер. После этого на другом (или этом же) узле командой

```
java Client hostname filename
```

запускается клиент (hostname — это имя узла, на котором выполняется сервер, а filename — имя файла, который хочет прочитать клиент). Сервер будет выполняться до его уничтожения или возникновения исключительной ситуации, так что в дальнейшем можно запустить дополнительные программы-клиенты. Читателю полезно поэкспериментировать с этими программами и посмотреть, как они ведут себя в различных ситуациях. Например, что случится, если клиент запустить перед сервером? Что будет, если два клиента попытаются подключиться к серверу почти одновременно?

Историческая справка

Идея передачи сообщений возникла в конце 1960-х годов, когда еще не было мультипроцессорных машин общего назначения и компьютерных сетей. Однако некоторые разработчики операционных систем уже тогда понимали, что организовывать операционную систему нужно в виде взаимодействующих процессов, поскольку каждый процесс выполняет определенную функцию и не влияет на другие процессы, не разделяя с ними переменные.

Многозадачное ядро, разработанное Пером Бринчем Хансеном (Per Brinch Hansen) [1970] для вычислительной машины RC 4000, было, по мнению автора, наиболее элегантным среди первых разработок с обменом сообщениями. Это ядро обеспечивало четыре примитива для поддержки взаимодействия типа "клиент-сервер", использующего разделяемую область буферов фиксированной длины. Позже он добавил два примитива для просмотра процессом своей очереди сообщений и буферов ответа, а также для получения определенных сообщений или ответов. В результате процесс мог вести несколько диалогов одновременно. Примитивы Бринча Хансена описаны в учебнике по операционным системам [Bic and Show, 1988, pp. 24–26].

Взаимодействие процессов с помощью разделяемых переменных более эффективно, чем с помощью передачи сообщений, поэтому большинство операционных систем однопроцессорных машин и мультипроцессоров с разделяемой памятью использует разделяемые переменные. Но для поддержки взаимодействия процессов на разных машинах сети эти же операционные системы поддерживают и передачу сообщений. Например, операционная система UNIX поддерживает сокеты и различные системные вызовы для отправки и приема сообщений. Операционные системы для мультипроцессоров с распределенной памятью по необходимости обеспечивают и используют примитивы для передачи сообщений. Обзоры аппаратного и программного обеспечения мультикомпьютеров можно найти в книгах [Almasi and Gottlieb, 1994] и [Hwang, 1993].

Многие языки программирования основаны на передаче сообщений или включают средства для этого. В данной книге описаны языки CSP и Occam (раздел 7.6), набор примитивов Linda (раздел 7.7), языки Java (раздел 7.9) и SR (раздел 8.7). В прекрасной обзорной статье [Bal, Steiner and Tanenbaum, 1989] описаны и классифицированы очень многие языки программирования для распределенных вычислений, включая несколько языков, основанных на асинхронной и синхронной передаче сообщений.

Дуальность мониторов и передачи сообщений была отмечена и проанализирована в работе [Lauer and Needham, 1978]. Страуструп [Stroustrup, 1982], создатель языка C++, исследовал эти два стиля программирования и обнаружил, что на машинах с разделяемой памятью в операциях типа "клиент-сервер" их производительность примерно одинакова, хотя иногда передача сообщений работает медленнее. С другой стороны, мониторы и передача сообщений обычно используются совершенно по-разному, что иллюстрируется примерами из глав 5 и 7. Кроме того, мониторы непосредственно не поддерживают фильтры или взаимодействующие равные, обеспечивая для этого лишь реализацию буферов взаимодействия.

Синхронное взаимодействие было представлено Хоаром [Hoare, 1978] в его классической работе о взаимодействующих последовательных процессах. Эта и его же более ранняя работа по мониторам [Hoare, 1974] являются образцом ясности и доходчивости. Механизмы, описанные в разделе 7.6, по сути, те же, что были представлены в этой работе. Основное отличие в том, что здесь допускаются операторы вывода в защите. Однако и сам Хоар признал их полезность (в разделе 7.8 своей работы), а в его более поздней книге [Hoare, 1985] по семантике CSP они уже допускаются.

Язык Occam — самый известный язык программирования, основанный на синхронной передаче сообщений. Он был разработан в начале 1980-х годов Дэвидом Мэем (David May) и его коллегами из британской компьютерной фирмы INMOS (эта компания разработала и транспьютер). Консультантом группы Мэя был Хоар, создатель CSP. В статье [May, 1983] дан обзор языка, который теперь называется Occam 1. Язык Occam 2 появился в 1987 г. В [Burns, 1988] приведен полный обзор Occam 2 и транспьютеров, а также сравнение языков Occam и Ada. Отличный источник информации по языку Occam и его реализациям — архив "Internet Parallel Computing Archive" по адресу `www.hensa.ac.uk/parallel`.

Современная версия языка CSP была представлена Хоаром в книге [Hoare, 1985], основная тема которой — семантика взаимодействующих последовательных процессов. Хоар и несколько его коллег продолжили разработку теории и нотации. Кроме того, они разработали (и продолжают совершенствовать) автоматизированные инструменты доказательства для CSP, программы имитации и графические средства, позволяющие экспериментировать со спецификациями CSP. Роско [Roscoe, 1998] описал последнюю версию CSP и теорию, на которой он основан, практические приложения и средства. На домашних страницах книги Роско есть обширная информация, а основной архив по CSP находится по адресам

```
www.comlab.ox.ac.uk/oucl/publications/books/concurrency/
www.comlab.ox.ac.uk/archive/csp.html
```

Группы разработчиков создали и библиотеки классов языка Java для поддержки модели CSP. Ссылки на них можно найти в указанном выше архиве "Internet Parallel Computing Archive" или в архиве CSP.

Набор примитивов Linda был представлен в 1983 году Гелернтером (David Gelernter) в докторской диссертации в государственном университете New York Stony Brook. Название возникло из студенческого юмора Гелернтера. Язык Ada был разработан тогда же и назван по имени Ады Августы Лавлейс (Ada Augusta Lovelace), которая считается первым программистом. Гелернтер в шутку решил назвать язык по имени другой Лавлейс — популярной в то время актрисы из фильмов категории "для взрослых". Название прилипло.

Примитивы Linda представлены в статье [Gelernter, 1985], а их использование для реализации распределенных структур данных и парадигмы программирования "портфель задач" — в статье [Carriero, Gelernter and Leichter, 1986]. В работе [Carriero and Gelernter, 1986] подробно описана реализация примитивов Linda, а в [Carriero and Gelernter, 1989a] примитивы Linda сравниваются с передачей сообщений, параллельными объектами, параллельным логическим и функциональным программированием. Библиография к этой статье содержит несколько ссылок на описания специальных приложений набора Linda. В обзоре [Carriero and Gelernter, 1989b] эти же авторы описали три вида параллелизма — результирующий (result), последовательный (agenda) и специальный (specialist) — и показали, как каждый из них реализуется примитивами Linda. Основой программы генерации простых чисел с помощью примитивов Linda (см. листинг 7.13) послужила программа из этой статьи.

Примитивы Linda были добавлены в несколько последовательных языков программирования, а их реализации существуют для нескольких рабочих станций, мультипроцессоров и мультикомпьютеров. Библиотека Linda распространяется организацией Scientific Computing Associates, `www.sca.com`.

Интерфейс MPI был определен в середине 1990-х годов [Snir et al., 1996]. Сейчас существует несколько его бесплатных реализаций. Автор использовал реализации LAM и MPICH; их домашние страницы находятся по адресам

```
www.mpi.nd.edu/lam/
www-unix.mcs.anl.gov/mpi/mpich/
```

На этих страницах есть обширная документация, ссылки на другие узлы и исходные коды.

Часто используется еще одна библиотека передачи сообщений — PVM (Parallel Virtual Machine — параллельная виртуальная машина) [Geist et al., 1994]. Опыт работы с библиотекой PVM существенно повлиял на разработку интерфейса MPI, созданного позже. Виды примитивов передачи сообщений в этих двух библиотеках аналогичны. Основное различие библиотек PVM и MPI состоит в том, что PVM поддерживает сетевой набор гетерогенных машин и имеет средства обеспечения отказоустойчивости. Для этого в библиотеке PVM используются процессы-демоны, которые выполняются в фоновом режиме на каждом узле. Демоны управляют узлами и координируют взаимодействие прикладных процессов. Домашняя страничка библиотеки PVM находится по адресу

```
www.epm.ornl.gov/pvm/pvm_home.html
```

На этой страничке есть много информации и программного обеспечения.

Источники информации по модулю `java.net` языка Java, использованные в данном тексте, — это две книги Фланагана [Flanagan, 1997a, 1997b]. Исходные коды примеров из книги [Flanagan, 1997b] доступны на Web-узле `www.ora.com/catalog/jenut`. На этом узле есть информация по некоторым другим источникам о Java, включая книги по потокам, сетевому программированию и распределенным вычислениям.

Другой подход к распределенному программированию на языке Java взят из книг [Hartley, 1998] и [Magee and Kramer, 1999]. В обеих книгах описано, как реализовать каналы, синхронную и асинхронную передачи сообщений в виде классов языка Java (практически так же, как и в данном тексте). Хартли, например, показал, как на языке Java запрограммировать многие алгоритмы из этой книги. Домашняя страничка этой книги находится по адресу

`www.mcs.drexel.edu/~shartley/ConcProgJava/`

Там же указан источник Java-программ этого автора.

Литература

Almasi, G. S., and A. Gottlieb. 1994. *Highly Parallel Computing*, 2nd ed. Menlo Park, CA: Benjamin/Cummings.

Bal, H. E., J. G. Steiner, and A. S. Tanenbaum. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys* 21, 3 (September): 261–322.

Bic, L., and A. C. Shaw. 1988. *The Logical Design of Operating Systems*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall.

Brinch Hansen, P. 1970. The nucleus of a multiprogramming system. *Comm. ACM* 13, 4 (April): 238–241.

Burns, A. 1988. *Programming in occam 2*. Reading, MA: Addison-Wesley.

Carriero, N., and D. Gelernter. 1986. The S/Net's Linda kernel. *ACM Trans. Computer Systems* 4, 2 (May): 110–129.

Carriero, N., and D. Gelernter. 1989a. Linda in Context. *Comm. ACM* 32, 4 (April): 444–458.

Carriero, N., and D. Gelernter. 1989b. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys* 21, 3 (September): 323–358.

Carriero, N., D. Gelernter, and J. Leichter. 1986. Distributed data structures in Linda. *Thirteenth ACM Symp. on Principles of Prog. Langs.*, January; 236–242.

Flanagan, D. 1997a. *Java in a Nutshell: A Desktop Quick Reference*, 2nd ed. Sebastopol, CA: O'Reilly & Associates.

Flanagan, D. 1997b. *Java Examples in a Nutshell: A Tutorial Companion to Java in a Nutshell.* Sebastopol, CA: O'Reilly & Associates.

Geist, A., A. Beguelin, J. Dongarra, W. Wiang, R. Manchek, and V. Sunderam. 1994. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: MIT Press.

Gelernter, D. 1985. Generative communication in Linda. *ACM Trans. on Prog. Languages and Systems* 7, 1 (January): 80–112.

Hartley, S. J. 1998. *Concurrent Programming: The Java Programming Language*. New York: Oxford University Press.

Hoare, C. A. R. 1974. Monitors: An operating system structuring concept. *Comm. ACM* 17, 10 (October): 549–557.

Hoare, C. A. R. 1978. Communicating sequential processes. *Comm. ACM* 21, 8 (August): 666–677.

Hoare, C. A. R. 1985. *Communicating Sequential Processes.* Englewood Cliffs, NJ: Prentice-Hall International. (Ч. Хоар. Взаимодействующие последовательные процессы. — М.: Мир, 1989.)

Hwang, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* New York: McGraw-Hill.

Lauer, H. C., and R. M. Needham. 1978. On the duality of operating system structures. *Proc. Second Int. Symp. on Operating Systems*, October; reprinted in *Operating Systems Review* 13, 2 (April 1979): 3–19.

Magee, J, and J. Kramer. 1999. *Concurrency: State Models and Java Programs.* New York: Wiley.

May, D. 1983. OCCAM. *SIGPLAN Notices* 18, 4 (April): 69–79.

Roscoe, A. W. 1998. *The Theory and Practice of Concurrency.* Englewood Cliffs, NJ: Prentice-Hall International.

Snir, M., S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. 1996. *MPI: The Complete Reference.* Cambridge, MA: MIT Press.

Stroustrup, B. 1982. An experiment with the interchangeability of processes and monitors. *Software — Practice and Experience* 12, 1011–1025.

Упражнения

7.1. Пусть в сортирующей сети (см. рис. 7.1) процессы слияния объявлены как массив `Merge[1:n]`. Напишите объявления каналов и тело процессов массива `Merge`. Поскольку процессы должны быть идентичными, вам нужно представлять себе, как дерево слияний накладывается на массив.

7.2. Рассмотрим следующий процесс-фильтр `Partition`: он получает неотсортированные значения из входного канала `in` и отправляет их в один из двух выходных каналов `out1` и `out2`. Чтобы разделить получаемые значения на два множества, процесс `Partition` использует первое полученное значение `v`. Все значения, которые меньше или равны `v`, он отправляет в `out1`, а остальные — в `out2`. В конце работы процесс `Partition` отправляет в `out1` значение `v` и маркер `EOS` в оба канала. Конец входного потока также указывается маркером `EOS`:

   а) разработайте реализацию процесса `Partition`. Сначала определите предикаты, описывающие содержимое каналов, а затем напишите тело процесса `Partition`;

   б) покажите, как из процессов `Partition` собрать сортирующую сеть. Можно предполагать, что вводятся `n` значений, и `n` является степенью 2. Каким будет размер сортирующей сети в худшем случае? (Это не лучший способ сортировки, а всего лишь упражнение!)

7.3. Массив из `n` значений можно отсортировать, используя конвейер из `n` процессов-фильтров. Первый процесс вводит сразу все значения, сохраняет минимальное из них и передает остальные следующему процессу. Каждый фильтр делает то же самое: получает поток значений от предыдущего процесса, сохраняет наименьшее и передает остальные значения следующему. Предположим, что каждый процесс имеет локальные переменные для хранения всего двух значений: следующего вводимого и минимального среди уже введенных:

   а) разработайте код для этих процессов-фильтров. Объявите каналы и используйте асинхронную передачу сообщений;

   б) сравните производительность вашей программы, полученной при решении пункта *а*, и сортирующей сети из процессов `Merge` (см. рис. 7.1). Сколько сообщений должна отправить каждая программа для сортировки `n` значений? Какие сообщения можно отправлять параллельно (при наличии соответствующего аппаратного обеспечения), а какие должны быть отправлены последовательно? Определите время выполнения каждой программы, выраженное в виде длины наибольшей цепочки последовательно упорядоченных сообщений.

7.4. Рассмотрим программу в листинге 7.6. Следующие пункты решите как независимые упражнения:

а) измените программу так, чтобы клиенты могли запрашивать и освобождать за один раз больше, чем один элемент;

б) программа показывает, как моделировать специальный монитор (см. листинг 7.5), используя серверный процесс. Разработайте программу для моделирования произвольного монитора. Для этого покажите, как моделируется каждый механизм монитора (см. раздел 5.1), включая примитивы `signal_all`, приоритетный `wait` и `empty`. Укажите действия, выполняемые клиентами и сервером.

7.5. Рассмотрите задачу об обедающих философах (см. раздел 4.3). Разработайте серверный процесс для синхронизации действий философов. Процессы должны взаимодействовать с помощью асинхронной передачи сообщений.

7.6. Рассмотрите задачу о читателях и писателях (см. раздел 4.4). Постройте серверный процесс для реализации базы данных. Представьте серверные интерфейсы процесса-читателя и процесса-писателя. Процессы должны взаимодействовать с помощью асинхронной передачи сообщений.

7.7. Разработайте реализацию процесса-сервера времени. Он должен поддерживать две операции, вызываемые клиентскими процессами: одну для получения времени и вторую для приостановки на заданный интервал времени. Кроме того, сервер времени должен периодически получать сообщения-"тики" от обработчика прерываний таймера. Представьте клиентский интерфейс сервера времени для выполнения операций получения текущего времени и приостановки.

7.8. *Задача о счете.* Несколько людей (процессов) используют общий счет. Каждый из них может помещать средства на счет и снимать их. Текущий баланс равен сумме всех вложенных средств минус сумма всех снятых. Баланс никогда не должен становиться отрицательным. Разработайте сервер для решения этой задачи, представьте клиентский интерфейс для этого сервера. Клиенты делают запросы двух типов: поместить на счет `amount` долларов и снять `amount` долларов (значение `amount` положительно). При изъятии средств возможна задержка, пока на счету не будет достаточной суммы.

7.9. В комнату входят процессы двух типов `A` и `B`. Процесс типа `A` не может выйти, пока не встретит два процесса `B`, а процесс `B` не может выйти, пока не встретит один процесс `A`. Как только процесс встречает необходимое число процессов другого типа, он выходит из комнаты:

а) разработайте серверный процесс для реализации такой синхронизации. Представьте серверный интерфейс процессов типа `A` и `B`;

б) измените ответ к пункту *а* так, чтобы первый из двух процессов `B`, встретившихся с процессом `A`, не выходил из комнаты, пока процесс `A` не встретит второй процесс `B`.

7.10. Предположим, что в компьютерном центре есть два принтера, `A` и `B`, которые похожи, но не одинаковы. Есть три типа процессов, использующих принтеры: только типа `A`, только типа `B`, обоих типов.

Разработайте код клиента каждого типа для получения и освобождения принтера. Ваше решение должно быть справедливым при условии, что принтеры в конце концов освобождаются.

7.11. *Узкий мост.* На узкий мост едут машины с севера и с юга. Машины, едущие в одном направлении, могут переезжать мост одновременно, а в противоположных — нет:

а) разработайте серверный процесс, управляющий использованием моста. Машины считайте клиентскими процессами. Для взаимодействия процессов используйте асинхронную передачу сообщений и покажите, как клиенты взаимодействуют с сервером;

б) напишите программу имитации вашего ответа к пункту *а*. Используйте язык C с библиотекой MPI или язык Java с модулем сокетов. Пусть машины многократно

пытаются проехать через мост. Они должны находиться на мосту и ждать перед повторным переездом через него в течение случайных отрезков времени. Напечатайте трассу основных событий при имитации.

7.12. *Американские горки.* Есть n процессов-пассажиров и один процесс-вагончик. Пассажиры ждут очереди проехать в вагончике, который вмещает C человек, C < n. Вагончик может ехать только заполненным:

а) разработайте код действий процессов-пассажиров и процесса-вагончика. Для взаимодействия используйте передачу сообщений;

б) обобщите ответ к пункту *а*, чтобы использовались m процессов-вагончиков, m > 1. Поскольку есть только одна дорога, обгон вагончиков невозможен, т.е. заканчивать движение по дороге вагончики должны в том же порядке, в котором начали. Как и ранее, вагончик может ехать только заполненным;

в) напишите программу имитации вашего ответа к пункту *а*. Используйте язык C с библиотекой MPI или язык Java с модулем сокетов. Пусть пассажиры циклически совершают R поездок, вагончик едет по дорожке фиксированное время, а пассажиры задерживаются перед следующей поездкой на случайные отрезки времени. Напечатайте трассу основных событий при имитации.

7.13. Рассмотрите процесс планирующего драйвера диска (см. листинг 7.7):

а) измените процесс, чтобы использовалась стратегия планирования CSCAN (см. листинг 5.9);

б) измените планирующий драйвер диска (см. листинг 7.7), чтобы он использовал стратегию планирования SCAN (алгоритм лифта), описанную в начале раздела 5.3.

7.14. Следующие упражнения касаются программы файлового сервера (см. листинг 7.8):

а) в программе файловые серверы разделяют канал `open`, получая из него данные. Во многих реализациях передачи сообщений каждый канал может иметь только одного получателя (но обычно много отправителей). Измените программу сервера, чтобы учесть это ограничение;

б) в программе каждый клиент использует отдельный файловый сервер. Допустим, что существует только один файловый сервер. Измените его программу, чтобы одним и тем же сервером пользовались все клиенты. Ваше решение должно позволять всем клиентам одновременно иметь открытые файлы;

в) в программе каждый клиент пользуется отдельным файловым сервером. Пусть клиенты, желающие получить доступ к одному и тому же файлу, используют один и тот же сервер. Если файл уже открыт, клиент должен связаться с сервером, который управляет этим файлом. Иначе клиент должен начинать диалог со свободным файловым сервером. Измените программу файлового сервера, чтобы учесть эти требования.

7.15. *Задача об устойчивом паросочетании* (*о стабильных браках*) состоит в следующем. Пусть `Man[1:n]` и `Woman[1:n]` — массивы процессов. Каждый мужчина (man) оценивает женщин (woman) числами от 1 до n, и каждая женщина так же оценивает мужчин. *Паросочетание* — это взаимно однозначное соответствие между мужчинами и женщинами. Паросочетание *устойчиво*, если для любых двух мужчин m_1 и m_2 и двух женщин w_1 и w_2, соответствующих им в этом паросочетании, выполняются оба следующих условия:

- m_1 оценивает w_1 выше, чем w_2, или w_2 оценивает m_2 выше, чем m_1;
- m_2 оценивает w_2 выше, чем w_1, или w_1 оценивает m_1 выше, чем m_2.

Иными словами, паросочетание неустойчиво, если найдутся мужчина и женщина, предпочитающие друг друга своей текущей паре. Решением задачи является множество n-паросочетаний, каждое из которых устойчиво:

а) напишите программу для решения задачи. Процессы должны взаимодействовать с помощью асинхронной передачи сообщений. Мужчины должны предлагать, а женщины — слушать. Женщина должна принять первое полученное сообщение, поскольку лучшее предложение может и не поступить. Но когда женщина получит лучшее предложение, она может оставить первого мужчину;

б) разработайте программу имитации вашего ответа к пункту *а*. Используйте язык C с библиотекой MPI или язык Java с модулем сокетов. Напечатайте трассу основных событий при имитации или добавьте анимацию, чтобы наблюдать изменение состояния программы в реальном времени. (При использовании анимации придется притормаживать работу процессов, чтобы можно было наблюдать за их действиями.)

7.16. Даны три процесса `F`, `G` и `H`. У каждого из них есть локальный массив целых чисел. Все три массива отсортированы в неубывающем порядке и содержат хотя бы один элемент. Напишите программу, в которой эти три процесса взаимодействуют, чтобы найти наименьшее общее число. Приведите для каждого из процессов ключевые утверждения. В одном сообщении может передаваться только одно число.

7.17. *Распределенное паросочетание.* Даны `n` процессов, каждый из которых соответствует вершине связного графа. Вершина может взаимодействовать только со своими соседями. Задача состоит в том, чтобы каждый процесс образовал пару с одним из соседей. Когда процессы заканчивают образование пар, каждый из них должен оказаться в паре или остаться в одиночестве, но никакие два соседних процесса не могут остаться в одиночестве.

Решите эту задачу, используя для взаимодействия процессов асинхронную передачу сообщений. Каждый процесс выполняет один и тот же алгоритм. Когда программа завершается, каждый процесс должен сохранить в локальной переменной `pair` индекс соседа, с которым он образовал пару. Если процесс `i` остался один, то окончательным значением его переменной `pair` должно быть `i`. Ваше решение может не быть оптимальным, т.е. не обязательно минимизировать число одиночных процессов (тогда задача станет *очень* сложной).

7.18. В листинге 7.12 показана программа на языке CSP для генерации простых чисел с помощью решета Эратосфена. Измените ее так, чтобы использовались примитивы асинхронной передачи сообщений, определенные в разделе 7.1. Для сбора и печати результатов добавьте управляющий процесс.

7.19. В листингах 7.12 и 7.13 представлены два алгоритма для генерации простых чисел:

а) реализуйте эти алгоритмы на языке C с использованием библиотеки MPI;

б) сравните производительность полученных программ при разных количествах простых чисел, которые нужно сгенерировать. Каково общее время работы? Каково общее количество переданных сообщений?

7.20. Рассмотрите три программы обмена значениями в разделе 7.4:

а) реализуйте каждый алгоритм на языке C с использованием библиотеки MPI. При необходимости используйте примитивы глобального взаимодействия. Программа должна выполнить `R` циклов обмена;

б) сравните производительность программ. Укажите, как время выполнения программы зависит от числа процессов и числа циклов;

в) обмен числами — это обычно лишь часть большей задачи, такой как сеточные вычисления (см. раздел 11.1). На каждом цикле процессы выполняют вычисления и затем обмениваются результатами. В программы к пункту *а* добавьте коды для имитации вычислений (используйте цикл, в котором выполняется что-нибудь простое, например, увеличение счетчика итераций `C`). Если можно, расположите цикл вычислений так, чтобы его выполнение могло перекрывать по времени операции взаи-

модействия (точнее, после отправки, но перед приемом данных). Как теперь время выполнения зависит от числа процессов, числа циклов и значения C?

7.21. Пространство кортежей для примитивов Linda можно реализовать с помощью серверного процесса. Прикладные процессы при использовании примитивов `OUT`, `IN` или `EVAL` будут обращаться к серверу. Разработайте код процесса-сервера, демонстрирующий реализацию пространства кортежей и шести примитивов Linda. Покажите, какой код должен выполнять прикладной процесс для каждого примитива. Приложение и сервер должны взаимодействовать с помощью асинхронной передачи сообщений.

7.22. Таблица 7.1 демонстрирует дуальность мониторов и передачи сообщений. В исторической справке упоминались эксперименты, показавшие, что производительность программ с этими двумя типами средств примерно одинакова на мультипроцессорах с разделяемой памятью. Составьте набор экспериментов, чтобы доказать (или опровергнуть) эти результаты. Возьмите несколько задач, таких как "обедающие философы", "читатели и писатели" и диспетчер доступа к диску и запрограммируйте их решения в обоих стилях. Например, используйте библиотеку Pthreads для реализации программ с мониторами и библиотеку MPI для программ с передачей сообщений. Измерьте производительность каждой программы. Для контроля реализуйте передачу сообщений непосредственно в библиотеке Pthreads — при этом можно определить относительную производительность библиотек MPI и Pthreads.

7.23. Рассмотрим программу на языке Java для удаленного чтения файлов (см. листинги 7.15 и 7.16:

а) запустите программу на своем узле, после чего ответьте на вопросы, поставленные в конце раздела 7.9, и на другие подобные вопросы, которые могли у вас возникнуть;

б) измените программу, чтобы она записывала, а не читала удаленные файлы. Для этого клиент должен считывать строки из стандартного ввода и передавать их удаленной программе для записи в файл;

в) объедините программы к пунктам *а* и *б*, обеспечивая и чтение, и запись файлов. Для того чтобы определить, что хочет клиент (писать или читать), понадобится дополнительная опция командной строки. Поэкспериментируйте с программой и представьте отчет о полученных результатах. Для облегчения экспериментов программу клиента можно сделать интерактивной.

7.24. Рассмотрим программу генерации простых чисел на языке C-Linda (см. листинг 7.13). Реализуйте ее на языке Java. Поскольку разделяемого пространства кортежей уже не будет, портфель задач реализуйте с помощью управляющего процесса. Для взаимодействия рабочих и управляющего процессов используйте сокеты.

7.25. Разработайте программу имитации обедающих философов на языке C с библиотекой MPI или на языке Java с модулем сокетов. В программе должно быть пять процессов-философов и один процесс-сервер, управляющий столом. Прием пищи и размышления философов моделируйте паузами в работе случайной длительности. Программа должна выводить трассу интересных событий, происходящих во время работы.

7.26. Разработайте, реализуйте и документируйте параллельную или распределенную программу, которая осмысленно использует несколько процессов. Воспользуйтесь доступным вам языком программирования или библиотекой подпрограмм.

Ниже представлены возможные проекты. Выберите один из них или создайте свой, имеющий сравнимый уровень сложности. Дайте руководителю краткое описание проекта перед началом работы и продемонстрируйте завершенный проект:

а) возьмите задачу, которую можно распараллелить или распределить различными способами, например, сортировку, генерацию простых чисел, задачу коммивояжера,

умножение матриц. Разработайте различные алгоритмы решения этой задачи и проведите серию экспериментов, чтобы определить, какой из них работает лучше в каких условиях. По возможности проведите эксперименты на процессоре с разделяемой памятью и на машине с распределенной памятью;

б) постройте автоматизированную банковскую систему с несколькими пользовательскими "терминалами" и несколькими банками. Каждый терминал связан с одним банком, банки связаны между собой. Каждый счет хранится в банке, но пользователь должен иметь возможность получить доступ к своему счету с любого терминала. В программе следует реализовать несколько типов пользовательских транзакций, таких как помещение денег на счет, снятие денег, запрос баланса. В идеале создайте интерфейс пользователя, как в банкомате;

в) постройте систему заказа авиабилетов, которая позволяет пользователям делать запросы о полетах, а также резервировать и сдавать билеты. Обеспечьте поддержку нескольких клиентов. Этот проект аналогичен реализации автоматизированной банковской системы;

г) разработайте программу, которая играет в какую-нибудь игру и использует несколько дисплеев, показывая результаты нескольким игрокам или позволяя игрокам играть друг против друга и, возможно, против "машины". Если вы возьметесь за этот проект, выберите достаточно простую игру, чтобы не очень отвлекаться на ее программирование. Например, возьмите простую игру с картами, такую как покер, "концентрация", или простую видеоигру;

д) реализуйте команду "разговор" (chat), которая позволяет пользователям с разных терминалов общаться между собой. Поддержите и частные диалоги, и общие конференции. Дайте возможность человеку подключаться к существующему разговору;

е) разработайте модель физической системы с дискретными событиями. Это может быть движение машин через перекрестки, прибытие и отбытие самолетов из аэропортов, движение людей в лифтах здания. Реализуйте объекты моделей в виде процессов, которые генерируют события и реагируют на них. Можно использовать прямое взаимодействие объектов по мере возникновения событий или управляющий процесс, который отслеживает события и поддерживает модельные часы. "Оживите" программу, чтобы пользователи могли наблюдать процесс имитации и, возможно, взаимодействовать с ним.

Глава 8

Удаленный вызов процедур и рандеву

Передача сообщений идеально подходит для программирования фильтров и взаимодействующих равных, поскольку в этих случаях процессы отсылают данные по каналам связи в одном направлении. Как было показано, передача сообщений применяется также в программировании клиентов и серверов. Но двусторонний поток данных между клиентом и сервером приходится программировать с помощью двух явных передач сообщений по двум отдельным каналам. Кроме того, каждому клиенту нужен отдельный канал ответа; все это ведет к увеличению числа каналов.

В данной главе рассмотрены две дополнительные программные нотации — удаленный вызов процедур (remote procedure call — RPC) и рандеву, идеально подходящие для программирования взаимодействий типа "клиент-сервер". Они совмещают свойства мониторов и синхронной передачи сообщений. Как и при использовании мониторов, модуль или процесс экспортирует операции, а операции запускаются с помощью оператора `call`. Как и синхронизированная отправка сообщения, выполнение оператора `call` приостанавливает работу процесса. Новизна RPC и рандеву состоит в том, что они работают с двусторонним каналом связи: от процесса, вызывающего функцию, к процессу, который обслуживает вызов, и в обратном направлении. Вызвавший функцию процесс ждет, пока будет выполнена необходимая операция и возвращены ее результаты.

RPC и рандеву различаются способом обслуживания вызовов операций. Первый способ — для каждой операции объявлять процедуру и для обработки вызова создавать новый процесс (по крайней мере, теоретически). Этот способ называется *удаленным вызовом процедуры* (RPC), поскольку вызывающий процедуру процесс и ее тело могут находиться на разных машинах. Второй способ — назначить встречу (*рандеву*) с существующим процессом. Рандеву обслуживается с помощью оператора ввода (приема), который ждет вызова, обрабатывает его и возвращает результаты. (Иногда этот тип взаимодействия называется расширенным рандеву в отличие от простого рандеву, при котором встречаются операторы передачи и приема при синхронной передаче сообщений.)

В разделах 8.1 и 8.2 описаны типичные примеры программной нотации для RPC и рандеву, продемонстрировано их использование. Как упоминалось, эти методы упрощают программирование взаимодействий типа "клиент-сервер". Их можно использовать и при программировании фильтров, но мы увидим, что это трудоемкое занятие, поскольку ни RPC, ни рандеву напрямую не поддерживают асинхронную связь. К счастью, эту проблему можно решить, если объединить RPC, рандеву и асинхронную передачу сообщений в мощный, но достаточно простой язык, представленный в разделе 8.3.

Использование нотаций, их преимущества и недостатки демонстрируются на нескольких примерах. В некоторых из них использованы задачи, рассмотренные ранее, что помогает сравнить различные виды передачи сообщений. Некоторые задачи приводятся впервые и демонстрируют применимость RPC и рандеву в программировании взаимодействий типа "клиент-сервер". Например, в разделе 8.4 показано, как реализовать инкапсулированную базу данных и дублирование файлов. В разделах 8.5–8.7 дан обзор механизмов распределенного программирования трех языков: Java (RPC), Ada (рандеву) и SR (совместно используемые примитивы).

8.1. Удаленный вызов процедур

В главе 5 были использованы два типа компонентов программ: мониторы и процессы. Мониторы инкапсулируют разделяемые переменные; процессы взаимодействуют и синхронизируются, вызывая процедуры монитора. Предполагается, что процессы и мониторы программы находятся в общем адресном пространстве.

При удаленном вызове процедуры (RPC) используется только один из этих компонентов — модуль, в котором есть и процессы, и процедуры. Модули могут находиться в разных адресных пространствах, например, на различных узлах сети.[14] Процессы внутри модуля могут разделять переменные и вызывать процедуры, объявленные в этом модуле. Но процессы из одного модуля могут связываться с процессами в другом модуле, только вызывая процедуры, экспортируемые другим модулем.

Модуль состоит из двух частей, которые позволяют отличать процедуры, локальные для модуля, от обеспечивающих каналы взаимодействия. Часть модуля с определениями содержит заголовки операций, которые можно вызывать из других модулей. Тело содержит процедуры, реализующие эти операции; оно также может содержать локальные переменные, код инициализации, локальные процедуры и процессы. Модуль имеет такой вид.

```
module mname
   заголовки экспортируемых операций;
body
   объявления переменных;
   код инициализации;
   процедуры для экспортируемых операций;
   локальные процедуры и процессы;
end mname
```

Чтобы отличать локальные процессы от процессов экспортируемых операций (см. ниже), локальные процессы называются *фоновыми*.

Заголовок операции opname задается декларацией op:

```
op opname(параметры) [returns результат]
```

Параметры и раздел `returns` указывают типы и, возможно, имена параметров и возвращаемого значения. Раздел `returns` необязателен. При использовании удаленного вызова процедур операция задается в виде `proc`.

```
proc opname(параметры) returns идентификатор результата
   объявления локальных переменных;
   операторы
end
```

Таким образом, декларация `proc` аналогична процедуре, но типы параметров и результата в ней повторять не обязательно, поскольку они указаны в объявлении op. (Процедуру можно рассматривать просто как сокращенную форму декларации op и `proc`.)

Как и при использовании мониторов, процесс (или процедура) одного модуля вызывает процедуру из другого модуля, выполняя код:

```
call mname.opname(аргументы)
```

---

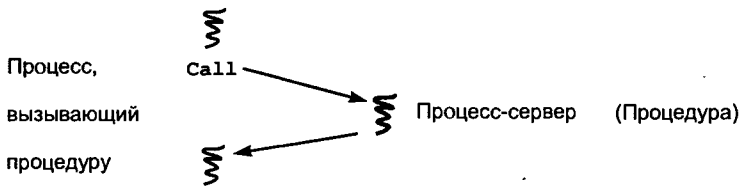
[14] Процессы в одном адресном пространстве часто называются *облегченными потоками*. Термин *поток* показывает, что каждый из процессов имеет отдельную последовательность выполнения, а *облегченный* — что при переключении контекста такому процессу нужно сохранять относительно мало информации. В отличие от облегченных, такие "тяжеловесные" процессы, как процессы в ОС UNIX, имеют собственное адресное пространство. Переключение контекста таких "тяжеловесных" процессов требует сохранения и загрузки таблиц управления памятью и регистров.

Ключевое слово `call` не обязательно для вызовов процедур, а в вызовах функций оно не используется. Для локальных вызовов имя модуля можно не указывать.

Реализация межмодульного вызова отличается от реализации локального вызова, поскольку модули могут находиться в разных адресных пространствах. В частности, вызов обслуживается *новым процессом*, а аргументы передаются в виде сообщений от вызывающего процесса к серверному. Вызывающий процесс ждет, пока процесс-сервер выполняет тело процедуры, реализующей операцию opname. Возвращаясь из opname, сервер отсылает результаты и возвращаемое значение вызвавшему процессу, а затем завершается. После получения результатов вызывавший процесс продолжается. Передача и получение результатов не программируются, поскольку происходят неявно.

Если вызывающий процесс и процедура находятся в одном адресном пространстве, часто можно не создавать новый процесс для обслуживания удаленного вызова, поскольку вызывающий процесс на некоторое время становится сервером и выполняет тело процедуры (подробности — в главе 10). Но в общем случае вызов будет удаленным, поэтому процесс нужно создавать или выделять из уже существующего набора серверов.

Следующая диаграмма иллюстрирует взаимодействие между процессом, вызывающим процедуру, и процессом-сервером.



Ось времени проходит по рисунку вниз, волнистые линии показывают ход выполнения процесса. Когда вызывающий процесс доходит до оператора `call`, он приостанавливается, пока сервер выполняет тело вызванной процедуры. После того как сервер возвратит результаты, вызвавший процесс продолжается.

8.1.1. Синхронизация в модулях

Сам по себе RPC — это механизм взаимодействия. Хотя вызывающий процесс синхронизируется со своим сервером, единственная роль сервера — действовать от имени вызывающего процесса. Теоретически все происходит так же, как если бы вызывающий процесс сам выполнял процедуру, поэтому синхронизация между вызывающим процессом и сервером происходит неявно.

Нужен также способ обеспечивать взаимно исключающий доступ процессов модуля к разделяемым переменным и их синхронизацию. Процессы модуля — это процессы-серверы, выполняющие удаленные вызовы, и фоновые процессы, объявленные в модуле.

Существует два подхода к обеспечению синхронизации в модулях. Первый — считать, что все процессы одного модуля должны выполняться со взаимным исключением, т.е. в любой момент времени может быть активен только один процесс. Этот метод аналогичен неявному исключению, определенному для мониторов, и гарантирует защиту разделяемых переменных от одновременного доступа. Но при этом нужен способ программирования условной синхронизации процессов, для чего можно использовать условные переменные (как в мониторах) или семафоры.

Второй подход — считать, что все процессы выполняются параллельно, и явным образом программировать взаимное исключение и условную синхронизацию. Тогда каждый модуль сам становится параллельной программой, и можно применить любой описанный метод. Например, можно использовать семафоры или локальные мониторы. В действительности, как будет показано в этой главе ниже, можно использовать рандеву (или даже передачу сообщений).

Программа, содержащая модули, выполняется, начиная с кода инициализации каждого из модулей. Коды инициализации разных модулей могут выполняться параллельно при условии,

что в них нет удаленных вызовов. Затем запускаются фоновые процессы. Если исключение неявное, то одновременно в модуле выполняется только один фоновый процесс; когда он приостанавливается или завершается, может выполняться другой фоновый процесс. Если процессы модуля выполняются параллельно, то все фоновые процессы модуля могут начинаться одновременно.

У неявной формы исключения есть два преимущества. Первое — модули проще программировать, поскольку разделяемые переменные автоматически защищаются от конфликтов одновременного доступа. Второе — реализация модулей, выполняемых на однопроцессорных машинах, может быть более эффективной. Дело в том, что переключение контекста происходит только в точках входа, возврата или приостановки процедур или процессов, а не в произвольных точках, когда регистры могут содержать результаты промежуточных вычислений.

С другой стороны, предположение о параллельном выполнении процессов является более общим. Параллельное выполнение — это естественная модель для программ, работающих на обычных теперь мультипроцессорах с разделяемой памятью. Кроме того, с помощью параллельной модели выполнения можно реализовать квантование времени, чтобы разделять его между процессами и "обуздывать" неуправляемые процессы (например, зациклившиеся). Это невозможно при использовании исключающей модели выполнения, если только процессы сами не освобождают процессор через разумные промежутки времени, поскольку контекст можно переключить, только когда выполняемый процесс достигает точки выхода или приостановки.

Итак, будем предполагать, что процессы внутри модуля выполняются параллельно, и поэтому необходимо программировать взаимное исключение и условную синхронизацию. В следующих двух разделах показано, как программировать сервер времени и кэширование в распределенной файловой системе с использованием семафоров.

8.1.2. Сервер времени

Рассмотрим задачу реализации сервера времени — модуля, который обслуживает работу с временными интервалами клиентских процессов из других модулей. Предположим, что в сервере времени определены две видимые операции: `get_time` и `delay`. Клиентский процесс получает время суток, вызывая операцию `get_time`, и блокируется на `interval` единиц времени с помощью операции `delay`. Сервер времени также содержит внутренний процесс, который постоянно запускает аппаратный таймер и при возникновении прерывания от таймера увеличивает время суток.

Листинг 8.1 содержит программу модуля сервера времени. Время суток хранится в переменной `tod` (time of day). Несколько клиентов могут вызывать функции `get_time` и `delay` одновременно, поэтому несколько процессов могут одновременно обслуживать вызовы. Такое обслуживание нескольких вызовов операции `get_time` безопасно для процессов, поскольку они просто считывают значение `tod`. Но операции `delay` и `tick` должны выполняться со взаимным исключением, обрабатывая очередь "уснувших" клиентских процессов `napQ`. Вместе с тем, в операции `delay` присваивание значения переменной `wake_time` может не быть критической секцией, поскольку переменная `tod` — это единственная разделяемая переменная, которая просто считывается. Кроме того, увеличение `tod` в процессе `Clock` также может не быть критической секцией, поскольку только процесс `Clock` может присваивать значение этой переменной.

Листинг 8.1. Модуль сервера времени

```
module TimeServer
  op get_time() returns int;  # получить время суток
  op delay(int interval);     # ждать interval "тиков"
body
  int tod = 0;          # время суток
  sem m = 1;            # семафор взаимного исключения
  sem d[n] = ([n] 0);   # скрытые семафоры задержек
  queue of (int waketime, int process_id) napQ;
```

```
## когда m == 1, tod < waketime для приостановленных процессов

  proc get_time() returns time {
    time = tod;
  }

  proc delay(interval) { # предполагается, что interval > 0
    int waketime = tod + interval;
    P(m);
    вставить(waketime, myid) в подходящую позицию napQ;
    V(m);
    P(d[myid]);    # ждать запуска
  }

  process Clock {
    запустить аппаратный таймер;
    while (true) {
      ждать прерывания, затем перезапустить аппаратный таймер;
      tod = tod+1;
      P(m);
      while (tod >= наименьшее waketime из napQ) {
        удалить (waketime, id) из napQ;
        V(d[id]);  # разбудить процесс id
      }
      V(m);
    }
  }
end TimeServer
```

Предполагается, что значение переменной `myid` в процессе `delay` является уникальным целым числом в промежутке от 0 до `n-1`. Оно используется для указания скрытого семафора, на котором приостановлен клиент. После прерывания от часов процесс `Clock` выполняет цикл проверки очереди `napQ`; он сигнализирует соответствующему семафору задержки, когда заданный интервал задержки заканчивается. Может быть несколько процессов, ожидающих одно и то же время запуска, поэтому используется цикл.

8.1.3. Кэширование в распределенной файловой системе

Рассмотрим упрощенную версию задачи, возникающей в распределенных файловых системах и базах данных. Предположим, что прикладные процессы выполняются на рабочей станции, а файлы данных хранятся на файловом сервере. Не будем останавливаться на том, как файлы открываются и закрываются, а сосредоточимся на их чтении и записи. Когда прикладной процесс хочет получить доступ к файлу, он вызывает процедуру `read` или `write` в локальном модуле `FileCache`. Будем считать, что приложения читают и записывают массивы символов (байтов). Иногда это может быть несколько символов, а иногда — тысячи.

Файлы хранятся на диске файлового сервера в блоках фиксированного размера (например, по 1024 байт). Модуль `FileServer` управляет доступом к блокам диска. Для чтения и записи целых блоков он обеспечивает две операции, `readblk` и `writeblk`.

Модуль `FileCache` кэширует последние считанные блоки данных. Когда приложение запрашивает чтение части файла, модуль `FileCache` сначала проверяет, есть ли эти данные в его кэш-памяти. Если есть, то он может быстро обработать запрос клиента. Если нет, он должен вызвать процедуру `readblk` из модуля `FileServer` для получения блоков диска с запрашиваемыми данными. (Модуль `FileCache` может производить упреждающее чтение, если определит, что происходят последовательные обращения к файлу. А это бывает часто.)

Запросы на запись обрабатываются аналогично. Когда приложение вызывает операцию `write`, данные сохраняются в блоке в локальной кэш-памяти. Когда блок заполнен или затребован для другого запроса, модуль `FileCache` вызывает операцию `writeblk` модуля `FileServer` для записи блока на диск. (Модуль `FileCache` также может использовать стратегию сквозной записи, при которой `writeblk` вызывается после каждого запроса на запись. Использование сквозной записи гарантирует сохранение данных на диске при завершении операции `write`, но замедляет ее выполнение.)

В листинге 8.2 показаны схемы обоих модулей. Каждый модуль выполняется на отдельной машине. Вызовы модуля `FileCache` из приложения фактически являются локальными, а вызовы в этом модуле операций из модуля `FileServer` — удаленными. Модуль `FileCache` является сервером для прикладных процессов, а `FileServer` — для многих клиентов модуля `FileCache`, по одному на рабочую станцию.

Листинг 8.2.1. Распределенная файловая система: файловый кэш

```
module FileCache   # расположен на бездисковых станциях
  op read(int count; result char buffer[*]);
  op write(int count; char buffer[]);
body
  кэш файловых блоков;
  переменные для хранения информации дескриптора файла;
  семафоры для синхронизации доступа к кэш-памяти (если нужны);

  proc read(count,buffer) {
    if (нужные данные не находятся в кэш-памяти) {
      выбрать блок кэш-памяти для использования;
      if (необходимо записать блок кэш-памяти)
        FileServer.writeblk(...);
      FileServer.readblk(...);
    }
    buffer = соответствующие count байт из блока кэш-памяти;
  }

  proc write(count,buffer) {
    if (соответствующий блок не находится в кэш-памяти) {
      выбрать блок кэш-памяти для использования;
      if (необходимо записать блок кэш-памяти)
        FileServer.writeblk(...);
    }
    блок кэш-памяти = count байт из buffer;
  }
end FileCache
```

Листинг 8.2.2. Распределенная файловая система: файловый сервер

```
module FileServer   # расположен на файловом сервере
  op readblk(int fileid, offset; result char blk[1024]);
  op writeblk(int fileid, offset; char blk[1024]);
body
  кэш-память дисковых блоков;
  очередь запросов на доступ к диску;
  семафоры для синхронизации доступа к кэш-памяти и очереди;

  # N.B. код синхронизации не показан
```

```
  proc readblk(fileid, offset, blk) {
    if (нужный блок не находится в кэш-памяти) {
      сохранить запрос на чтение в очереди диска;
      ждать выполнения операции чтения;
    }
    blk = подходящий дисковый блок;
  }
  proc writeblk(fileid, offset, blk) {
    выбрать блок из кэш-памяти;
    if (необходимо записать выбранный блок) {
      сохранить запрос на запись в очереди диска;
      ждать записи блока на диск;
    }
    блок кэш-памяти = blk;
  }

  process DiskDriver {
    while (true) {
      ждать запроса на доступ к диску;
      начать дисковую операцию; ждать прерывания;
      запустить процесс, ожидающий завершения данного запроса;
    }
  }
end FileServer
```

При условии, что у каждого прикладного процесса есть отдельный модуль `FileCache`, внутренняя синхронизация в этом модуле не нужна, поскольку в любой момент времени может выполняться только один запрос чтения или записи. Но если несколько прикладных процессов используют один модуль `FileCache` или в нем есть процесс, который реализует упреждающее чтение, то для обеспечения взаимно исключающего доступа к разделяемой кэш-памяти в этом модуле необходимо использовать семафоры.

В модуле `FileServer` внутренняя синхронизация необходима, поскольку он совместно используется несколькими модулями `FileCache` и содержит внутренний процесс `DiskDriver`. В частности, необходимо синхронизировать процессы, обрабатывающие вызовы операций `writeblk` и `readblk`, и процесс `DiskDriver`, чтобы защитить доступ к кэш-памяти дисковых блоков и планировать операции доступа к диску. В листинге 8.2 код синхронизации не показан, но его нетрудно написать, используя методы из главы 4.

8.1.4. Сортирующая сеть из фильтров слияния

Хотя RPC упрощает программирование взаимодействий "клиент-сервер", его неудобно использовать для программирования фильтров и взаимодействующих равных. В этом разделе еще раз рассматривается задача реализации сортирующей сети из фильтров слияния, представленной в разделе 7.2, и показан способ поддержки динамических каналов связи с помощью указателей на операции в других модулях.

Напомним, что фильтр слияния получает два входных потока и производит один выходной. Предполагается, что каждый входной поток отсортирован, и задача фильтра — объединить значения из входных потоков в отсортированный выходной. Как и в разделе 7.2, предположим, что конец входного потока обозначен маркером `EOS`.

Первая проблема при программировании фильтра слияния с помощью RPC состоит в том, что RPC не поддерживает непосредственное взаимодействие процесс-процесс. Вместо этого в программе нужно явно реализовывать межпроцессное взаимодействие, поскольку для него нет примитивов, аналогичных примитивам для передачи сообщений.

Еще одна проблема — связать между собой экземпляры фильтров. Каждый фильтр должен направить свой выходной поток во входной поток другого фильтра, но имена операций, представляющих каналы взаимодействия, являются различными идентификаторами. Таким образом, каждый входной поток должен быть реализован отдельной процедурой. Это затрудняет использование статического именования, поскольку фильтр слияния должен знать символьное имя операции, которую нужно вызвать для передачи выходного значения следующему фильтру. Намного удобнее использовать динамическое именование, при котором каждому фильтру передается ссылка на операцию, используемую для вывода. Динамическая ссылка представляется *мандатом доступа* (capabililty), который можно рассматривать как указатель на операцию.

Листинг 8.3 содержит модуль, реализующий массив фильтров `Merge`. В первой строке модуля дано глобальное определение типа операций `stream`, получающих в качестве аргумента одно целое число. Каждый модуль экспортирует две операции, `in1` и `in2`. Они обеспечивают входные потоки и могут использоваться другими модулями для получения входных значений. Модули экспортируют третью операцию, `initialize`, которую вызывает главный модуль (не показан), чтобы передать фильтру мандат доступа к используемому выходному потоку. Например, главный модуль может дать фильтру `Merge[i]` мандат доступа к операции `in2` фильтра `Merge[j]` с помощью следующего кода:

```
call Merge[i].initialize(Merge[j].in2)
```

Предполагается, что каждый модуль инициализируется до начала любых операций вывода.

Листинг 8.3. Фильтры для сортировки слиянием, использующие RPC

```
optype stream = (int);  # тип операций с потоком данных

module Merge[i = 1 to n]
  op in1 stream, in2 stream;  # входные потоки
  op initialize(cap stream);  # ссылка на выходной поток
body
  int v1, v2;     # входные значения из потоков 1 и 2
  cap stream out;  # мандат доступа ко входному потоку
  sem empty1 = 1, full1 = 0, empty2 = 1, full2 = 0;

  proc initialize(output) {  # обеспечить выходной поток
    out = output;
  }

  proc in1(value1) {  # создать новое значение для потока 1
    P(empty1); v1 = value1; V(full1);
  }

  proc in2(value2) {  # создать новое значение для потока 2
    P(empty2); v2 = value2; V(full2);
  }

  process M {
    P(full1); P(full2);  # ожидать два входных значения
    while (v1 != EOS and v2 != EOS)
      if (v1 <= v2)
          { call out(v1); V(empty1); P(full1); }
      else  # v2 < v1
          { call out(v2); V(empty2); P(full2); }
    # считать остаток непустого входного потока
    if (v1 == EOS)
      while (v2 != EOS)
          { call out(v2); V(empty2); P(full2); }
```

```
    else  # v2 == EOS
      while (v1 != EOS)
          { call out(v1); V(empty1); P(full1); }
    call out(EOS); # присоединить маркер конца
  }
end Merge
```

Остальная часть модуля аналогична процессу `Merge` (см. листинг 7.2). Переменные `v1` и `v2` соответствуют одноименным переменным в листинге 7.2, а процесс `M` повторяет действия процесса `Merge`. Однако процесс `M` для помещения следующего значения в выходной канал `out` использует оператор `call`, а не `send`. Процесс `M` для получения следующего числа из соответствующего входного потока использует операции семафора. Внутри модуля неявные серверные процессы, которые обрабатывают вызовы операций `in1` и `in2`, являются производителями, а процесс `M` — потребителем. Эти процессы синхронизируются так же, как процессы производителей и потребителей в листинге 4.3.

Сравнение программ в листингах 8.3 и 7.2 четко показывает недостатки PRC по отношению к передаче сообщений при программировании фильтров. Хотя процессы в обоих листингах похожи, для работы программы 8.3 необходимы дополнительные фрагменты. В результате программа работает примерно с такой же производительностью, но, используя RPC, программист должен написать намного больше.

8.1.5. Взаимодействующие равные: обмен значений

RPC можно использовать и для программирования обмена информацией, возникающего при взаимодействии равных процессов. Однако по сравнению с использованием передачи сообщений программы получаются громоздкими. В качестве примера рассмотрим взаимодействие двух процессов из разных модулей, которым необходимо обменять значения. Чтобы связаться с другим модулем, каждый процесс должен использовать RPC, поэтому каждый модуль должен экспортировать процедуру, вызываемую из другого модуля.

В листинге 8.4 показан один из способов программирования обмена значений. Для пересылки значения из одного модуля в другой используется операция `deposit`. Для реализации обмена каждый из рабочих процессов выполняет два шага: передает значение `myvalue` в другой модуль, а затем ждет, пока другой процесс не присвоит это значение своей локальной переменной. (Выражение `3-i` в каждом модуле задает номер модуля, с которым нужно взаимодействовать; например, модуль 1 должен обратиться к модулю с номером `3-1`, т.е. 2.) В модулях используется семафор `ready`; он гарантирует, что рабочий процесс не получит доступ к переменной `othervalue` до того, как ей будет присвоено значение в операции `deposit`.

Листинг 8.4. Обмен значений с использованием RPC

```
module Exchange[i = 1 to 2]
  op deposit(int);
body
  int othervalue;
  sem ready = 0;   # используется для сигнализации
  proc deposit(other) {  # вызывается из другого модуля
    othervalue = other;  # сохранить полученное значение
    V(ready);        # разрешить процессу Worker забрать его
  }

  process Worker {
    int myvalue;
    call Exchange[3-i].deposit(myvalue);  # отослать другому
```

```
    P(ready);     # ждать получения значения из другого процесса
    ...
  }
end Exchange
```

8.2. Рандеву

Сам по себе RPC обеспечивает только механизм межмодульного взаимодействия. Внутри модуля все равно нужно программировать синхронизацию. Иногда приходится определять дополнительные процессы, чтобы обрабатывать данные, передаваемые с помощью RPC. Это было показано в модуле `Merge` (см. листинг 8.3).

Рандеву сочетает взаимодействие и синхронизацию. Как и при PRC, клиентский процесс вызывает операцию с помощью оператора `call`. Но операцию обслуживает уже существующий, а не вновь создаваемый процесс. В частности, процесс-сервер использует *оператор ввода*, чтобы ожидать и затем действовать в пределах одного вызова. Следовательно, операции обслуживаются по одной, а не параллельно.

Как и в предыдущем разделе, часть модуля с определениями содержит объявления заголовков операций, экспортируемых модулем, но тело модуля теперь состоит из одного процесса, обслуживающего операции. (В следующем разделе это обобщается.) Используются также массивы операций, объявляемые с помощью добавления диапазона значений индекса к имени операции.

8.2.1. Операторы ввода

Предположим, что модуль экспортирует следующую операцию.

```
op opname(типы параметров);
```

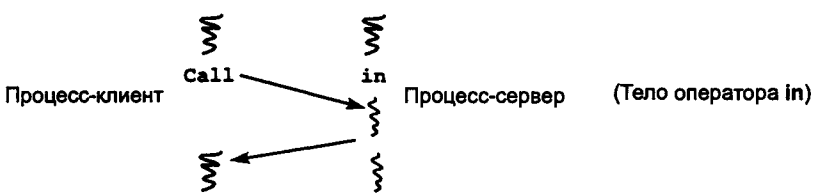
Процесс-сервер этого модуля осуществляет рандеву с процессом, вызвавшим операцию `opname`, выполняя *оператор ввода*. Простейший вариант оператора ввода имеет вид:

```
in opname(параметры) -> S; ni
```

Область между ключевыми словами `in` и `ni` называется *защищенной операцией*. Защита именует операцию и обеспечивает идентификаторы для ее параметров (если они есть). `S` обозначает список операторов, обслуживающих вызов операции. Областью видимости параметров является вся защищенная операция, поэтому операторы из `S` могут считывать и записывать значения параметров.

Оператор ввода приостанавливает работу процесса-сервера до появления хотя бы одного вызова операции `opname`. Затем процесс выбирает самый старый из ожидающих вызовов, копирует значения его аргументов в параметры, выполняет список операторов `S` и, наконец, возвращает результирующие параметры вызвавшему процессу. В этот момент и процесс-сервер, выполняющий `in`, и клиентский процесс, который вызывал `opname`, могут продолжать работу.

Следующая диаграмма отражает отношения между вызывающим и серверным процессами. Время возрастает в диаграмме сверху вниз, а волнистые линии показывают, когда процесс выполняется.



Как и при использовании RPC, процесс, достигший оператора `call`, приостанавливается и возобновляется после того, как процесс-сервер выполнит вызванную операцию. Однако при использовании рандеву сервер является активным процессом, который работает и до, и после обслуживания удаленного вызова. Как было указано выше, сервер также задерживается, достигая оператора `in`, если нет ожидающих выполнения вызовов. Читателю было бы полезно сравнить приведенную диаграмму с аналогичной диаграммой для RPC.

Приведенный выше оператор ввода обслуживает *одну* операцию. Как сказано в разделе 7.6, защищенное взаимодействие полезно тем, что позволяет процессу ожидать выполнения одного из *нескольких* условий. Можно объединить рандеву и защищенное взаимодействие, используя общую форму оператора ввода.

```
in op1(параметры1) and B1 by e1 -> S1;
[] ...
[] opn(параметрыn) and Bn by en -> Sn;
ni
```

Каждая ветвь оператора `in` является защищенной операцией. Часть кода перед символами `->` называется *защитой*; каждое S_i обозначает последовательность операторов. Защита содержит имя операции, ее параметры, необязательное *условие синхронизации* `and` B_i и необязательное *выражение планирования* `by` e_i. В этих условиях и выражениях могут использоваться параметры операции.

В языке Ada (раздел 8.6) поддержка рандеву реализована с помощью оператора `accept`, а защищенное взаимодействие — оператора `select`. Оператор `accept` очень похож на `in` в простой форме, а оператор `select` — на общую форму `in`. Но `in` в общей форме предоставляет больше возможностей, чем `select`, поскольку в операторе `select` нельзя использовать аргументы операции и выражения планирования. Эти различия обсуждаются в разделе 8.6.

Защита в операторе ввода *пропускает*, если была вызвана операция и соответствующее условие синхронизации истинно (или отсутствует). Область видимости параметров включает всю защищенную операцию, поэтому условие синхронизации может зависеть от значений параметров, т.е. от значений аргументов в вызове операции. Таким образом, один вызов операции может привести к тому, что защита пропустит, а другой — что не пропустит.

Выполнение оператора `in` приостанавливает работу процесса, пока не пропустит какая-нибудь защита. Если пропускают несколько защит (и нет условий планирования), то оператор `in` обслуживает первый (по времени) вызов, пропускаемый защитой. Аргументы этого вызова копируются в параметры, и затем выполняется соответствующий список операторов. По завершении операторов результирующие параметры и возвращаемое значение (если есть) возвращаются процессу, вызвавшему операцию. В этот момент операторы `call` и `in` завершаются.

Выражение планирования используется для изменения порядка обработки вызовов, используемого по умолчанию (первым обслуживается самый старый вызов). Если есть несколько вызовов, пропускаемых защитой, то первым обслуживается самый старый вызов, у которого выражение планирования имеет минимальное значение. Как и условие синхронизации, выражение планирования может ссылаться на параметры операции, и, следовательно, его значение может зависеть от аргументов вызова операции. В действительности, если в выражении планирования используются только локальные переменные, его значение одинаково для всех вызовов и не влияет на порядок обслуживания вызовов.

Как мы увидим далее, условия синхронизации и выражения планирования очень полезны. Они используются не только в рандеву, но и в синхронной и асинхронной передаче сообщений. Например, можно позволить операторам `receive` задействовать свои параметры, и многие библиотеки передачи сообщений обеспечивают средства для управления порядком получения сообщений. Например, в библиотеке MPI получатель сообщения может определять отправителя и тип сообщения.

8.2.2. Примеры взаимодействий типа "клиент-сервер"

В данном разделе представлены небольшие примеры, иллюстрирующие использование операторов ввода. Вернемся к задаче реализации кольцевого буфера. Нам нужен процесс, который имеет локальный буфер на n элементов и обслуживает две операции: deposit и fetch. Вызывая операцию deposit, производитель помещает элемент в буфер, а с помощью операции fetch потребитель извлекает элемент из буфера. Как обычно, операция deposit должна задерживаться, если в буфере уже есть n элементов, а операция fetch — пока в буфере не появится хотя бы один элемент.

Листинг 8.5 содержит модуль, реализующий кольцевой буфер. Процесс Buffer объявляет локальные переменные, которые представляют буфер, и затем циклически выполняет оператор ввода. На каждой итерации процесс Buffer ждет вызова операции deposit или fetch. Условия синхронизации в защитах обеспечивают необходимые задержки операций deposit и fetch.

Листинг 8.5. Реализация кольцевого буфера с помощью рандеву

```
module BoundedBuffer
  op deposit(typeT), fetch(result typeT);
body
  process Buffer {
    typeT buf[n];
    int front = 0, rear = 0, count = 0;
    while (true)
      in deposit(item) and count < n ->
            buf[rear] = item;
            rear = (rear+1) mod n; count = count+1;
      [] fetch(item) and count > 0 ->
            item = buf[front];
            front = (front+1) mod n; count = count-1;
      ni
  }
end BoundedBuffer
```

Полезно сравнить процесс Buffer и монитор в листинге 5.3. Интерфейсы клиентских процессов и результаты вызова операций deposit и fetch у них одинаковые, а реализации совершенно разные. Тела процедур в реализации монитора превратились в список операторов в операторе ввода, и условие синхронизации выражается с помощью логических выражений, а не условных переменных.

Еще один пример: листинг 8.6 содержит модуль, реализующий централизованное решение задачи об обедающих философах. Структура процесса Waiter аналогична структуре процесса Buffer. Вызов операции getforks может быть обслужен, если ни один из соседей не ест, а вызов операции relforks — всегда. Философ передает свой индекс i процессу Waiter, который использует этот индекс в условии синхронизации защиты для getforks. Предполагается, что в этой защите вызовы функций left(i) и right(i) возвращают индексы соседей слева и справа философа Philosopher[i].

Листинг 8.6. Централизованное решение задачи об обедающих философах на основе рандеву

```
module Table
  op getforks(int), relforks(int);
body
  process Waiter {
```

```
    bool eating[5] = ([5] false);
    while (true)
      in getforks(i) and not (eating[left(i)] and
           not eating[right(i)] -> eating[i] = true;
      [] relforks(i) ->
           eating[i] = false;
      ni
  }
end Table

process Philosopher[i = 0 to 4] {
  while (true) {
    call getforks(i);
    поесть;
    call relforks(i);
    поразмышлять;
  }
}
```

Листинг 8.7 содержит модуль сервера времени, по назначению аналогичный модулю в листинге 8.1. Операции `get_time` и `delay` экспортируются для клиентов, а `tick` — для обработчика прерывания часов. В листинге 8.7 аргументом операции `delay` является время, в которое должен быть запущен клиентский процесс. Клиентский интерфейс данного модуля несколько отличается от интерфейса, приведенного в листинге 8.1. Клиентские процессы должны передавать время запуска, чтобы для управления порядком обслуживания вызовов `delay` можно было использовать условие синхронизации. В программе с применением рандеву процесс `Timer` может не поддерживать очередь приостановленных процессов; вместо этого приостановленными являются те процессы, время запуска которых еще не пришло. (Их вызовы остаются в очереди канала `delay`.)

Листинг 8.7. Сервер времени с использованием рандеву

```
module TimeServer
  op get_time() returns int;
  op delay(int);
  op tick();       # вызывается обработчиком прерывания часов
body TimeServer
  process Timer {
    int tod = 0;  # время суток
    while (true)
      in get_time() returns time -> time = tod;
      [] delay(waketime) and waketime <= tod -> skip;
      [] tick() -> { tod = tod+1; перезапустить таймер; }
      ni
  }
end TimeServer
```

В последнем примере наряду с условием синхронизации используется выражение планирования. В листинге 8.8 показан модуль для распределения ресурсов по принципу "кратчайшее задание". Клиентский интерфейс в данном случае тот же, что и в мониторе в листинге 5.5, но реализация операций `request` и `release` значительно отличается. Процесс SJN, как и процесс `Timer`, может не поддерживать внутренние очереди. Вместо этого он просто откладывает прием вызовов `request` до освобождения ресурса, а затем выбирает вызовы с наименьшим значением аргумента, соответствующего `time`.

Листинг 8.8. Распределение ресурсов по принципу "кратчайшее задание" с помощью рандеву

```
module SJN_Allocator
  op request(int time), release();
body
  process SJN {
    bool free = true;
    while (true)
      in request(time) and free by time -> free = false;
      [] release() -> free = true;
      ni
  }
end SJN_Allocator
```

8.2.3. Сортирующая сеть из фильтров слияния

Снова рассмотрим задачу реализации сортирующей сети с использованием фильтров слияния и решим ее, используя механизм рандеву. Есть два пути. Первый — использовать два вида процессов: один для реализации фильтров слияния и один для реализации буферов взаимодействия. Между каждой парой фильтров поместим процесс-буфер, реализованный в листинге 8.5. Каждый процесс-фильтр будет извлекать новые значения из буферов между этим процессом и его предшественниками в сети фильтров, сливать их и помещать свой выход в буфер между ним и следующим фильтром сети.

Аналогично описанному сети фильтров реализованы в операционной системе UNIX, где буферы обеспечиваются так называемыми каналами UNIX. Фильтр получает входные значения, читая из входного канала (или файла), а отсылает результаты, записывая их в выходной канал (или файл). Каналы реализуются не процессами; они больше похожи на мониторы, но процессы фильтров используют их таким же образом.

Второй путь для программирования фильтров — использовать операторы ввода для извлечения входных значений и операторы `call` для передачи выходных. При таком подходе фильтры взаимодействуют между собой напрямую. В листинге 8.9 показан массив фильтров для сортировки слиянием, запрограммированных по второму методу. Как и в листинге 8.3, фильтр получает значения из двух входных потоков и отсылает результаты в выходной поток. Здесь также используется динамическое именование, чтобы с помощью операции `initialize` дать каждому процессу мандат доступа к выходному потоку, который он должен использовать. Этот поток связан со входным потоком другого элемента массива модулей `Merge`. Несмотря на эти общие черты, программы в листингах 8.3 и 8.8 совершенно разные, поскольку рандеву, в отличие от RPC, поддерживает прямую связь между процессами. Поэтому для программирования процессов-фильтров легче использовать рандеву.

Листинг 8.9. Фильтры сортировки слиянием с использованием рандеву

```
optype stream = (int);  # тип потоков данных

module Merge[i = 1 to n]
  op in1 stream, in2 stream;  # входные потоки
  op initialize(cap stream);  # ссылка на выходной поток
body
  process Filter {
    int v1, v2;       # значения из входных потоков
    cap stream out;   # мандат доступа к выходному потоку
    in initialize(c) -> out = c ni
    # получить первые значения из входных потоков
    in in1(v) -> v1 = v; ni
```

```
    in in2(v) -> v2 = v; ni
    while (v1 != EOS and v2 != EOS)
      if (v1 <= v2)
          { call out(v1); in in1(v) -> v1 = v; ni }
      else   # v2 < v1
          { call out(v2); in in2(v) -> v2 = v; ni }
    # получить оставшиеся значения из непустого входного потока
    if (v1 == EOS)
      while (v2 != EOS)
          { call out(v2); in in2(v) -> v2 = v; ni }
    else  # v2 == EOS
      while (v1 != EOS)
          { call out(v1); in in1(v) -> v1 = v; ni }
    call out(EOS);
  }
end Merge
```

Процесс в листинге 8.9 похож на процесс из программы в листинге 7.2, который был запрограммирован с помощью асинхронной передачи сообщений. Операторы взаимодействия запрограммированы по-разному, но находятся в одних и тех же местах программ. Однако, поскольку оператор `call` является блокирующим, выполнение процесса гораздо теснее связано с рандеву, чем с асинхронной передачей сообщений. В частности, разные процессы `Filter` будут выполняться примерно с одинаковой скоростью, поскольку каждый поток будет всегда содержать не больше одного числа. (Процесс-фильтр не может вывести в поток второе значение, пока другой фильтр не получит первое.)

8.2.4. Взаимодействующие равные: обмен значений

Вернемся к задаче о процессах из двух модулей, которые обмениваются значениями переменных. Из листинга 8.4 видно, насколько сложно решить эту задачу с использованием RPC. Упростить решение можно, используя рандеву, хотя это хуже, чем передача сообщений.

Используя рандеву, процессы могут связываться между собой напрямую. Но, если оба процесса сделают вызовы одновременно, они заблокируют друг друга. Аналогично процессы одновременно не могут выполнять операторы `in`. Таким образом, решение должно быть асимметричным; один процесс должен выполнить оператор `call` и затем `in`, а другой — сначала `in`, а затем `call`. Это решение представлено в листинге 8.10. Требование асимметрии процессов приводит к появлению оператора `if` в каждом процессе `Worker`. (Асимметричное решение можно получить, имитируя программу с RPC в листинге 8.4, но это еще сложнее.)

Листинг 8.10. Обмен значений с использованием рандеву

```
module Exchange[i = 1 to 2]
  op deposit(int);
body
  process Worker {
    int myvalue, othervalue;
    if (i == 1) {    # один процесс вызывает
      call Exchange[2].deposit(myvalue);
      in deposit(othervalue) -> skip; ni
    } else {         # другой процесс получает
      in deposit(othervalue) -> skip; ni
      call Exchange[1].deposit(myvalue);
    }
    ...
  }
end Exchange
```

8.3. Нотация совместно используемых примитивов

При использовании RPC и рандеву процесс инициирует взаимодействие, выполняя оператор `call`, который блокирует вызвавший процесс до того, как вызов будет обслужен и результаты возвращены. Такая последовательность действий идеальна для программирования взаимодействий типа "клиент-сервер", но, как видно из двух последних разделов, усложняет программирование фильтров и взаимодействующих равных. С другой стороны, односторонний поток информации между фильтрами и равными процессами легче программировать с помощью асинхронной передачи сообщений.

В данном разделе описана программная нотация, которая объединяет RPC, рандеву и асинхронную передачу сообщений в единое целое. Такая составная нотация (нотация совместно используемых примитивов) сочетает преимущества всех трех ее компонентов и обеспечивает дополнительные возможности.

8.3.1. Вызов и обслуживание операций

По своей структуре программа будет набором модулей. Видимые операции объявляются в области определений модуля. Эти операции могут вызываться процессами из других модулей, а обслуживаются процессом или процедурой модуля, в котором объявлены. Могут использоваться также *локальные* операции, которые объявляются, вызываются и обслуживаются в теле одного модуля.

В составной нотации операция может быть вызвана либо синхронным оператором `call`, либо асинхронным `send`. Они имеют следующий вид.

```
call Mname.opname(аргументы);
send Mname.opname(аргументы);
```

Оператор `call` завершается, когда операция обслужена и возвращены результирующие аргументы, а оператор `send` — как только вычислены аргументы. Если операция возвращает результат, ее можно вызывать в выражении; ключевое слово `call` опускается. Если операция имеет результирующие параметры и вызывается оператором `send`, или функция вызывается оператором `send`, или вызов функции находится не в выражении, то возвращаемое значение игнорируется.

В составной нотации операция может быть обслужена либо в процедуре (`proc`), либо с помощью рандеву (операторы `in`). Выбор — за программистом, который объявляет операцию в модуле. Это зависит от того, нужен ли программисту новый процесс для обслуживания вызова, или удобнее использовать рандеву с существующим процессом. Преимущества и недостатки каждого способа демонстрируются в дальнейших примерах.

Когда операцию обслуживает процедура (`proc`), для обработки вызова создается новый процесс. Вызов операции даст тот же результат, что и при использовании RPC. Если операция вызвана оператором `send`, результат будет тем же, что и при создании нового процесса, поскольку вызвавший операцию процесс продолжается асинхронно по отношению к процессу, обслуживающему вызов. В обоих случаях вызов обслуживается немедленно, и очереди ожидающих обработки вызовов нет.

Другой способ обслуживания операций состоит в использовании операторов ввода, которые имеют вид, указанный в разделе 8.2. С каждой операцией связана очередь ожидающих обработки вызовов, и доступ к этой очереди является неделимым. Выбор операции для обслуживания происходит в соответствии с семантикой операторов ввода. При вызове такой операции процесс приостанавливается, поэтому результат аналогичен использованию рандеву. Если такую операцию вызвать с помощью оператора `send`, результат будет аналогичным использованию асинхронной передачи сообщений, поскольку отправитель сообщения продолжает работу.

Итак, есть два способа вызова операции (операторы `call` и `send`) и два способа обслуживания вызова — `proc` и `in`. Эти четыре комбинации приводят к таким результатам.

Вызов	Обслуживание	Результат
call	proc	Вызов процедуры
call	in	Рандеву
send	proc	Динамическое создание процесса
send	in	Асинхронная передача сообщения

Если вызывающий процесс и процедура proc находятся в одном модуле, то вызов является локальным, иначе — удаленным. Операцию нельзя обслуживать как с помощью proc, так и в операторе in, поскольку тогда возникает неопределенность — обслужить операцию немедленно или поместить в очередь. Но операцию можно обслуживать в нескольких операторах ввода; они могут находиться в нескольких процессах модуля, в котором объявлена операция. В этом случае процессы совместно используют очередь ожидающих вызовов, но доступ к ней является неделимым.

Для мониторов и асинхронной передачи сообщений был определен примитив empty, который проверяет, есть ли объекты в канале сообщений или очереди условной переменной. В этой главе будет использован аналогичный, но несколько отличающийся примитив. Если opname является операцией, то ?opname — это функция, которая возвращает число ожидающих вызовов этой операции. Эту функцию удобно использовать в операторах ввода. Например, в следующем фрагменте кода операция op1 имеет приоритет перед операцией op2.

```
in op1(...) -> S1;
[] op2(...) and ?op1 == 0 -> S2;
ni
```

Условие синхронизации во второй защите разрешает выбор операции op2, только если при вычислении ?op1 определено, что вызовов операции op1 нет.

8.3.2. Примеры

Различные способы вызова и обслуживания операций проиллюстрируем тремя небольшими, тесно связанными примерами. Вначале рассмотрим реализацию очереди (листинг 8.11). Когда вызывается операция deposit, в buf помещается новый элемент. Если deposit вызвана оператором call, то вызывающий процесс ждет; если deposit вызвана оператором send, то вызывающий процесс продолжает работу (в этом случае процессу, вызывающему операцию, возможно, стоило бы убедиться, не переполнен ли буфер). Когда вызывается операция fetch, из массива buf извлекается элемент. Ее необходимо вызывать оператором call, иначе вызывающий процесс не получит результат.

Модуль Queue пригоден для использования одним процессом в другом модуле. Его не могут совместно использовать несколько процессов, поскольку в модуле нет критических секций для защиты переменных модуля. При параллельном вызове операций может возникнуть взаимное влияние.

Если нужна синхронизированная очередь, модуль Queue можно изменить так, чтобы он реализовывал кольцевой буфер. В листинге 8.5 был представлен именно такой модуль. Видимые операции в этом модуле те же, что и в модуле Queue. Но их вызовы обслуживаются оператором ввода в одном процессе, т.е. по одному. Операция fetch должна вызываться оператором call, однако для операции deposit вполне можно использовать оператор send.

Модули в листингах 8.5 и 8.11 демонстрируют два разных способа реализации одного и того же интерфейса. Выбор определяется программистом и зависит от того, как именно используется очередь. Но есть еще один способ реализации кольцевого буфера, иллюстрирующий еще одно сочетание различных способов вызова и обслуживания операций в нотации совместно используемых примитивов.

Листинг 8.11. Последовательная очередь

```
module Queue
  op deposit(typeT), fetch(result typeT);
body
  typeT buf[n];
  int front = 1, rear = 1, count = 0;

  proc deposit(item) {
    if (count < n) {
      buf[rear] = item;
      rear = (rear+1) mod n; count = count+1;
    } else
      предпринять действия, соответствующие переполнению;
  }

  proc fetch(item) {
    if (count > 0) {
      item = buf[front];
      front = (front+1) mod n; count = count-1;
    } else
      предпринять действия, соответствующие опустошению;
  }
end Queue
```

Рассмотрим следующий оператор ввода, который ждет вызова операции op, а затем присваивает параметры постоянным переменным:

```
in op(f1, ..., fn) -> v1 = f1; ...; vn = fn; ni
```

Результат *идентичен* действию оператора `receive`:

```
receive op(v1, ..., vn);
```

Поскольку оператор `receive` является просто сокращенной формой оператора `in`, будем использовать `receive`, когда нужно обработать вызов именно таким образом.

Теперь рассмотрим операцию, которая не имеет аргументов, вызывается оператором `send` и обслуживается оператором `receive` (или эквивалентным `in`). *Такая операция эквивалентна семафору*, причем `send` выступает в качестве `V`, а `receive` — `P`. Начальное значение семафора равно нулю. Его текущее значение — это число "пустых" сообщений, переданных операции, минус число полученных сообщений.

В листинге 8.12 представлена еще одна реализация модуля `BoundedBuffer`, в которой для синхронизации использованы семафоры. Операции `deposit` и `fetch` обслуживаются процедурами так же, как в листинге 8.11. Следовательно, одновременно может существовать несколько активных экземпляров этих процедур. Однако для реализации взаимного исключения и условной синхронизации в этой программе используются семафорные операции, как в листинге 4.5. Структура этого модуля аналогична структуре монитора (см. листинг 5.3), но синхронизация реализована с помощью семафоров, а не исключений монитора и условных переменных.

Листинг 8.12. Кольцевой буфер, использующий семафорные операции

```
module BoundedBuffer
  op deposit(typeT), fetch(result typeT);
body
  typeT buf[n];
  int front = 1, rear = 1;
  # локальные операции для имитации семафоров
  op empty(), full(), mutexD(), mutexF();
```

```
  send mutexD(); send mutexF();
  for [i = 1 to n]  # инициализировать пустой "семафор"
    send empty();

  proc deposit(item) {
    receive empty(); receive mutexD();
    buf[rear] = item; rear = (rear+1) mod n;
    send mutexD(); send full();
  }

  proc fetch(item) {
    receive full(); receive mutexF();
    item = buf[front]; front = (front+1) mod n;
    send mutexF(); send empty();
  }
end BoundedBuffer
```

Две реализации кольцевого буфера (см. листинги 8.5 и 8.11) иллюстрируют важную взаимосвязь между условиями синхронизации в операторах ввода и явной синхронизацией в процедурах. Во-первых, их часто используют с одинаковой целью. Во-вторых, поскольку условия синхронизации операторов ввода могут зависеть от аргументов ожидающих вызовов, эти два метода синхронизации обладают равной мощью. Но пока не нужна параллельность, которую обеспечивают несколько вызовов процедур, эффективнее использовать рандеву клиентов с одним процессом.

8.4. Новые решения задачи о читателях и писателях

С применением нотации совместно используемых примитивов можно программировать фильтры и взаимодействующие равные так же, как в главе 7. Поскольку эта нотация включает и RPC, и рандеву, можно программировать процессы-клиенты и процессы-серверы, как в разделах 8.1 и 8.2. Совместно используемые примитивы обеспечивают дополнительную гибкость, которая демонстрируется здесь. Сначала разработаем еще одно решение задачи о читателях и писателях (см. раздел 4.4). В отличие от предыдущих решений, здесь инкапсулирован доступ к базе данных. Затем расширим решение до распределенной реализации дублируемых файлов или баз данных. Обсудим также, как можно программировать эти решения, используя только RPC и локальную синхронизацию или только рандеву.

8.4.1. Инкапсулированный доступ

Напомним, что в задаче о читателях и писателях читательские процессы просматривают базу данных, возможно, параллельно. Процессы-писатели могут изменять базу данных, поэтому им нужен исключающий доступ к базе данных. В предыдущих решениях (с помощью семафоров в листингах 4.9 и 4.12 или с помощью мониторов в листинге 5.4) база данных была глобальной по отношению к процессам читателей и писателей, чтобы читатели могли обращаться к ней параллельно. При этом процессы должны были перед доступом к базе данных запрашивать разрешение, а после работы с ней освобождать доступ. Намного лучше инкапсулировать доступ к базе данных в модуле, чтобы спрятать протоколы запроса и освобождения, тем самым гарантируя их выполнение. Такой подход поддерживается в языке Java (это показано в конце раздела 5.4), поскольку программист может выбирать, будут методы (экспортируемые операции) выполняться параллельно или по одному. При использовании составной нотации решение можно структурировать аналогичным образом, причем полученное решение будет еще короче, а синхронизация — яснее.

В листинге 8.13 представлен модуль, инкапсулирующий доступ к базе данных. Клиентские процессы просто вызывают операции `read` или `write`, а вся синхронизация скрыта внутри модуля. В реализации модуля использованы и RPC, и рандеву. Операция `read` реализована в виде `proc`, поэтому несколько процессов-читателей могут выполняться параллельно, но операция `write` реализована на основе рандеву с процессом `Writer`, поэтому операции записи обслуживаются по очереди. Модуль также содержит две локальные операции, `startread` и `endread`, которые используются, чтобы обеспечить взаимное исключение операций чтения и записи. Их также обслуживает процесс `Writer`, поэтому он может отслеживать количество активных процессов-читателей и при необходимости откладывать выполнение операции `write`. Благодаря использованию рандеву в процессе `Writer` ограничения синхронизации выражаются непосредственно в виде логических условий, без обращений к условным переменным или семафорам. Отметим, что локальная операция `endread` вызывается оператором `send`, а не `call`, поскольку процесс-читатель не обязан ждать завершения обслуживания `endread`.

Листинг 8.13. Читатели и писатели с инкапсулированной базой данных

```
module ReadersWriters
  op read(result типы результатов);  # использует RPC
  op write(типы значений);           # использует рандеву
body
  op startread(), endread();         # локальные операции
  память для хранения буферов базы данных или передачи файлов;

  proc read(результаты) {
    call startread();    # получить разрешение на чтение
    читать из базы данных;
    send endread();      # освободить доступ
  }

  process Writer {
    int nr = 0;
    while (true) {
      in startread() -> nr = nr+1;
      [] endread() -> nr = nr-1;
      [] write(значения) and nr == 0 ->
          записать в базу данных;
      ni
    }
  }
end ReadersWriters
```

В языке, поддерживающем RPC, но не рандеву, этот модуль пришлось бы запрограммировать иначе. Например, обе операции `read` и `write` должны быть реализованы как экспортируемые процедуры, которые в свою очередь должны вызывать локальные процедуры для запроса разрешения на доступ к базе данных и для освобождения доступа. Внутренняя синхронизация должна обеспечиваться семафорами или мониторами.

Хуже, если язык поддерживает только рандеву. Операции обслуживаются существующими процессами, и каждый процесс может обслуживать одновременно только одну операцию. Таким образом, есть только один способ обеспечить параллельное чтение базы данных — экспортировать массив операций чтения, по одной для каждого клиента, и для их обслуживания использовать отдельные процессы. Такое решение по меньшей мере неуклюже и неэффективно.

Решение в листинге 8.13 отдает предпочтение читателям. Приоритет писателям можно дать, изменив оператор ввода с помощью функции `?` для приостановки вызовов операции `startread`, когда есть задержанные вызовы операции `write`.

```
in startread() and ?write == 0 -> nr = nr+1;
[] endread() -> nr = nr-1;
[] write(значения) and nr == 0 -> записать в базу данных;
ni
```

Разработка решения со справедливым планированием оставляется читателю.

Модуль в листинге 8.13 блокирует для читателей или для писателей всю базу данных. Обычно этого достаточно для небольших файлов данных. Однако для транзакций в базах данных обычно нужно блокировать отдельные записи по мере их обработки, поскольку заранее не известно, какие конкретно записи понадобятся в транзакции. Например, транзакции нужно просмотреть некоторые записи, чтобы определить, какие именно считывать далее. Для реализации такого вида динамической блокировки можно использовать много модулей, по одному на каждую запись. Но тогда не будет инкапсулирован доступ к базе данных. Вместо этого в модуле `ReadersWriters` можно применить более сложную схему блокировки с мелкомодульной структурой. Например, выделять необходимые блокировки для каждой операции `read` и `write`. Именно этот подход типичен для систем управления базами данных.

8.4.2. Дублируемые файлы

Простой способ повысить доступность файла с критическими данными — хранить его резервную копию на другом диске, обычно расположенном на другой машине. Пользователь может делать это вручную, периодически копируя файлы, либо файловая система будет автоматически поддерживать копию таких файлов. В любом случае при необходимости доступа к файлу пользователь должен сначала проверить доступность основной копии файла и, если она недоступна, использовать резервную копию. (С этой задачей связана проблема обновления основной копии, когда она вновь становится доступной.)

Третий подход состоит в том, что файловая система обеспечивает прозрачное копирование. Предположим, что есть n копий файла данных и n серверных модулей. Каждый сервер обеспечивает доступ к одной копии файла. Клиент взаимодействует с одним из серверных модулей, например, с тем, который выполняется на одном процессоре с клиентом. Серверы взаимодействуют между собой, создавая у клиентов иллюзию работы с файлом. На рис. 8.1 показана структура такой схемы взаимодействия.

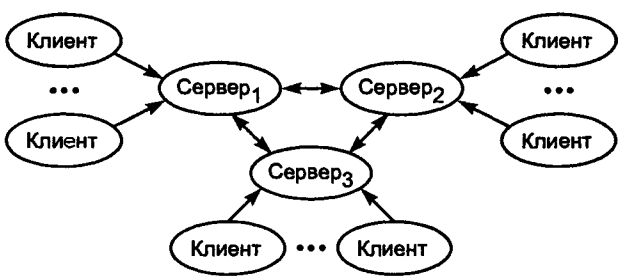


Рис. 8.1. Схема взаимодействия сервера дублируемого файла

Каждый модуль сервера экспортирует четыре операции: `open`, `close`, `read` и `write`. Желая обратиться к файлу, клиент вызывает операцию `open` своего сервера с указанием, собирается он записывать в файл или читать из него. Затем клиент общается с тем же сервером, вызывая операции `read` и `write`. Если файл был открыт для чтения, можно использовать только операции `read`, а если для записи — и `read`, и `write`. В конце концов клиент заканчивает диалог, вызвав операцию `close`. (Эта схема взаимодействия совпадает с приведенной в конце раздела 7.3.)

Файловые серверы взаимодействуют, чтобы поддерживать согласованность копий файла и не давать делать записи в файл одновременно нескольким процессам. В каждом файловом

сервере есть локальный процесс-диспетчер блокировок, реализующий решение задачи о читателях и писателях. Когда клиент открывает файл для чтения, операция open вызывает операцию startread локального диспетчера блокировок. Но когда клиентский процесс открывает файл для записи, операция open вызывает startwrite для всех n диспетчеров блокировок.

В листинге 8.14 представлен модуль FileServer. Для простоты использован массив и статическое именование, хотя на практике серверы должны создаваться динамически и размещаться на разных процессорах. Так или иначе, в реализации этого модуля есть несколько интересных аспектов.

Листинг 8.14. Дублируемые файлы с использованием блокировок для каждой копии

```
module FileServer[myid = 1 to n]
  type mode = (READ, WRITE);
  op open(mode), close(),          # клиентские операции
     read(result типы результатов), write(типы значений);
  op startwrite(), endwrite(),  # серверные операции
     remote_write(типы значений);
body
  op startread(), endread();    # локальные операции
  mode use; объявления для файловых буферов;

  proc open(m) {
    if (m == READ) {
      call startread();  # получить локальную блокировку чтения
      use = READ;
    } else {     # предполагается, что состояние - WRITE
      # получить блокировки записи для всех копий
      for [i = 1 to n]
        call FileServer[i].startwrite();
      use = WRITE;
    }
  }

  proc close() {
   if (use == READ) # освободить локальную блокировку чтения
      send endread();
   else # use == WRITE, поэтому освободить все блокировки записи
      for [i = 1 to n]
        send FileServer.endwrite()
  }

  proc read(результаты) {
    читать из локальной копии файла и возвратить результаты;
  }

  proc write(значения) {
    if (use == READ)
      возвратиться с ошибкой: файл не был открыт для записи;
    записать значения в локальную копию файла;
    # параллельно обновить все удаленные копии
    co [i = 1 to n st i != myid]
      call FileServer[i].remote_write(значения);
  }

  proc remote_write(значения) { # вызывается другими серверами
```

```
    записать значения в локальную копию файла;
  }

  process Lock {
    int nr = 0, nw = 0;
    while (true) {
      ## RW: (nr == 0 ∨ nw == 0) ∧ nw <= 1
      in startread() and nw == 0 -> nr = nr+1;
      [] endread() -> nr = nr-1;
      [] startwrite() and nr == 0 and nw == 0 ->
            nw = nw+1;
      [] endwrite() -> nw = nw-1;
      ni
    }
  }
end FileServer
```

- Каждый модуль `FileServer` экспортирует два набора операций: вызываемые его клиентами и вызываемые другими файловыми серверами. Операции `open`, `close`, `read` и `write` реализованы процедурами, но `read` — единственная, которая должна быть реализована процедурой, чтобы допустить параллельное чтение. Операции блокировки реализованы с помощью рандеву с диспетчером блокировок.
- Каждый модуль следит за текущим режимом доступа (каким образом файл был открыт последний раз), чтобы не разрешать записывать в файл, открытый для чтения, и определять, какие действия нужно выполнить при закрытии файла. Но модуль не защищен от клиента, получившего доступ к файлу, предварительно не открыв его. Эту проблему можно решить с помощью динамического именования или дополнительных проверок в других операциях.
- В процедуре `write` модуль сначала обновляет свою локальную копию файла, а затем параллельно обновляет все удаленные копии. Это аналогично использованию стратегии сквозного обновления кэш-памяти. Вместо этого можно использовать стратегию с обратной записью, при которой операция `write` обновляет только локальную копию, а удаленные копии файла обновляются при его закрытии.
- В операции `open` клиент получает блокировки записи, по одной от каждого блокирующего процесса. Чтобы не возникла взаимоблокировка клиентов, все клиенты получают блокировку в одном и том же порядке. В операции `close` клиент освобождает блокировки записи с помощью оператора `send`, а не `call`, поскольку процессу не нужно ждать освобождения блокировок.
- Диспетчер блокировок реализует решение задачи о читателях и писателях с классическим предпочтением для читателей. Это активный монитор, поэтому инвариант его цикла тот же, что инвариант монитора в листинге 5.4.

В программе в листинге 8.14 читатель для чтения файла должен получить только одну блокировку чтения. Писатель же должен получить все n блокировок записи, по одной от каждого экземпляра модуля `FileServer`. В обобщении этой схемы нужно использовать так называемое *взвешенное голосование.*

Пусть `readWeight` — это число блокировок, необходимых для чтения файла, а `writeWeight` — для записи в файл. В нашем примере `readWeight` равно 1, а `writeWeight` — n. Можно было бы использовать другие значения — для `readWeight` значение 2, а для `writeWeight` — `n-2`. Это означало бы, что читатель должен получить две блокировки чтения и `n-2` блокировок записи. Можно использовать любые весовые значения, лишь бы выполнялись следующие условия.

```
writeWeight > n/2 и
(readWeight + writeWeight) > n
```

При использовании взвешенного голосования, когда писатель закрывает файл, нужно обновить только копии, заблокированные для записи. Но каждая копия должна иметь метку времени последней записи в файл. Первое указанное выше требование гарантирует, что самые свежие данные и последние метки времени будут по меньшей мере у половины копий. (Как реализовать глобальные часы и метки времени последнего доступа к файлам, описывается в разделе 9.4.)

Открывая файл и получая блокировки чтения, читатель также должен прочитать метки времени последних изменений в каждой копии файла и использовать копию с самой последней меткой. Второе из указанных выше требований гарантирует, что будет хотя бы одна копия с самым последним временем изменений и, следовательно, с самыми последними данными. В нашей программе в листинге 8.14 не нужно было заботиться о метках времени, поскольку при закрытии файла обновлялись все его копии.

8.5. Учебные примеры: язык Java

В разделе 5.4 был представлен язык Java и программы для чтения и записи простой разделяемой базы данных. В разделе 7.9 было показано, как создать приложение типа "клиент-сервер", используя передачу сообщений через сокеты и модуль `java.net`. Язык Java также поддерживает RPC для распределенных программ. Поскольку операции с объектами языка Java называются методами, а не процедурами, RPC в языке Java называется *удаленным вызовом метода* (remote method invocation — RMI). Он поддерживается модулями `java.rmi` и `java.rmi.server`.

Ниже приведен обзор удаленного вызова методов, а его использование иллюстрируется реализацией простой удаленной базы данных. База данных имеет тот же интерфейс, что и в предыдущих примерах на языке Java из главы 5, но в ее реализации использованы клиент и сервер, которые выполняются на разных машинах.

8.5.1. Удаленный вызов методов

Приложение, использующее удаленный вызов методов, состоит из трех компонентов: интерфейса, в котором объявлены заголовки удаленных методов, серверного класса, реализующего этот интерфейс, и одного или нескольких клиентов, которые вызывают удаленные методы. Приложение программируется следующим образом.

- Пишется интерфейс Java, который расширяет интерфейс `Remote`, определенный в модуле `java.rmi`. Для каждого метода интерфейса нужно объявить, что он возбуждает удаленные исключительные ситуации.
- Строится серверный класс, расширяющий класс `UniclassRemoteObject` и реализующий методы интерфейса. (Сервер, конечно же, может содержать также защищенные поля и методы.) Пишется код, который создает экземпляр сервера и регистрирует его имя с помощью *службы регистрации* (см. ниже). Этот код может находиться в главном методе серверного класса или в другом классе.
- Пишется класс клиента, взаимодействующий с сервером. Клиент должен сначала установить диспетчер безопасности RMI, чтобы защитить себя от ошибочного кода *заглушки* (stub code, см. ниже), который может быть загружен через сеть. После этого клиент вызывает метод `Naming.lookup`, чтобы получить объект-сервер от службы регистрации. Теперь клиент может вызывать удаленные методы сервера.

Как компилировать и выполнять эти компоненты, показано ниже. Конкретный пример рассматривается в следующем разделе.

Программа-клиент вызывает методы сервера так, как если бы они были локальными (на той же виртуальной Java-машине). Но когда вызываются удаленные методы, взаимодействие

между клиентом и сервером в действительности управляется серверной *заглушкой* и *скелетом* сервера. Они создаются после компиляции программы при выполнении команды `rmic`. Заглушка и скелет — это части кода, которые включаются в программу автоматически. Они находятся между реальным клиентом и кодом сервера в исходной Java-программе. Когда клиент вызывает удаленный метод, он в действительности вызывает метод в заглушке. Заглушка *упорядочивает* аргументы удаленного вызова, собирая их в единое сообщение, и отсылает по сети скелету. Скелет получает сообщение с аргументами, генерирует локальный вызов метода сервера, ждет результатов и отсылает их назад заглушке. Наконец заглушка возвращает результаты коду клиента. Таким образом, заглушки и скелет скрывают подробности сетевого взаимодействия.

Еще одно следствие использования удаленных методов состоит в том, что клиент и сервер являются отдельными программами, которые выполняются на разных узлах сети. Следовательно, им нужен способ именования друг друга, причем имена серверов должны быть уникальными, поскольку одновременно могут работать многие серверы. По соглашению удаленные серверы именуются с помощью схемы URL. Имя имеет вид: `rmi://hostname:port/pathname`, где `hostname` — доменное имя узла, на котором будет выполняться сервер, `port` — номер порта, а `pathname` — путевое имя на сервере, выбираемые пользователем.

Служба регистрации — это специализированная программа, которая управляет списком имен серверов, зарегистрированных на узле. Она запускается на серверной машине в фоновом режиме командой "`rmiregistry port &`". (Программа-сервер может также обеспечивать собственную службу регистрации, используя модуль `java.rmi.registry`.) Интерфейс для службы регистрации обеспечивается объектом `Naming`; основные методы этого объекта — `bind` для регистрации имени и сервера и `lookup` для поиска сервера, связанного с именем.

Последний шаг в выполнении программы типа "клиент-удаленный сервер" — запуск сервера и клиента (клиентов) с помощью интерпретатора `java`. Сначала запускается сервер на машине `hostname`. Клиент запускается на любом узле, подключенном к серверу. Но тут есть два предостережения: пользователь должен иметь разрешение на чтение файлов `.class` языка Java на обеих машинах, а на серверной машине должны быть разрешены удаленные вызовы с клиентских машин.

8.5.2. Пример: удаленная база данных

В листинге 8.15 представлена законченная программа для простого, но интересного примера использования удаленного вызова методов. Интерфейс `RemoteDatabase` определяет методы `read` и `write`, реализуемые сервером. Эти методы будут выполняться удаленно по отношению к клиенту, поэтому они объявлены как возбуждающие исключительную ситуацию `RemoteException`.

Листинг 8.15. Интерфейс удаленной базы данных, клиент и сервер

```
import java.rmi.*;
import java.rmi.server.*;

public interface RemoteDatabase extends Remote {
 public int read() throws RemoteException;
 public void write(int value) throws RemoteException;
}

class Client {
 public static void main(String[] args) {
  try {
   // установить стандартный диспетчер безопасности RMI
   System.setSecurityManager(new RMISecurityManager());
   // получить объект удаленной базы данных
```

```
    String name =
        "rmi://paloverde:9999/database";
    RemoteDatabase db =
        (RemoteDatabase) Naming.lookup(name);
    // прочитать аргумент командной строки и
    // получить доступ к базе данных
    int value, rounds = Integer.parseInt(args[0]);
    for (int i = 0; i < rounds; i++) {
      value = db.read();
      System.out.println("read: " + value);
      db.write(value+1);
    }
   }
   catch (Exception e) {
     System.err.println(e);
   }
  }
}

class RemoteDatabaseServer extends UnicastRemoteObject
          implements RemoteDatabase {
 protected int data = 0;  // "база данных"

 public int read() throws RemoteException {
   return data;
 }

 public void write(int value) throws RemoteException {
   data = value;
   System.out.println("new value is: " + data);
 }

 // необходим конструктор, поскольку есть часть throws
 public RemoteDatabaseServer() throws RemoteException {
   super();
 }

 public static void main(String[] args) {
  try {
   // создать объект сервера удаленной базы данных
   RemoteDatabaseServer server =
       new RemoteDatabaseServer();
   // зарегистрировать имя и начать обслуживание
   String name =
       "rmi://paloverde:9999/database";
   Naming.bind(name, server);
   System.out.println(name + " is running");
  }
  catch (Exception e) {
    System.err.println(e);
  }
 }
}
```

Класс `Client` определяет клиентскую часть программы. Он состоит из метода `main`, который можно запускать на любой машине. Клиент сначала устанавливает диспетчер безопасности RMI для защиты от ошибочного кода заглушки сервера. Затем клиент ищет имя сервера и получает ссылку на сервер. В остальной части программы клиента циклически вызыва-

ются методы `read` и `write` сервера. Число `round` повторений цикла задается как аргумент в командной строке при запуске клиента.

Серверный класс `RemoteDatabaseServer` реализует интерфейс сервера. Сама "база данных" представляет собой просто защищенную целочисленную переменную. Метод `main` сервера создает экземпляр сервера, регистрирует его имя и, чтобы показать, что сервер работает, выводит строку на терминал узла сервера. Сервер работает, пока его процесс не уничтожат. В программе используются такие имя и номер порта сервера: `paloverde:9999` (рабочая станция автора).

Обе части программы должны быть записаны в одном файле, который называется `RemoteDatabase.java`, поскольку это имя интерфейса. Чтобы откомпилировать программу, создать заглушки, запустить службу регистрации и сервер, на узле `paloverde` нужно выполнить следующие команды.

```
javac RemoteDatabase.java
rmic RemoteDatabaseServer
rmiregistry 9999 &
java RemoteDatabaseServer
```

Клиентская программа запускается на машине `paloverde` или на другой машине той же сети с помощью такой команды.

```
java Client rounds
```

Читателю рекомендуется поэкспериментировать с этой программой, чтобы понять, как она себя ведет. Например, что происходит, если выполняются несколько клиентов или если запустить клиентскую программу до запуска сервера?

8.6. Учебные примеры: язык Ada

Язык Ada был создан при содействии министерства обороны США в качестве стандартного языка программирования приложений для обороны (от встроенных систем реального времени до больших информационных систем). Возможности параллелизма языка Ada являются его важной частью; они критичны для его использования по назначению. В языке Ada также есть большой набор механизмов для последовательного программирования.

Язык Ada стал результатом широкого международного конкурса разработок в конце 1970-х годов и впервые был стандартизован в 1983 г. В языке Ada 83 был представлен механизм рандеву для межпроцессного взаимодействия. Сам термин *рандеву* был выбран потому, что руководителем группы разработчиков был француз. Вторая версия языка Ada была стандартизована в 1995 г. Язык Ada 95 совместим снизу вверх с языком Ada 83 (поэтому старые программы остались работоспособными), но в нем появилось несколько новых свойств. Два самых интересных свойства, связанных с параллельным программированием, — это защищенные типы, подобные мониторам, и оператор `requeue`, позволяющий программисту более полно управлять синхронизацией и планированием.

В данном разделе сначала представлен обзор основных механизмов параллельности в языке Ada: задачи, рандеву и защищенные типы. Далее показано, как запрограммировать барьер в виде защищенного типа, а решение задачи об обедающих философах — в виде набора задач, которые взаимодействуют с помощью рандеву. В примерах также демонстрируются возможности языка Ada для последовательного программирования.

8.6.1. Задачи

Программа на языке Ada состоит из подпрограмм, модулей (*пакетов*) и задач. Подпрограмма — это процедура или функция, пакет — набор деклараций, а задача — независимый процесс. Каждый компонент имеет раздел определений и тело. Определения объявляют ви-

димые объекты, тело содержит локальные декларации и операторы. Подпрограммы и модули могут быть настраиваемыми (generic), т.е. параметризоваться типами данных.

Базовая форма определения задачи имеет следующий вид.

```
task Name is
  декларации точек входа;
end;
```

Декларации точек входа аналогичны декларациям op в модулях. Они определяют операции, которые обслуживает задача, и имеют такой вид.

```
entry Identifier(параметры);
```

Параметры передаются путем копирования в подпрограмму (по умолчанию), копирования из подпрограммы или обоими способами. Язык Ada поддерживает массивы точек входа, которые называются *семействами точек входа.*

Базовая форма тела задачи имеет следующий вид.

```
task body Name is
  локальные декларации
begin
  операторы;
end Name;
```

Задача должна быть объявлена внутри подпрограммы или пакета. Простейшая параллельная программа на языке Ada является, таким образом, процедурой, содержащей определения задач и их тел*а*. Объявления в любом компоненте *обрабатываются* по одному в порядке их появления. Обработка объявления задачи приводит к созданию экземпляра задачи. После обработки всех объявлений начинают выполняться последовательные операторы подпрограммы в виде безымянной задачи.

Пара "определение задачи-тело" определяет одну задачу. Язык Ada также поддерживает массивы задач, но способ поддержки не такой, как в других языках программирования. Программист сначала объявляет тип задачи, а затем — массив экземпляров этого типа. Для динамического создания задач программист может использовать типы задач совместно с указателями (в языке Ada они называются *типами доступа*).

8.6.2. Рандеву

В языке Ada 83 первичным механизмом взаимодействия и единственной схемой синхронизации было рандеву. (Задачи на одной машине могли также считывать и записывать значения разделяемых переменных.) Все остальные схемы взаимодействия нужно было программировать с помощью рандеву. Язык Ada 95 также поддерживает защищенные типы для синхронизированного доступа к разделяемым объектам; это описано в следующем разделе.

Предположим, что в задаче T объявлена точка входа E. Задачи из области видимости задачи T могут вызвать E следующим образом.

```
call T.E(аргументы);
```

Как обычно, выполнение call приостанавливает работу вызывающего процесса до тех пор, пока не завершится E (будет уничтожена или вызовет исключение).

Задача T обслуживает вызовы точки входа E с помощью оператора accept, имеющего следующий вид.

```
accept E(параметры) do
  список операторов;
end;
```

Выполнение оператора accept приостанавливает задачу, пока не появится вызов E, копирует значения входных аргументов во входные параметры и выполняет список операторов. Когда выполнение списка операторов завершается, значения выходных параметров копируются

в выходные аргументы. В этот момент продолжают работу и процесс, вызвавший точку входа, и процесс, выполняющий ее. Оператор `accept`, таким образом, похож на оператор ввода (раздел 8.2) с одной защитой, без условия синхронизации и выражения планирования.

Для поддержки недетерминированного взаимодействия задач в языке Ada используются операторы `select` трех типов: селективное ожидание, условный вызов точки входа и синхронизированный вызов точки входа. Оператор селективного ожидания поддерживает защищенное взаимодействие. Его обычная форма такова.

```
select when B1 => accept оператор; дополнительные операторы;
or ...
or     when Bn => accept оператор; дополнительные операторы;
end select;
```

Каждая строка называется *альтернативой*. Каждое B_i — это логическое выражение, части `when` необязательны. Говорят, что альтернатива *открыта*, если условие B_i истинно или часть `when` отсутствует.

Эта форма селективного ожидания приостанавливает выполняющий процесс до тех пор, пока не сможет выполниться оператор `accept` в одной из открытых альтернатив, т.е. если есть ожидающий вызов точки входа, указанной в операторе `accept`. Поскольку каждая защита B_i предшествует оператору `accept`, защита *не может* ссылаться на параметры вызова точки входа. Также язык Ada не поддерживает выражения планирования. Как отмечалось в разделе 8.2 и будет видно из примеров следующих двух разделов, это усложняет решение многих задач синхронизации и планирования.

Оператор селективного ожидания может содержать необязательную альтернативу `else`, которая выбирается, если нельзя выбрать ни одну из остальных альтернатив. Вместо оператора `accept` программист может использовать оператор `delay` или альтернативу `terminate`. Открытая альтернатива с оператором `delay` выбирается, если истек интервал ожидания; этим обеспечивается механизм управления временем простоя. Альтернатива `terminate` выбирается, если завершились или ожидают в своих альтернативах `terminate` все задачи, которые взаимодействуют с помощью рандеву с данной задачей (см. пример в листинге 8.18).

Условный вызов точки входа используется, если одна задача должна опросить другую. Он имеет такой вид.

```
select вызов точки входа; дополнительные операторы;
else   операторы;
end select;
```

Вызов точки входа выбирается, если его можно выполнить немедленно, иначе выбирается альтернатива `else`.

Синхронизированный вызов точки входа используется, когда вызывающая задача должна ожидать не дольше заданного интервала времени. По форме такой вызов аналогичен условному вызову точки входа.

```
select вызов точки входа; дополнительные операторы;
or     delay оператор; дополнительные операторы;
end select;
```

Здесь выбирается вызов точки входа, если он может быть выполнен до истечения заданного интервала времени задержки.

Языки Ada 83 и Ada 95 обеспечивают несколько дополнительных механизмов для параллельного программирования. Задачи могут разделять переменные, однако обновление значений этих переменных гарантированно происходит только в точках синхронизации (например, в операторах рандеву). Оператор `abort` позволяет одной задаче прекращать выполнение другой. Существует механизм установки приоритета задачи. Кроме того, задача имеет так называемые *атрибуты*. Они позволяют определить, можно ли вызвать задачу, или она уже прекращена, а также узнать количество ожидающих вызовов точек входа.

8.6.3. Защищенные типы

Язык Ada 95 развил механизмы параллельного программирования языка Ada 83 по нескольким направлениям. Наиболее существенные дополнения — защищенные типы, которые поддерживают синхронизированный доступ к разделяемым данным, и оператор `requeue`, обеспечивающий планирование и синхронизацию в зависимости от аргументов вызова.

Защищенный тип инкапсулирует разделяемые данные и синхронизирует доступ к ним. Экземпляр защищенного типа аналогичен монитору, а его раздел определений имеет следующий вид.

```
protected type Name is
   декларации функций, процедур или точек входа;
private
   декларации переменных;
end Name;
```

Тело имеет такой вид.

```
protected body Name is
   тела функций, процедур или точек входа;
end Name;
```

Защищенные функции обеспечивают доступ только для чтения к скрытым переменным; следовательно, функцию могут вызвать одновременно несколько задач. Защищенные процедуры обеспечивают исключительный доступ к скрытым переменным для чтения и записи. Защищенные точки входа похожи на защищенные процедуры, но имеют еще часть `when`, которая определяет логическое условие синхронизации. Защищенная процедура или точка входа в любой момент времени может выполняться только для одной вызвавшей ее задачи. Вызов защищенной точки входа приостанавливается, пока условие синхронизации не станет истинным и вызывающая задача не получит исключительный доступ к скрытым переменным. Условие синхронизации не может зависеть от параметров вызова.

Вызовы защищенных процедур и точек входа обслуживаются в порядке FIFO, но в зависимости от условий синхронизации точек входа. Чтобы отложить завершение обслуживаемого вызова, в теле защищенной процедуры или точки входа можно использовать оператор `requeue`. (Его можно использовать и в теле оператора `accept`.) Он имеет следующий вид.

```
requeue Opname;
```

`Opname` — это имя точки входа или защищенной процедуры, которая или не имеет параметров или имеет те же параметры, что и обслуживаемая операция. В результате выполнения оператора `requeue` вызов помещается в очередь операции `Opname`, как если бы задача непосредственно вызвала операцию `Opname`.

В качестве примера использования защищенного типа и оператора `requeue` рассмотрим код `N`-задачного барьера-счетчика в листинге 8.16. Предполагается, что `N` — глобальная константа. Экземпляр барьера объявляется и используется следующим образом.

```
B : Barrier;   -- декларация барьера
  ...
B.Arrive;      -- или "call B.Arrive;"
```

Первые `N-1` задач, подходя к барьеру, увеличивают значение счетчика барьера `count` и задерживаются в очереди на скрытой точке входа `Go`. Последняя прибывшая к барьеру задача присваивает переменной `time_to_leave` значение `True`; это позволяет запускать по одному процессы, задержанные в очереди операции `Go`. Каждая задача перед выходом из барьера уменьшает значение `count`, а последняя уходящая задача переустанавливает значение переменной `time_to_leave`, поэтому барьер можно использовать снова. (Семантика защищенных типов гарантирует, что каждая приостановленная в очереди `Go` задача будет выполняться до обслуживания любого последующего вызова процедуры `Arrive`.) Читателю полезно сравнить этот барьер с барьером в листинге 5.12, который запрограммирован с использованием библиотеки Pthreads.

Листинг 8.16. Барьерная синхронизация на языке Ada

```
protected type Barrier is
  procedure Arrive;
private
  entry Go;  -- используется для приостановки первых прибывших
  count : Integer := 0; -- число прибывших
  time_to_leave : Boolean := False;
end Barrier;

protected body Barrier is
  procedure Arrive is begin
    count := count+1;
    if count < N then
      requeue Go;    -- ждать остальных
    else
      count := count-1; time_to_leave := True;
    end if;
  end;

  entry Go when time_to_leave is begin
    count := count-1;
    if count = 0 then time_to_leave := False; end if;
  end;
end Barrier;
```

8.6.4. Пример: обедающие философы

В данном разделе представлена законченная Ada-программа для задачи об обедающих философах (см. раздел 4.3). Программа иллюстрирует использование как задач и рандеву, так и некоторых общих свойств языка Ada. Для удобства предполагается, что существуют две функции `left(i)` и `right(i)`, которые возвращают индексы соседей философа `i` слева и справа.

В листинге 8.17 представлена главная процедура `Dining_Philosophers`. Перед процедурой находятся декларации `with` и `use`. В декларации `with` сообщается, что эта процедура использует объекты пакета `Ada.Text_IO`, а `use` делает имена экспортируемых объектов этого пакета непосредственно видимыми (т.е. их не нужно уточнять именем пакета).

Листинг 8.17. Решение задачи об обедающих философах на языке Ada: главная программа[15]

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Dining_Philosophers is
  subtype ID is Integer range 1..5;

  task Waiter is          -- спецификация задачи-официанта
    entry Pickup(I : in ID);
    entry Putdown(I : in ID);
  end
  task body Waiter is separate;

  task type Philosopher is    -- тип задачи-философа
```

[15] В теле задачи-философа отсутствует имитация случайных промежутков времени, в течение которых философы едят или думают. — *Прим. ред.*

```
    entry init(who : ID);
  end;

  DP : array(ID) of Philosopher;  -- философы
  rounds : Integer;               -- число циклов

  task body Philosopher is   -- тело задачи-философа
    myid : ID;
  begin
    accept init(who); myid := who; end;
    for j in 1..rounds loop
      -- "подумать"
      Waiter.Pickup(myid);  -- взять вилки
      -- "поесть"
      Waiter.Putdown(myid); -- положить вилки
    end loop;
  end Philosopher;

begin  -- считать число циклов и запустить задачи-философы
  Get(rounds);
  for j in ID loop
    DP(j).init(j);
  end loop;
end Dining_Philosophers;
```

В главной процедуре сначала объявлен тип `ID` с целыми значениями от 1 до 5. В спецификации задачи `Waiter` объявлены два входа `Pickup` и `Putdown`, вызываемых философами, чтобы взять (pick up) или положить (put down) вилку. Тело задачи `Waiter` компилируется отдельно; оно представлено в листинге 8.18.

Листинг 8.18. Решение задачи об обедающих философах на языке Ada: задача-официант

```
separate (Dining_Philosophers)
task body Waiter is
  entry Wait(ID);  -- для постановки философов в очередь
  eating : array (ID) of Boolean; -- кто занят едой
  want : array (ID) of Boolean;   -- кто хочет есть
begin
  for j in ID loop           -- инициализация массивов
    eating(j) := False; want(j) := False;
  end loop;
  loop             -- основной цикл сервера
   select
     accept Pickup(i : in ID) do  -- вилки нужны для DP(i)
       if not(eating(left(i)) or eating(right(i))) then
         eating(i) := True;
       else
         want(i) := True; requeue Wait(i);
       end if;
     end;
   or
     accept Putdown(i : in ID) do  -- DP(i) закончил есть
       eating(i) := False;
     end;
     -- проверить, могут ли теперь есть соседи
     if want(left(i)) and not eating(left(left(i)))
```

```
        then accept Wait(left(i));
        eating(left(i)) := True; want(left(i)) := False;
      end if;
      if want(right(i)) and not eating(right(right(i)))
        then accept Wait(right(i));
        eating(right(i)) := True; want(right(i)) := False;
      end if;
    or
      terminate;  -- выход, когда философы закончили
    end select;
  end loop;
end Waiter;
```

В листинге 8.17 имя Philosopher определено как тип задачи, чтобы можно было объявить массив DP из пяти таких задач. Экземпляры пяти задач-философов создаются при обработке декларации массива DP. Каждый философ сначала ждет, чтобы принять вызов своей инициализирующей точки входа init, затем выполняет rounds итераций. Переменная rounds объявлена глобальной по отношению к телу задач философов, поэтому все они могут ее читать. Переменная rounds инициализируется вводимым значением в теле главной процедуры (конец листинга 8.17). Затем каждому философу передается его индекс с помощью вызова DP(j).init(j).

Листинг 8.18 содержит тело задачи Waiter (официант). Оно сложнее, чем процесс Waiter в листинге 8.6, поскольку условие when в операторе select языка Ada не может ссылаться на входные параметры. Waiter многократно принимает вызовы операций Pickup и Putdown. Принимая вызов Pickup, официант проверяет, ест ли хотя бы один сосед философа i, вызвавшего эту операцию. Если нет, то философ i может есть. Но если хотя бы один сосед ест, то вызов Pickup должен быть вновь поставлен в очередь так, чтобы задача-философ не была слишком рано запущена вновь. Для приостановки ожидающих философов используется локальный массив из пяти точек входа Wait(ID); каждый философ ставится в очередь отдельного элемента этого массива.

Поев, философ вызывает операцию Putdown. Принимая этот вызов, официант проверяет, хочет ли есть каждый сосед данного философа и может ли он приступить к еде. Если да, официант принимает отложенный вызов операции Wait, чтобы запустить задачу-философ, вызов операции Pickup которой был поставлен в очередь. Оператор accept, обслуживающий операцию Putdown, мог бы охватывать всю альтернативу в операторе select, т.е. заканчиваться после двух операторов if. Однако он заканчивается раньше, поскольку незачем приостанавливать задачу-философ, вызвавшую операцию Putdown.

8.7. Учебные примеры: язык SR

Язык синхронизирующих ресурсов (synchronizing resources — SR) был создан в 1980-х годах. В его первой версии был представлен механизм рандеву (см. раздел 8.2), затем он был дополнен поддержкой совместно используемых примитивов (см. раздел 8.3). Язык SR поддерживает разделяемые переменные и распределенное программирование, его можно использовать для непосредственной реализации почти всех программ из данной книги. SR-программы могут выполняться на мультипроцессорах с разделяемой памятью и в сетях рабочих станций, а также на однопроцессорных машинах.

Хотя язык SR содержит множество различных механизмов, все они основаны на нескольких ортогональных концепциях. Последовательный и параллельный механизмы объединены,

чтобы похожие результаты достигались аналогичными способами. В разделах 8.2 и 8.3 уже были представлены и продемонстрированы многие аспекты языка SR, хотя явно об этом не говорилось. В данном разделе описываются такие вопросы, как структура программы, динамическое создание и размещение, дополнительные операторы. В качестве примера представлена программа, имитирующая выполнение процессов, которые входят в критические секции и выходят из них.

8.7.1. Ресурсы и глобальные объекты

Программа на языке SR состоит из ресурсов и глобальных компонентов. *Декларация ресурса* определяет схему модуля и имеет почти такую же структуру, как и `module`.

```
resource name     # раздел описаний
  описания импортируемых объектов
  декларации операций и типов
body name(параметры)   # тело
  декларации переменных и других локальных объектов
  код инициализации
  процедуры и процессы
  код завершения
end name
```

Декларация ресурса содержит описания импортируемых объектов, если ресурс использует декларации, экспортируемые другими ресурсами или глобальными компонентами. Декларации и код инициализации в теле могут перемежаться; это позволяет использовать динамические массивы и управлять порядком инициализации переменных и создания процессов. Раздел описаний может быть опущен. Раздел описаний и тело могут компилироваться отдельно.

Экземпляры ресурса создаются динамически с помощью оператора `create`. Например, код

```
rcap := create name(аргументы)
```

передает аргументы (по значению) новому экземпляру ресурса `name` и затем выполняет код инициализации ресурса. Когда код инициализации завершается, возвращается *мандат доступа к ресурсу*, который присваивается переменной `rcap`. В дальнейшем эту переменную можно использовать для вызова операций, экспортируемых ресурсом, или для уничтожения экземпляра ресурса. Ресурсы уничтожаются динамически с помощью оператора `destroy`. Выполнение `destroy` останавливает любую работу в указанном ресурсе, выполняет его код завершения (если есть) и освобождает выделенную ему память.

По умолчанию компоненты SR-программы размещаются в одном адресном пространстве. Оператор `create` можно также использовать для создания дополнительных адресных пространств, которые называются *виртуальными машинами*.

```
vmcap := create vm() on machine
```

Этот оператор создает на указанном узле виртуальную машину и возвращает мандат доступа к ней. Последующие операторы создания ресурса могут использовать конструкцию "`on vmcap`", чтобы размещать новые ресурсы в этом адресном пространстве. Таким образом, язык SR, в отличие от Ada, дает программисту полный контроль над распределением ресурсов по машинам, которое может зависеть от входных данных программы.

Для объявления типов, переменных, операций и процедур, разделяемых ресурсами, применяется *глобальный компонент*. Это, по существу, одиночный экземпляр ресурса. На каждой виртуальной машине, использующей глобальный компонент, хранится одна его копия. Для

этого при создании ресурса неявно создаются все глобальные объекты, из которых он импортирует (если они не были созданы ранее).

SR-программа содержит один отдельный главный ресурс. Выполнение программы начинается с неявного создания одного экземпляра этого ресурса. Затем выполняется код инициализации главного ресурса; обычно он создает экземпляры других ресурсов.

SR-программа завершается, когда завершаются или блокируются все процессы, или когда выполняется оператор `stop`. В этот момент система поддержки программы (run-time system) выполняет код завершения (если он есть) главного ресурса и затем коды завершения (если есть) глобальных компонентов. Это обеспечивает программисту управление, чтобы, например, напечатать результаты или данные о времени работы программы.

В качестве простого примера приведем SR-программу, которая печатает две строки.

```
resource silly()
  write("Hello world.")
  final
    write("Goodbye world.")
  end
end
```

Этот ресурс создается автоматически. Он выводит строку и завершается. Тогда выполняется код завершения, выводящий вторую строку. Результат будет тем же, если убрать слова `final` и первое `end`.

8.7.2. Взаимодействие и синхронизация

Отличительным свойством языка SR является многообразие механизмов взаимодействия и синхронизации. Переменные могут разделяться процессами одного ресурса, а также ресурсами в одном адресном пространстве (с помощью глобальных компонентов). Процессы также могут взаимодействовать и синхронизироваться, используя все примитивы, описанные в разделе 8.3, — семафоры, асинхронную передачу сообщений, RPC и рандеву. Таким образом, язык SR можно использовать для реализации параллельных программ как на мультипроцессорных машинах с разделяемой памятью, так и в распределенных системах.

Декларации операций начинаются ключевым словом `op`; их вид уже был представлен в данной главе. Такие декларации можно записывать в разделе описаний ресурса, в теле ресурса и даже в процессах. Операция, объявленная в процессе, называется *локальной*. Процесс, который объявляет операцию, может передавать мандат доступа к локальной операции другому процессу, позволяя ему вызывать эту операцию. Эта возможность поддерживает непрерывность диалога (см. раздел 7.3).

Операция вызывается с помощью синхронного оператора `call` или асинхронного `send`. Для указания вызываемой операции оператор вызова использует мандат доступа к операции или поле мандата доступа к ресурсу. Внутри ресурса, объявившего операцию, ее мандатом фактически является ее имя, поэтому в операторе вызова можно использовать непосредственно имя операции. Мандаты ресурсов и операций можно передавать между ресурсами, поэтому пути взаимодействия могут изменяться динамически.

Операцию обслуживает либо процедура (`proc`), либо оператор ввода (`in`). Для обслуживания каждого удаленного вызова `proc` создается новый процесс; вызовы в пределах одного адресного пространства оптимизируются так, чтобы тело процедуры выполнял процесс, который ее вызвал. Все процессы ресурса работают параллельно, по крайней мере, теоретически.

Оператор ввода `in` поддерживает рандеву. Его вид указан в разделе 8.2; он может содержать условия синхронизации и выражения планирования, зависящие от параметров. Оператор ввода может содержать необязательную часть `else`, которая выбирается, если не пропускает ни одна из защит.

Язык SR содержит несколько механизмов, являющихся сокращениями других конструкций. Декларация process — это сокращение декларации op и определения proc для обслуживания вызовов операции. Один экземпляр процесса создается неявным оператором send при создании ресурса. (Программист может объявлять и массивы процессов.) Декларация procedure также является сокращением декларации op и определения proc для обслуживания вызовов операции.

Еще два сокращения — это оператор receive и семафоры. Оператор receive выполняется так же, как оператор ввода, который обслуживает одну операцию и просто записывает значения аргументов в локальные переменные. Декларация семафора (sem) является сокращенной формой объявления операции без параметров. Оператор P представляет собой частный случай оператора receive, а V — оператора send.

Язык SR обеспечивает несколько дополнительных полезных операторов. Оператор reply — это вариант оператора return. Он возвращает значения, но выполняющий его процесс продолжает работу. Оператор forward можно использовать, передавая вызов другому процессу для последующего обслуживания. Он аналогичен оператору requeue языка Ada. Наконец, в языке SR есть оператор co для параллельного вызова операций.

8.7.3. Пример: моделирование критической секции

В листинге 8.19 приведена законченная программа, использующая несколько механизмов передачи сообщений языка SR. Программа также показывает, как разработать простую модель решения задачи (в данном случае — задачи критической секции).

Листинг 8.19. SR-программа, моделирующая критические секции

```
global CS
  op CSenter(id: int) {call}  # вызывается оператором call
  op CSexit()  # вызывается оператором call или send
body CS
  process arbitrator
    do true ->
      in CSenter(id) by id ->
          write("user", id, "in its CS at", age())
      ni
      receive CSexit()
    od
  end
end

resource main()
  import CS
  var numusers, rounds: int
  getarg(1, numusers); getarg(2, rounds)
  process user(i := 1 to numusers)
    fa j := 1 to rounds ->
      call CSenter(i)          # войти в критическую секцию
      nap(int(random(100)))    # ждать до 100 мс
      send CSexit()            # выйти из критической секции
      nap(int(random(1000)))   # ждать до 1 сек
    af
  end
end
```

Глобальный компонент CS экспортирует две операции: CSenter и CSexit. Тело CS содержит процесс arbitrator, реализующий эти операции. Для ожидания вызова операции CSenter в нем использован оператор ввода.

```
in CSenter(id) by id ->
    write("user", id, "in its CS at", age())
ni
```

Это механизм рандеву языка SR. Если есть несколько вызовов операции CSenter, то выбирается вызов с наименьшим значением параметра id, после чего печатается сообщение. Затем процесс arbitrator использует оператор receive для ожидания вызова операции CSexit. В этой программе процесс arbitrator и его операции можно было бы поместить внутрь ресурса main. Однако, находясь в глобальном компоненте, они могут использоваться другими ресурсами в большей программе.

Ресурс main считывает из командной строки два аргумента, после чего создает numusers экземпляров процесса user. Каждый процесс с индексом i выполняет цикл "для всех" (for all — fa), в котором вызывает операцию Csenter с аргументом i, чтобы получить разрешение на вход в критическую секцию. Длительность критической и некритической секций кода имитируется "сном" (nap) каждого процесса user в течение случайного числа миллисекунд. После "сна" процесс вызывает операцию CSexit. Операцию CSenter можно вызвать только синхронным оператором call, поскольку процесс user должен ожидать получения разрешения на вход в критическую секцию. Это выражено ограничением {call} в объявлении операции CSenter. Однако операцию CSexit можно вызывать асинхронным оператором send, поскольку процесс user может не задерживаться, покидая критическую секцию.

В программе использованы несколько предопределенных функций языка SR. Оператор write печатает строку, а getarg считывает аргумент из командной строки. Функция age в операторе write возвращает длительность работы программы в миллисекундах. Функция nap заставляет процесс "спать" в течение времени, заданного ее аргументом в миллисекундах. Функция random возвращает псевдослучайное число в промежутке от 0 до значения ее аргумента. Использована также функция преобразования типа int, чтобы преобразовать результат, возвращаемый функцией random, к целому типу, необходимому для аргумента функции nap.

Историческая справка

Удаленный вызов процедур (RPC) и рандеву появились в конце 1970-х годов. Исследования по семантике, использованию и реализации RPC были начаты и продолжаются разработчиками операционных систем. Нельсон (Bruce Nelson) провел много экспериментов по этой теме в исследовательском центре фирмы Xerox в Пало-Альто (PARC) и написал отличную диссертацию [Nelson, 1981]. Эффективная реализация RPC в ядре операционной системы представлена в работе [Birrell and Nelson, 1984]. Перейдя из PARC в Стэнфордский университет, Спектор [Alfred Spector, 1982] написал диссертацию по семантике и реализации RPC.

Бринч Хансен [Brinch Hansen, 1978] выдвинул основные идеи RPC (хотя и не дал этого названия) и разработал первый язык программирования, основанный на удаленном вызове процедур. Он назвал свой язык "Распределенные процессы" (Distributed Processes — DP). Процессы в DP могут экспортировать процедуры. Процедура, вызванная другим процессом, выполняется в новом потоке управления. Процесс может также иметь один "фоновый" поток, который выполняется первым и может продолжать работу в цикле. Потоки в процессе

выполняются со взаимным исключением. Они синхронизируются с помощью разделяемых переменных и оператора when, аналогичного оператору `await` (глава 2).

RPC был включен в некоторые другие языки программирования, такие как Cedar, Eden, Emerald и Lynx. На RPC основаны языки Argus, Aeolus, Avalon и другие. В этих трех языках RPC объединен с так называемыми *неделимыми транзакциями*. Транзакция — это группа операций (вызовов процедур). Транзакция неделима, если ее нельзя прервать и она обратима. Если транзакция *совершается* (commit), выполнение всех операций выглядит единым и неделимым. Если транзакция *прекращается* (abort), то никаких видимых результатов ее выполнения нет. Неделимые транзакции возникли в области баз данных и использовались для программирования отказоустойчивых распределенных приложений.

В статье [Stamos and Gifford, 1990] представлено интересное обобщение RPC, которое названо *удаленными вычислениями* (remote evaluation — REV). С помощью RPC серверный модуль обеспечивает фиксированный набор предопределенных сервисных функций. С помощью REV клиент может в аргументы удаленного вызова включить программу. Получая вызов, сервер выполняет программу и возвращает результаты. Это позволяет серверу обеспечивать неограниченный набор сервисных функций. В работе Стамоса и Гиффорда показано, как использование REV может упростить разработку многих распределенных систем, и описан опыт разработчиков с реализацией прототипа. Аналогичные возможности (хотя чаще всего для клиентской стороны) предоставляют аплеты Java. Например, аплет обычно возвращается сервером и выполняется на машине клиента.

Рандеву были предложены в 1978 г. параллельно и независимо Жаном-Раймоном Абриалем (Jean-Raymond Abrial) из команды Ada и автором этой книги при разработке SR. Термин *рандеву* был введен разработчиками Ada (многие из них были французами). На рандеву основан еще один язык — Concurrent C [Gehani and Roome, 1986, 1989]. Он дополняет язык C процессами, рандеву (с помощью оператора `accept`) и защищенным взаимодействием (с помощью `select`). Оператор `select` похож на одноименный оператор в языке Ada, а оператор `accept` обладает большей мощью. Concurrent C позаимствовал из SR две идеи: условия синхронизации могут ссылаться на параметры, а оператор `accept` — содержать выражение планирования (часть `by`). Concurrent C также допускает вызов операций с помощью `send` и `call`, как и в SR. Позже Джехани и Руми [Gehani and Roome, 1988] разработали Concurrent C++, сочетавший черты Concurrent C и C++.

Совместно используемые примитивы включены в несколько языков программирования, самый известный из которых — SR. Язык StarMod (расширение Modula) поддерживает синхронную передачу сообщений, RPC, рандеву и динамическое создание процессов. Язык Lynx поддерживает RPC и рандеву. Новизна Lynx заключается в том, что он поддерживает динамическую реконфигурацию программы и защиту с помощью так называемых *связей*.

Обзор [Bal, Steiner and Tanenbaum, 1989] представляет исчерпывающую информацию и ссылки по всем упомянутым здесь языкам. Антология [Gehani and McGettrick, 1988] содержит перепечатки основных работ по нескольким языкам (Ada, Argus, Concurrent C, DP, SR), сравнительные обзоры и оценки языка Ada.

Кэширование файлов на клиентских рабочих станциях реализовано в большинстве распределенных операционных систем. Файловая система, представленная схематически в листинге 8.2, по сути та же, что в операционной системе Amoeba. В статье [Tanenbaum et al., 1990] дан обзор системы Amoeba и описан опыт работы с ней. Система Amoeba использует RPC в качестве базовой системы взаимодействия. Внутри модуля потоки выполняются параллельно и синхронизируются с помощью блокировок и семафоров.

В разделе 8.4 были описаны способы реализации дублируемых файлов. Техника взвешенного голосования подробно рассмотрена в работе [Gifford, 1979]. Основная причина использования дублирования — необходимость отказоустойчивости файловой системы. Отказоустойчивость и дополнительные способы реализации дублируемых файлов рассматриваются в исторической справке к главе 9.

Удаленный вызов методов (RMI) появился в языке Java, начиная с версии 1.1. Пояснения к RMI и примеры его использования можно найти в книгах [Flanagan, 1997] и [Hartley, 1998]. (Подробнее об этих книгах и их Web-узлах сказано в конце исторической справки к главе 7.) Дополнительную информацию по RMI можно найти на главном Web-узле языка Java `www.javasoft.com`.

В 1974 году Министерство обороны США (МО США) начало программу "универсального языка программирования высокого уровня" (как ответ на рост стоимости разработки и поддержки программного обеспечения). На ранних этапах программы появилась серия документов с основными требованиями, которые вылились в так называемые спецификации Стилмена (Steelman). Четыре команды разработчиков, связанных с промышленностью и университетами, представили проекты языков весной 1978 г. Для завершающей стадии были выбраны два из них, названные Красным и Зеленым. На доработку им было дано несколько месяцев. "Красной" командой разработчиков руководил Intermetrics, "зеленой" — Cii Honey Bull. Обе команды сотрудничали с многочисленными внешними экспертами. Весной 1979 г. был выбран Зеленый проект. (Интересно, что вначале Зеленый проект основывался на синхронной передаче сообщений, аналогичной используемой в CSP; разработчики заменили ее рандеву летом и осенью 1978 г.)

МО США назвало новый язык Ada в честь Августы Ады Лавлейс, дочери поэта Байрона и помощницы Чарльза Бэббеджа, изобретателя аналитической машины. Первая версия языка Ada с учетом замечаний и опыта использования была усовершенствована и стандартизована в 1983 г. Новый язык был встречен и похвалой, и критикой; похвалой за превосходство над другими языками, которые использовались в МО США, а критикой — за размеры и сложность. Оглядываясь в прошлое, можно заметить, что этот язык уже не кажется таким сложным. Некоторые замечания по Ada 83 и опыт его использования в следующем десятилетии привели к появлению языка Ada 95, который содержит новые механизмы параллельного программирования, описанные в разделе 8.6.

Реализации языка Ada и среды разработки для множества различных аппаратных платформ производятся несколькими компаниями. Этот язык описан во многих книгах. Особое внимание на большие возможности языка Ada обращено в книге [Gehani, 1983]; алгоритм решения задачи об обедающих философах (см. листинги 8.17 и 8.18) в своей основе был взят именно оттуда. В книге [Burns and Wellings, 1995] описаны механизмы параллельного программирования языка Ada 95 и показано, как его использовать для программирования систем реального времени и распределенных систем. Исчерпывающий Web-источник информации по языку Ada находится по адресу `www.adahome.com`.

Основные идеи языка SR (ресурсы, операции, операторы ввода, синхронные и асинхронные вызовы) были задуманы автором этой книги в 1978 г. и описаны в статье [Andrews, 1981]. Полностью язык был определен в начале 1980-х и реализован несколькими студентами. Эндрюс и Олсон (Olsson) разработали новую версию этого языка в середине 1980-х годов. Были добавлены RPC, семафоры, быстрый ответ и некоторые дополнительные механизмы [Andrews et al., 1988]. Последующий опыт и желание обеспечить оптимальную поддержку для параллельного программирования привели к разработке версии 2.0. SR 2.0 представлен в книге [Andrews and Olsson, 1992], где также приведены многочисленные примеры и обзор реализации. Параллельное программирование на SR описано в книге [Hartley, 1995], задуманной как учебное руководство по операционным системам и параллельному программированию. Адрес домашний страницы проекта SR и реализаций: `www.cs.arizona.edu/sr`.

Основная тема этой книги — как писать многопоточные, параллельные и/или распределенные программы. Близкая тема, но имеющая более высокий уровень, — как связывать существующие или будущие прикладные программы, чтобы они совместно работали в распределенном окружении, основанном на Web. Программные системы, которые обеспечивают такую связь, называются *микропрограммными средствами* (middleware). CORBA, Active-X и DCOM — это три наиболее известные системы. Они и большинство других основаны на

объектно-ориентированных технологиях. CORBA (Common Object Request Broker Architecture — технология построения распределенных объектных приложений) — это набор спецификаций и инструментальных средств для обеспечения возможности взаимодействия программ в распределенных системах. Active-X — это технология для объединения таких приложений Web, как броузеры и Java-аплеты с настольными сервисами типа процессоров документов и таблиц. DCOM (Distributed Component Object Model — распределенная модель компонентных объектов) служит основой для удаленных взаимодействий, например, между компонентами Active-X. Эти и многие другие технологии описаны в книге [Umar, 1997]. Полезным Web-узлом по CORBA является `www.omg.org`, а по Active-X и DCOM — `www.activex.org`.

Литература

Andrews, G..R. 1981. Synchronizing resources. *ACM Trans. on Prog. Languages and Systems* 3, 4 (October): 405–430.

Andrews, G. R., and R. A. Olsson. 1992. *Concurrent Programming in SR.* Menlo Park, CA: Benjamin/Cummings.

Andrews, G. R., R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. 1988. An overview of the SR language and implementation. *ACM Trans. on Prog. Languages and Systems* 10, 1 (January): 51–86.

Bal, H. E., J. G. Steiner, and A. S. Tanenbaum. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys* 21, 3 (September): 261–322.

Birrell, A. D., and B. J. Nelson. 1984. Implementing remote procedure calls. *ACM Trans. on Computer Systems* 2, 1 (February): 39–59.

Brinch Hansen, P. 1978. Distributed processes: A concurrent programming concept. *Comm. ACM* 21, 11 (November): 934–941.

Burns, A., and A. Wellings. 1995. *Concurrency in Ada.* Cambridge, England: Cambridge University Press.

Flanagan, D. 1997. *Java Examples in a Nutshell: A Tutorial Companion to Java in a Nutshell.* Sebastopol, CA: O'Reilly & Associates.

Gehani, N. 1983. *Ada: An Advanced Introduction.* Englewood Cliffs, NJ: Prentice-Hall.

Gehani, N. H., and A. D. McGettrick. 1988. *Concurrent Programming.* Reading, MA: Addison-Wesley.

Gehani, N. H., and W. D. Roome. 1986. Concurrent C. *Software — Practice and Experience* 16, 9 (September): 821–844.

Gehani, N. H., and W. D. Roome. 1988. Concurrent C++: Concurrent programming with class(es). *Software — Practice and Experience* 18, 12 (December): 1157–1177.

Gehani, N. H., and W. D. Roome. 1989. *The Concurrent C Programming Language.* Summit, NJ: Silicon Press.

Gifford, D. K. 1979. Weighted voting for replicated data. *Proc. Seventh Symp. on Operating Systems Principles* (December): 150–162.

Hartley, S. J. 1995. *Operating Systems Programming: The SR Programming Language.* New York: Oxford University Press.

Hartley, S. J. 1998. *Concurrent Programming: The Java Programming Language.* New York: Oxford University Press.

Nelson, B. J. 1981. Remote procedure call. Doctoral dissertation, CMU-CS-81-119, Carnegie-Mellon University, May.

Spector, A. Z. 1982. Performing remote operations efficiently on a local computer network. *Comm. ACM* 25, 4 (April): 246–260.

Stamos, J. W, and D. K. Gifford. 1990. Remote evaluation. *ACM Trans. on Prog. Languages and Systems* 12, 4 (October): 537–565.

Tanenbaum, A. S, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. 1990. Experiences with the Amoeba distributed operating system. *Comm. ACM* 33, 12 (December): 46–63.

Umar, A. 1997. *Object-Oriented Client/Server Internet Environments.* Englewood Cliffs, NJ: Prentice-Hall.

Упражнения

8.1. Измените модуль сервера времени в листинге 8.1 так, чтобы процесс часов не запускался при каждом "тике" часов. Вместо этого процесс часов должен устанавливать аппаратный таймер на срабатывание при наступлении следующего интересующего события. Предполагается, что время суток исчисляется в миллисекундах, а таймер можно устанавливать на любое число миллисекунд. Кроме того, процессы могут считывать количество времени, оставшееся до срабатывания аппаратного таймера, и таймер можно переустанавливать в любой момент времени.

8.2. Рассмотрим распределенную файловую систему в листинге 8.2:

а) разработайте законченные программы модулей кэширования файлов и файлового сервера. Разработайте реализацию кэша, добавьте код синхронизации и т.д.;

б) модули распределенной файловой системы запрограммированы с помощью RPC. Перепрограммируйте файловую систему, используя примитивы рандеву, определенные в разделе 8.2. Уточните программу, чтобы по степени детализациии она была сравнима с программой в листинге 8.2.

8.3. Предположим, что модули имеют вид, показанный в разделе 8.1, а процессы разных модулей взаимодействуют с помощью RPC. Кроме того, пусть процессы, которые обслуживают удаленные вызовы, выполняются со взаимным исключением (как в мониторах). Условная синхронизация программируется с помощью оператора `when B`, который приостанавливает выполняемый процесс до тех пор, пока логическое условие `B` не станет истинным. Условие `B` может использовать любые переменные из области видимости выражения:

а) перепишите модуль сервера времени (см. листинг 8.1), чтобы он использовал эти механизмы;

б) перепрограммируйте модуль фильтра слияния (см. листинг 8.3), используя эти механизмы.

8.4. Модуль `Merge` (см. листинг 8.3) имеет три процедуры и локальный процесс. Измените реализацию, чтобы избавиться от процесса `M` (процессы, обслуживающие вызовы операций `in1` и `in2`, должны взять на себя роль процесса `M`).

8.5. Перепишите процесс `TimeServer` (см. листинг 8.7), чтобы операция `delay` задавала интервал времени, как в листинге 8.1, а не действительное время запуска программы. Используйте только примитивы рандеву, определенные в разделе 8.2. (*Указание.* Вам понадобится одна или несколько дополнительных операций, а клиенты не смогут просто вызывать операцию `delay`.)

8.6. Рассмотрим процесс планирующего драйвера диска (см. листинг 7.7). Предположим, что процесс экспортирует только операцию `request(cylinder, ...)`. Покажите, как использовать рандеву и оператор `in` для реализации каждого из следующих алгоритмов планирования работы диска: наименьшего времени поиска, циклического сканирования (CSCAN) и лифта. (*Указание.* Используйте выражения планирования.)

8.7. Язык Ada обеспечивает примитивы рандеву, аналогичные определенным в разделе 8.2 (подробности — в разделе 8.6). Но в эквиваленте оператора `in` в языке Ada условия синхронизации не могут ссылаться на параметры операций. Кроме того, язык Ada не поддерживает выражения планирования.

Используя примитивы рандеву, определенные в разделе 8.2, и указанную ограниченную форму оператора `in` или примитивы языка Ada `select` и `accept`, перепрограммируйте следующие алгоритмы:

а) централизованное решение задачи об обедающих философах (см. листинг 8.6);

б) сервер времени (см. листинг 8.7);

в) диспетчер распределения ресурсов по принципу "кратчайшее задание" (см. листинг 8.8).

8.8. Рассмотрим следующее описание программы поиска минимального числа в наборе целых чисел. Дан массив процессов `Min[1:n]`. Вначале каждый процесс имеет одно целое значение. Процессы многократно взаимодействуют, и при взаимодействии каждый процесс пытается передать другому минимальное из увиденных им значений. Отдавая свое минимальное значение, процесс завершается. В конце концов останется один процесс, и он будет знать минимальное число исходного множества:

а) разработайте программу решения этой задачи, используя только примитивы RPC, описанные в разделе 8.1;

б) напишите программу решения задачи с помощью только примитивов рандеву (см. раздел 8.2);

в) разработайте программу с помощью совместно используемых примитивов, описанных в разделе 8.3. Ваша программа должна быть максимально простой.

8.9. В алгоритме работы читателей и писателей (см. листинг 8.13), предпочтение отдается читателям:

а) измените оператор ввода в процессе `Writer`, чтобы преимущество имели писатели;

б) измените оператор ввода в процессе `Writer` так, чтобы читатели и писатели получали доступ к базе данных по очереди.

8.10. В модуле `FileServer` (см. листинг 8.14) для обновления удаленных копий использован оператор `call`. Предположим, что его заменили асинхронным оператором `send`. Работает ли программа? Если да, объясните, почему. Если нет, объясните, в чем ошибка.

8.11. Предположим, что процессы взаимодействуют только с помощью механизмов RPC, определенных в разделе 8.1, а процессы внутри модуля — с помощью семафоров. Перепрограммируйте каждый из указанных ниже алгоритмов:

а) модуль `BoundedBuffer` (см. листинг 8.5);

б) модуль `Table` (см. листинг 8.6);

в) модуль `SJN_Allocator` (см. листинг 8.8);

г) модуль `ReadersWriters` (см. листинг 8.13);

д) модуль `FileServer` (см. листинг 8.14).

8.12. Разработайте серверный процесс, который реализует повторно используемый барьер для `n` рабочих процессов. Сервер имеет одну операцию — `arrive()`. Рабочий процесс вызывает операцию `arrive`, когда приходит к барьеру. Вызов завершается, когда к барьеру приходят все `n` процессов. Для программирования сервера и рабочих процессов используйте примитивы рандеву из раздела 8.2. Предположим, что

доступна функция ?opname, определенная в разделе 8.3, которая возвращает число задержанных вызовов операции opname.

8.13. В листинге 7.12 был представлен алгоритм проверки простоты чисел с помощью решета из процессов-фильтров, написанный с использованием синхронной передачи сообщений языка CSP. Другой алгоритм, представленный в листинге 7.13, использовал управляющий процесс и портфель задач. Он был написан с помощью пространства кортежей языка Linda:

а) перепишите алгоритм из листинга 7.12 с помощью совместно используемых примитивов, определенных в разделе 8.3;

б) измените алгоритм из листинга 7.13 с помощью совместно используемых примитивов (см. раздел 8.3);

в) сравните производительность ответов к пунктам *а* и *б*. Сколько сообщений необходимо отправить для проверки всех нечетных чисел от 3 до n? Пара операторов send-receive учитывается как одно сообщение, а оператор call — как два, даже если нет возвращаемых значений.

8.14. *Задача о счете.* Несколько людей (процессов) используют общий счет. Каждый из них может помещать средства на счет и снимать их. Текущий баланс равен сумме всех вложенных средств минус сумма всех снятых. Баланс никогда не должен становиться отрицательным.

Используя составную нотацию, разработайте сервер для решения этой задачи, представьте его клиентский интерфейс. Сервер экспортирует две операции: deposit(amount) и withdraw(amount). Предполагается, что значение amount положительно, а выполнение операции withdraw откладывается, если на счету недостаточно денег.

8.15. В комнату входят процессы двух типов A и B. Процесс типа A не может выйти, пока не встретит два процесса B, а процесс B не может выйти, пока не встретит один процесс A. Встретив необходимое число процессов другого типа, процесс сразу выходит из комнаты:

а) разработайте серверный процесс для реализации такой синхронизации. Представьте серверный интерфейс процессов A и B. Используйте составную нотацию, определенную в разделе 8.3;

б) измените ответ к пункту *а* так, чтобы первый из двух процессов B, встретившихся с процессом A, не выходил из комнаты, пока процесс A не встретит второй процесс B.

8.16. Предположим, что в компьютерном центре есть два принтера, A и B, которые похожи, но не одинаковы. Есть три типа процессов, использующих принтеры: только типа A, только типа B, обоих типов.

Используя составную нотацию, разработайте код клиентов каждого типа для получения и освобождения принтера, а также серверный процесс для распределения принтеров. Ваше решение должно быть справедливым при условии, что принтеры в конце концов освобождаются.

8.17. *Американские горки.* Есть n процессов-пассажиров и один процесс-вагончик. Пассажиры ждут очереди проехать в вагончике, который вмещает C человек, C < n. Вагончик может ехать только заполненным:

а) разработайте коды процессов-пассажиров и процесса-вагончика с помощью составной нотации;

б) обобщите ответ к пункту *а*, чтобы использовались m процессов-вагончиков, m > 1. Поскольку есть только одна дорога, обгон вагончиков невозможен, т.е. заканчивать

движение по дороге вагончики должны в том же порядке, в котором начали. Как и ранее, вагончик может ехать только заполненным.

8.18. *Задача об устойчивом паросочетании* (*о стабильных браках*) состоит в следующем. Пусть `Man[1:n]` и `Woman[1:n]` — массивы процессов. Каждый мужчина (man) оценивает женщин (woman) числами от 1 до n, и каждая женщина так же оценивает мужчин. *Паросочетание* — это взаимно однозначное соответствие между мужчинами и женщинами. Паросочетание *устойчиво*, если для любых двух мужчин m_1 и m_2 и двух женщин w_1 и w_2, соответствующих им в этом паросочетании, выполняются оба следующих условия:

- m_1 оценивает w_1 выше, чем w_2, или w_2 оценивает m_2 выше, чем m_1;
- m_2 оценивает w_2 выше, чем w_1, или w_1 оценивает m_1 выше, чем m_2.

Иными словами, паросочетание неустойчиво, если найдутся мужчина и женщина, предпочитающие друг друга своей текущей паре. Решением задачи является множество n-паросочетаний, каждое из которых устойчиво:

а) с помощью совместно используемых примитивов напишите программу для решения задачи устойчивого брака;

б) обобщением этой задачи является задача об устойчивом соседстве. Есть 2n человек. У каждого из них есть список предпочтения возможных соседей. Решением задачи об устойчивом соседстве является набор из n пар, каждая из которых стабильна в том же смысле, что и в задаче об устойчивых браках. Используя составную нотацию, напишите программу решения задачи об устойчивом соседстве.

8.19. Модуль `FileServer` в листинге 8.14 использует по одной блокировке на каждую копию файла. Измените программу так, чтобы в ней использовалось взвешенное голосование, определенное в конце раздела 8.4.

8.20. В листинге 8.14 показано, как реализовать дублируемые файлы, используя составную нотацию (см. раздел 8.3). Для решения этой же задачи напишите программу на языке:

а) Java. Используйте RMI и синхронизированные методы;

б) Ada;

в) SR.

Поэкспериментируйте с программой, помещая различные файловые серверы на разные машины сети. Составьте краткий отчет с описанием программы, проведенных экспериментов и полученных результатов.

8.21. Проведите эксперименты с Java-программой для удаленной базы данных (см. листинг 8.15). Запустите программу и посмотрите, что происходит. Измените программу для работы с несколькими клиентами. Измените программу для работы с более реалистичной базой данных (по крайней мере, чтобы операции занимали больше времени). Составьте краткий отчет с описанием ваших действий и полученных результатов.

8.22. В листинге 8.15 представлена Java-программа, реализующая простую удаленную базу данных. Перепишите программу на языке Ada или SR, проведите эксперименты с ней. Например, добавьте возможность работы с несколькими клиентами, сделайте базу данных более реалистичной (по крайней мере, чтобы операции занимали больше времени). Составьте краткий отчет о том, как вы реализовали программу на SR или Ada, какие эксперименты провели, что узнали.

8.23. В листингах 8.17 и 8.18 представлена программа на языке Ada, которая реализует имитацию задачи об обедающих философах:

а) запустите программу;

б) перепишите программу на Java или SR.

Проведите эксперименты с программой. Например, пусть философы засыпают на случайные промежутки времени во время еды или размышлений. Попробуйте использовать разные количества циклов. Составьте краткий отчет о том, как вы реализовали программу на SR или Java (для пункта *б*), какие эксперименты провели, что изучили.

8.24. В листинге 8.19 показана SR-программа моделирования решения задачи критической секции:

а) запустите программу;

б) перепишите программу на языке Java или Ada.

Поэкспериментируйте с программой. Например, измените интервалы задержки или приоритеты планирования. Составьте краткий отчет о том, как вы реализовали программу на SR или Ada (для пункта *б*), какие эксперименты провели, что изучили.

8.25. В упражнении 7.26 описаны несколько проектов для параллельного и распределенного программирования. Выберите один из них или подобный им, разработайте и реализуйте решение, используя язык Java, Ada, SR или библиотеку подпрограмм, которая поддерживает RPC или рандеву. Закончив работу, составьте отчет с описанием вашей задачи и решения, продемонстрируйте работу программы.

Глава 9

Модели взаимодействия процессов

Как уже отмечалось, существуют три основные схемы взаимодействия процессов: производитель-потребитель, клиент-сервер и взаимодействующие равные. В главе 7 было показано, как их программировать с помощью передачи сообщений, в главе 8 — с помощью RPC и рандеву.

Эти три основные схемы можно сочетать различными способами. В данной главе описаны некоторые из таких укрупненных схем и проиллюстрировано их использование. Каждая схема является *парадигмой* (моделью) взаимодействия процессов; она имеет уникальную структуру, которую можно использовать для решения многих задач. В этой главе описаны следующие парадигмы:

- управляющий-рабочие, представляющая собой распределенную реализацию портфеля задач;
- алгоритмы пульсации, в которых процессы периодически обмениваются информацией, используя передачу, а затем прием сообщений;
- конвейерные алгоритмы, пересылающие информацию от одного процесса к другому с помощью приема, а затем передачи;
- зонды (посылки) и эхо (приемы), которые рассылают и собирают информацию в деревьях и графах;
- алгоритмы рассылки, используемые для децентрализованного принятия решений;
- алгоритмы передачи маркера — еще один способ децентрализованного принятия решений;
- дублируемые серверные процессы, которые управляют несколькими экземплярами такого ресурса, как файл.

Первые три парадигмы обычно используются в синхронных параллельных вычислениях, остальные четыре — в распределенных системах. В данной главе показано, как эти парадигмы применяются для решения различных задач, включая умножение разреженных матриц, обработку изображений, распределенное умножение матриц, построение топологии сети, распределенное взаимное исключение, распределенное определение завершения и децентрализованное решение задачи об обедающих философах. Далее, в главе 11, три парадигмы синхронных параллельных вычислений используются для решения научных вычислительных задач. В упражнениях описаны дополнительные приложения, включая задачи сортировки и коммивояжера.

9.1. Управляющий-рабочие (распределенный портфель задач)

В разделе 3.6 представлена парадигма портфеля задач и показано, как реализовать ее, используя для синхронизации и взаимодействия разделяемые переменные. Напомним основную идею: несколько рабочих процессов совместно используют портфель независимых задач. Рабочий многократно берет из портфеля задачу, выполняет ее и, возможно, порождает одну или несколько новых задач, помещая их в портфель. Преимуществами этого подхода к реализации параллельных вычислений являются легкость варьирования числа рабочих процессов и относительная простота обеспечения того, что процессы выполняют приблизительно одинаковые объемы работы.

Здесь показано, как реализовать парадигму портфеля задач с помощью передачи сообщений вместо разделяемых переменных. Для этого портфель задач реализуется управляющим процессом, который выбирает задачи, собирает результаты и определяет завершение работы. Рабочие процессы получают задачи и возвращают результаты, взаимодействуя с управляющим, который, по сути, является сервером, а рабочие процессы — клиентами.

В первом примере, приведенном ниже, показано, как умножаются разреженные матрицы (большинство их элементов равны нулю). Во втором примере используется сочетание интервалов статического и адаптивного интегрирования в уже знакомой задаче квадратуры. В обоих примерах общее число задач фиксировано, а объем работы, выполняемой каждой задачей, изменяется.

9.1.1. Умножение разреженных матриц

Пусть A и B — матрицы размером n × n. Нужно определить произведение матриц A × B = C. Для этого необходимо вычислить n2 скалярных произведений векторов (сумм, образованных произведениями соответствующих элементов двух векторов длины n).

Матрица называется *плотной*, если большинство ее элементов не равны нулю, и *разреженной*, если большинство элементов нулевые. Если A и B — плотные матрицы, то матрица C тоже будет плотной (если только в скалярных произведениях не произойдет значительного сокращения).

С другой стороны, если A или B — разреженные матрицы, то C тоже будет разреженной, поскольку каждый нуль в A или B даст нулевой вклад в n скалярных произведений. Например, если в строке матрицы A есть только нули, то и вся соответствующая строка C будет состоять из нулей.

Разреженные матрицы возникают во многих задачах, например, при численной аппроксимации решений дифференциальных уравнений в частных производных или в больших системах линейных уравнений. Примером разреженной матрицы является трехдиагональная матрица, у которой равны нулю все элементы, кроме главной диагонали и двух диагоналей непосредственно над и под ней. Если известно, что матрицы разреженные, то запоминание только ненулевых элементов экономит память, а игнорирование нулевых элементов при умножении уменьшает затраты времени.

Разреженная матрица A представляется информацией о ее строках:

```
int lengthA[n];
pair *elementsA[n];
```

Значение `lengthA[i]` является числом ненулевых элементов в строке `i` матрицы A. Переменная `elementsA[i]` указывает на список ненулевых элементов строки `i`. Каждый элемент представлен парой значений (записью): целочисленным индексом столбца и значением соответствующего элемента матрицы (числом удвоенной точности). Таким образом, если значение `lengthA[i]` равно 3, то в списке `elementsA[i]` есть три пары. Они упорядочены по возрастанию индексов столбцов. Рассмотрим конкретный пример.

```
lengthA  elementsA
  1      (3, 2.5)
  0
  0
  2      (1, -1.5)  (4, 0.6)
  0
  1      (0, 3.4)
```

Здесь записана матрица размерами 6 на 6, в которой есть четыре ненулевых элемента: в строке 0 и столбце 3, в строке 3 и столбце 1, в строке 3 и столбце 4, в строке 5 и столбце 0.

Матрицу C представим так же, как и A. Чтобы упростить умножение, представим матрицу B не строками, а столбцами. Тогда значения в `lengthB` будут указывать число ненулевых элементов в каждом столбце матрицы B, а в `elementsB` — храниться пары вида (номер строки, значение элемента).

Для вычисления произведения матриц A и B, как обычно, нужно просмотреть n2 пар строк и столбцов. Для разреженных матриц самый подходящий объем задачи соответствует одной строке результирующей матрицы C, поскольку вся эта строка представляется одним вектором пар (столбец, значение). Таким образом, нужно столько задач, сколько строк в матрице A. (Очевидно, что для оптимизации можно пропускать строки матрицы A, полностью состоящие из нулей, т.е. строки, для которых `lengthA[i]` равно 0, поскольку соответствующие строки матрицы C тоже будут нулевыми. Но в реальных задачах это встречается редко.)

В листинге 9.1 показан код, реализующий умножение разреженных матриц с помощью одного управляющего и нескольких рабочих процессов. Предполагается, что матрица A в управляющем процессе уже инициализирована, а у каждого рабочего есть инициализированная копия матрицы B. Процессы взаимодействуют с помощью примитивов рандеву (см. главу 8), что упрощает программу. Для использования асинхронной передачи сообщений управляющий процесс должен быть запрограммирован в стиле активного монитора (см. раздел 7.3), а вызовы `call` в рабочих процессах нужно заменить вызовами `send` и `receive`.

Листинг 9.1, *а*. Умножение разреженных матриц: управляющий процесс

```
module Manager
  type pair = (int index, double value);
  op getTask(result int row, len; result pair [*]elems);
  op putResult(int row, len; pair [*]elems);
body Manager
  int lengthA[n], lengthC[n];
  pair *elementsA[n], *elementsC[n];
  # предполагается, что матрица A инициализирована
  int nextRow = 0, tasksDone = 0;

  process manager {
    while (nextRow < n or tasksDone < n) {
# нужны еще задачи для выполнения или результаты
      in getTask(row, len, elems) ->
          row = nextRow;
          len = lengthA[row];
          копировать пары из *elementsA[row] в elems;
          nextRow++;
      [] putResult(row, len, elems) ->
          lengthC[row] = len;
          копировать пары из elems в *elementsC[row];
          tasksDone++;
      ni
    }
  }
end Manager
```

Листинг 9.1, *б*. Умножение разреженных матриц: рабочие процессы

```
process worker[w = 1 to numWorkers] {
  int lengthB[n];
  pair *elementsB[n];  # предполагается инициализированным
  int row, lengthA, lengthC;
  pair *elementsA, *elementsC;
  int r, c, na, nb;  # используются при вычислениях
  double sum;        #   промежуточные произведения
  while (true) {
    # получить строку из A и вычислить строку в C
    call getTask(row, lengthA, elementsA);
    lengthC = 0;
```

```
    for [i = 0 to n-1]
      INNER_PRODUCT(i);  # см. текст
    send putResult(row, lengthC, elementsC);
  }
}
```

Управляющий процесс (листинг 9.1, *а*) реализует портфель задач и собирает результаты. Он обслуживает операции `getTask` и `putResult`. Целочисленная переменная `nextRow` указывает на следующую задачу, т.е. следующую строку матрицы. Целочисленная переменная `tasksDone` подсчитывает число возвращенных задачами результатов. После выбора задачи увеличивается на 1 значение переменной `nextRow`, а после получения результатов ее выполнения — `tasksDone`. Когда все задачи выполнены, оба значения равны `n`.

Код рабочих процессов приведен в листинге 9.1, *б*. Рабочий процесс циклически получает и выполняет новую задачу, а затем передает результат ее выполнения управляющему. Задача состоит в вычислении строки результирующей матрицы `C`, поэтому рабочий процесс выполняет цикл `for`, чтобы получить `n` промежуточных произведений, по одному для каждого столбца матрицы `B`. Но код вычисления скалярного произведения (`INNER_PRODUCT`) двух разреженных векторов существенно отличается от цикла вычисления скалярного произведения двух плотных векторов. Поэтому в листинге рабочего процесса `INNER_PRODUCT(i)` обозначает следующий фрагмент кода.

```
sum = 0.0; na = 1; nb = 1;
c = elementsA[na]->index;      # номер столбца в строке A
r = elementsB[i][nb]->index;   # номер строки в столбце B
while (na <= lengthA and nb <= lengthB) {
  if (r == c) {
    sum += elementsA[na]->value *
           elementsB[i][nb]->value;
    na++; nb++;
    c = elementsA[na]->index;
    r = elementsB[i][nb]->index;
  } else if (r < c) {
    nb++; r = elementsB[i][nb]->index;
  } else { # r > c
    na++; c = elementsA[na]->index;
  }
}
if (sum != 0.0) {  # дополнить строку матрицы C
  elementsC[lengthC] = pair(i, sum);
  lengthC++;
}
```

Основная идея состоит в том, чтобы просмотреть разреженное представление строки матрицы `A` и столбца `i` матрицы `B`. (В указанном выше коде предполагается наличие хотя бы одного ненулевого значения в строке и столбце.) Промежуточное произведение не равно нулю, только если есть хотя бы одна пара ненулевых значений, имеющих *одинаковые* индексы `c` в столбце `A` и `r` в строке `B`. В цикле `while` выполняется поиск таких пар значений и прибавление их произведений. Если значение переменной `sum` при завершении цикла не равно нулю, к вектору, представляющему строку `C`, добавляется новая пара чисел. (Предполагается, что память под эти элементы уже выделена.) Вычислив все `n` промежуточных произведений, рабочий процесс отсылает строку `C` управляющему процессу и запрашивает у него новую задачу.

9.1.2. Вернемся к адаптивной квадратуре

Задача квадратуры была представлена в разделе 1.5, где рассматривались статический (итерационный) и динамический (рекурсивный) алгоритмы аппроксимации значения инте-

грала функции f(x) на интервале от a до b. В разделе 3.6 было показано, как реализовать адаптивную квадратуру с помощью разделяемого портфеля задач.

Здесь для реализации распределенного портфеля задач используется управляющий процесс. Но вместо "чистого" алгоритма адаптивной квадратуры, использованного в листинге 3.18, применяется комбинация статического и динамического алгоритмов. Интервал от a до b делится на фиксированное число подынтервалов, и для каждого из них используется алгоритм адаптивной квадратуры. Такое решение сочетает простоту итерационного алгоритма и высокую точность адаптивного. Использование фиксированного числа задач упрощает программы управляющего и рабочих процессов, а также уменьшает число взаимодействий между ними.

В листинге 9.2 приведен код управляющего и рабочих процессов. Поскольку управляющий процесс, по существу, является серверным, для его взаимодействия с рабочими процессами здесь также используются рандеву. Таким образом, управляющий процесс имеет ту же структуру, что и в листинге 9.1, *а*, и также экспортирует операции getTask и putResult, но параметры операций отличаются. Теперь задача определяется конечными точками интервала (переменные left и right), а результатом вычислений является площадь под графиком f(x) на этом интервале. Предполагается, что значения переменных a, b и numIntervals заданы, например, как аргументы командной строки. По этим значениям управляющий процесс вычисляет ширину интервалов. Затем он выполняет цикл, принимая вызовы операций getTask и putTask, пока не получит по одному результату вычислений для каждого интервала (каждой задачи). Отметим использование условия синхронизации "st x< b" в ветви для операции getTask оператора ввода — оно предохраняет операцию getTask от выдачи следующей задачи, когда портфель задач уже пуст.

Листинг 9.2. Адаптивная квадратура с использованием парадигмы "управляющий-рабочие"

```
module Manager
  op getTask(result double left, right);
  op putResult(double area);
body Manager
  process manager {
    double a, b;          # интервал интегрирования
    int numIntervals;     # число используемых интервалов
    double width = (b-a)/numIntervals;
    double x = a, totalArea = 0.0;
    int tasksDone = 0;
    while (tasksDone < numIntervals) {
      in getTask(left, right) st x < b ->
          left = x; x += width; right = x;
      [] putResult(area) ->
          totalArea += area;
          tasksDone++;
      ni
    }
    печать результата totalArea;
  }
end Manager

double f() { ... }          # интегрируемая функция
double quad(...) { ... }    # функция адаптивной квадратуры

process worker[w = 1 to numWorkers] {
  double left, right, area = 0.0;
  double fleft, fright, lrarea;
  while (true) {
```

```
    call getTask(left, right);
    fleft = f(left); fright = f(right);
    lrarea = (fleft + fright) * (right - left) / 2;
    # рекурсивное вычисление площади, как в разделе 1.5
    area = quad(left, right, fleft, fright, lrarea);
    send putResult(area);
  }
}
```

Рабочие процессы в листинге 9.2 разделяют или имеют собственные копии кода функций `f` и `quad` (рекурсивная функция `quad` приведена в разделе 1.5). Рабочий процесс циклически получает задачу от управляющего, вычисляет необходимые для функции `quad` аргументы, вызывает ее для аппроксимации значения площади под графиком функции от `f(left)` до `f(right)`, а затем отсылает результат управляющему процессу.

Когда программа в листинге 9.2 завершается, рабочие процессы блокируются в своих вызовах функции `getTask`. Обычно это безопасно, как и здесь, но в некоторых случаях этот способ завершения выполнения рабочих процессов может привести к зависанию. Задача реализации нормального завершения рабочих процессов оставляется читателю. (*Указание.* Измените функцию `getTask`, чтобы она возвращала `true` или `false`.)

В рассматриваемой программе объем работы, приходящейся на одну задачу, зависит от скорости изменения функции `f`. Таким образом, если число задач приблизительно равно числу рабочих процессов, то вычислительная нагрузка почти наверняка будет несбалансированной. С другой стороны, если задач слишком много, то между управляющим и рабочими процессами будут происходить ненужные взаимодействия, что приведет к излишним накладным расходам. Было бы идеально иметь такое число задач, при котором общий объем работы приблизительно одинаков для всех рабочих процессов. Разумным является число задач, которое в два-три раза больше, чем число рабочих процессов (в данной программе значение `numIntervals` должно быть в два-три раза больше значения `numWorkers`).

9.2. Алгоритмы пульсации

Модель портфеля задач полезна для решения задач, которые возникают при использовании стратегии "разделяй и властвуй" или требуют фиксированного числа независимых задач. *Парадигму пульсации* можно применять во многих итерационных приложениях, параллельных по данным. Например, ее можно использовать, когда данные разделяются между рабочими процессами, каждый из которых отвечает за изменение определенной части данных, причем новые значения зависят от данных из этой же части или непосредственно прилегающих частей. Среди таких приложений — сеточные вычисления, возникающие при обработке изображений или решении дифференциальных уравнений в частных производных, и клеточные автоматы, используемые при моделировании таких процессов, как лесной пожар или биологический рост.

Предположим, что есть массив данных. Каждый рабочий процесс отвечает за определенную часть данных и строится по следующей схеме.

```
process worker[w = 1 to numWorkers] {
  декларации локальных переменных;
  инициализация локальных переменных;
  while (не выполнено) {
    send значения соседям;
    receive значения от соседей;
    обновить локальные значения;
  }
}
```

Этот тип межпроцессного взаимодействия называется *алгоритмом пульсации*, поскольку действия рабочих процессов напоминают работу сердца: расширение при отправке информации, сокращение при сборе новой информации, затем обработка информации и повторение цикла.

Если данные образуют двухмерную сетку, их можно разделить на полосы или блоки. При делении на полосы получим вектор рабочих процессов, у каждого из которых (кроме двух крайних) будет по два соседа. При делении на блоки получим матрицу рабочих процессов, и у каждого из них будет от двух до восьми соседей, в зависимости от положения блока в массиве данных (внутри, на границе или в углу) и количества соседних значений, необходимых для обновления значений в блоке. Трехмерные массивы данных можно делить аналогичным образом на плоскости, прямоугольные призмы или кубы.

Взаимодействие по схеме `send-receive` в алгоритме пульсации приводит к появлению "нечеткого" барьера между рабочими процессами. Напомним, что барьер — это точка синхронизации, которой должны достичь все рабочие процессы перед тем, как продолжить работу. В итерационных вычислениях барьер не позволяет начать новую итерацию, пока все рабочие процессы не закончат предыдущую. Чтобы новая фаза обновления значений не начиналась до того, как все процессы завершат предыдущую фазу, используется обмен сообщениями. Рабочие процессы, которые не являются соседями, могут порознь проводить больше одной итерации, но для соседних процессов это запрещено. Настоящий барьер здесь не нужен, поскольку рабочие процессы разделяют данные только со своими соседями.

Далее разрабатываются алгоритмы пульсации для двух задач: выделения областей (пример обработки изображений) и игры "Жизнь" (пример клеточного автомата). Дополнительные примеры приложений есть в упражнениях и в главе 11.

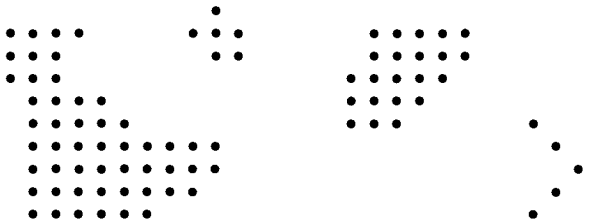
9.2.1. Обработка изображений: выделение областей

Изображение — это представление картинки; обычно оно состоит из матрицы чисел. Элемент изображения называется *пикселем* (от англ. picture element — pixel, элемент картины), и его значение представляет собой интенсивность света или цвет.

Существует множество операций обработки изображений, и каждая из них может выиграть от распараллеливания. Более того, одна и та же операция иногда применяется к потоку изображений. Операции обработки изображений бывают точечными (работают с отдельными пикселями, как, например, при контрастировании), локальными (обрабатывают группы пикселей, как при сглаживании или подавлении шумов) и глобальными (над всеми пикселями, например, при кодировании или декодировании).

Рассмотрим локальную операцию, которая называется *выделением областей*. Пусть изображение представлено матрицей `image[m,n]` целых чисел. Для простоты предположим, что каждый пиксель имеет значение `1` (освещено) или `0` (не освещено). Его соседями считаются пиксели, расположенные сверху, снизу, слева и справа. (У пикселей в углах изображения по два соседа, на границах — по три.)

Задача выделения области состоит в поиске областей освещенных пикселей и присвоении каждой найденной области уникальной метки. Два освещенных пикселя принадлежат одной области, если являются соседями. Рассмотрим, например, следующее изображение, в котором освещенные пиксели сигнала обозначены точками, а неосвещенные — пробелами.



В изображении есть *три* области. "Кривая" в правом нижнем углу не образует область, поскольку ее точки соединены по диагоналям, а не горизонтальным или вертикальным линиям.[16]

Метки областей хранятся во второй матрице `label[m,n]`. Вначале каждая точка изображения получает уникальную метку вроде линейной функции `m*i+j` от координат точки `i` и `j`. Окончательное значение элементов массива `label[i,j]` должно быть равно максимальной из начальных меток в области, содержащей точку `(i,j)`.

Естественный способ решения этой задачи — итерационный алгоритм. На каждой итерации просматриваются все точки и их соседи. Если текущий пиксель и его сосед имеют значение `1`, то меткой пикселя становится максимальная из меток его и соседа. Эти действия можно выполнять для всех пикселей параллельно, поскольку метки никогда не уменьшаются.

Алгоритм завершается, если в течение итерации не изменяется ни одна метка. Обычно области достаточно компактны, и алгоритм прекращает работу примерно через $O(m)$ итераций. Однако в худшем случае потребуется $O(m*n)$ итераций, поскольку область может "виться" по всему изображению.

В данной задаче пиксели независимы, поэтому можно использовать `m*n` параллельных задач. Это решение подходит для SIMD-машины с массовым параллелизмом, но для MIMD-машины такие маленькие задачи использовать неэффективно. Предположим, что есть MIMD-машина с `P` процессорами, и `m` кратно `P`. Тогда было бы правильно решать задачу выделения областей, разделив изображение на `P` полос или блоков пикселей и назначив для каждой полосы или блока отдельный рабочий процесс. Используем деление на полосы — оно проще программируется и требует меньшего числа сообщений, чем деление на блоки, поскольку у рабочих процессов меньше соседей. (На машинах, организованных как сетки или кубы, было бы эффективней использовать блоки точек, поскольку сеть связи в таких машинах поддерживает одновременные передачи сообщений.)

Каждый рабочий процесс вычисляет метки пикселей своей полосы. Для этого ему нужна собственная полоса изображения `image` и полоса матрицы меток `label`, а также значения граничных элементов полос, расположенных над и под его полосой. Поскольку области могут накрывать границы блоков, процесс должен взаимодействовать со своими соседями. Для этого на каждой итерации процесс обменивается метками пикселей на границах своей полосы с двумя соседями, а затем вычисляет новые метки.

В листинге 9.3, *а* показана схема рабочего процесса. После инициализации локальных переменных рабочий процесс обменивается значениями на границе своей части матрицы `image` с соседями. Сначала он отправляет граничные значения соседу сверху и соседу снизу, затем получает значения от соседа снизу и от соседа сверху. Для обмена используются два массива каналов `first` и `second`. Как показано на схеме, рабочие процессы `1` и `P` представляют собой частные случаи, поскольку у них есть только по одному соседу.

В начале каждого повторения цикла `while` соседи-рабочие обмениваются граничными значениями своих частей массива `label`, используя описанную выше схему передачи сообщений. Затем они обновляют метки пикселей своей полосы. Код обновления мог бы обращаться к каждому пикселю один раз или выполняться циклически, пока изменяются метки в полосе. Последний способ приводит к меньшему числу сообщений для обмена метками между рабочими процессами, повышая производительность за счет уменьшения доли вычислений, необходимых для взаимодействия.

В этом приложении рабочий процесс не может сам определить, когда нужно завершить работу. Даже если на итерации не было локальных изменений, могли изменяться метки в другой полосе, а соответствующие им пиксели могли принадлежать области, захватывающей несколько полос. Вычисления заканчиваются, только если не изменяются метки во всем изображении. (В действительности это происходит на одну итерацию раньше, но определить это сразу невозможно.)

---

[16] Неявно предполагается, что область состоит из более, чем одного пикселя. — *Прим. ред.*

Листинг 9.3, *а*. Выделение областей: рабочие процессы[17]

```
chan first[1:P](int edge[n]); #для обмена границами
chan second[1:P](int edge[n]);
chan answer[1:P](bool);       # для проверки завершения

process Worker[w = 1 to P] {
  int stripSize = m/W;
  int image[stripSize+2,n];     # локальные значения и
  int label[stripSize+2,n];     # границы от соседей
  int change = true;
  инициализировать image[1:stripSize,*] и label[1:stripSize,*];
  # обмен границами изображения с соседями
  if (w != 1)
    send first[w-1](image[1,*]);   # рабочему соседу сверху
  if (w != P)
    send second[w+1](image[stripSize,*]);   # соседу снизу
  if (w != P)
    receive first[w](image[stripSize+1,*]); # снизу
  if (w != 1)
    receive second[w](image[0,*]);   # от соседа сверху
  while (change) {
    обменяться границами label с соседями, как показано выше;
    обновить значения label[1:stripSize,*] и присвоить переменной
    change значение "истина", если изменилась хотя бы одна метка;
    send result(change);          # сообщить управляющему
    receive answer[w](change);  # и получить ответ
  }
}
```

Для определения момента завершения программы используется управляющий процесс (листинг 9.3, *б*). (Его функции мог бы выполнять один из рабочих процессов, но для упрощения кода используется отдельный процесс.) В конце каждой итерации все рабочие процессы передают управляющему сообщения, указывающие, изменялись ли метки каждым из процессов. Управляющий процесс объединяет сообщения и отсылает рабочим ответ. Для этих взаимодействий используются каналы `result` и `answer[n]`.

Листинг 9.3, *б*. Выделение областей: управляющий процесс

```
chan result(bool);  # для результатов от рабочих процессов

process Coordinator {
  bool chg, change = true;
  while (change) {
    change = false;
    # посмотреть, были ли изменения в полосах
    for [i = 1 to P] {
      receive result(chg);
      change = change or chg;
    }
    # разослать ответ всем рабочим процессам
    for [i = 1 to P]
      send answer[i](change);
  }
}
```

[17] В процессах не используются каналы `first[P]` и `second[1]`, поэтому массивы каналов можно

Для проверки завершения работы с помощью управляющего процесса на одной итерации нужно обменяться `2*P` сообщениями. Если бы ответ управляющего процесса мог рассылаться сразу всем рабочим, то было бы достаточно `P+1` сообщений. Однако в обоих случаях время работы управляющего процесса составляет $O(P)$, поскольку он получает сообщения с результатами по одному. Используя дерево управляющих процессов, общее время их работы можно снизить до $O(\log_2 P)$. Еще лучше, если доступна операция редукции (сведения) для глобального сбора сообщений, например, операция `MPI_AllReduce` из библиотеки MPI. В результате упростится код программы и, возможно, повысится производительность, в зависимости от того, как реализована библиотека MPI на данной машине.

9.2.2. Клеточный автомат: игра "Жизнь"

Многие биологические и физические системы можно промоделировать в виде набора объектов, которые с течением времени циклически взаимодействуют и развиваются. Некоторые системы, особенно простые, можно моделировать с помощью клеточных автоматов. (Более сложная система — гравитационное взаимодействие — рассматривается в главе 11.) Основная идея — разделить пространство физической или биологической задачи на отдельные клетки. Каждая клетка — это конечный автомат. После инициализации все клетки сначала совершают один переход в новое состояние, затем второй переход и т.д. Результат каждого перехода зависит от текущего состояния клетки и ее соседей.

Здесь клеточный автомат использован для моделирования так называемой игры "Жизнь". Дано двухмерное поле клеток. Каждая клетка либо содержит организм (жива), либо пуста (мертва). В этой задаче каждая клетка имеет восемь соседей, которые расположены сверху, снизу, слева, справа и по четырем диагоналям от нее. У клеток в углах по три соседа, а на границах — по пять.

Игра "Жизнь" происходит следующим образом. Сначала поле инициализируется. Затем каждая клетка проверяет состояние свое и своих соседей и изменяет свое состояние в соответствии со следующими правилами.

- Живая клетка, возле которой меньше двух живых клеток, умирает от одиночества.
- Живая клетка, возле которой есть две или три живые клетки, выживает еще на одно поколение.
- Живая клетка, возле которой находится больше трех живых клеток, умирает от перенаселения.
- Мертвая клетка, рядом с которой есть ровно три живых соседа, оживает.

Этот процесс повторяется некоторое число шагов (поколений).

Листинг 9.4 содержит схему программы для имитации игры "Жизнь". Процессы взаимодействуют с помощью парадигмы пульсации. На каждой итерации клетка посылает сообщения каждому из соседей и получает сообщения от них, после чего обновляет свое состояние в соответствии с приведенными правилами. Как обычно при использовании алгоритма пульсации, для процессов не нужна жесткая пошаговая синхронизация, но соседи никогда не опережают друг друга более, чем на одну итерацию.

Для простоты каждая клетка запрограммирована как процесс, хотя поле можно разделить на полосы или блоки клеток. Также не учтены особые случаи угловых и граничных клеток. Каждый процесс `cell[i,j]` получает сообщения из элемента `exchange[i,j]` матрицы каналов связи, а отсылает сообщения в соседние элементы матрицы `exchange`. (Напомним, что каналы буферизуются, а операция `send` — неблокирующая.) Читателю было бы полезно реализовать эту программу с отображением состояния клеток в графической форме.

было бы объявить как `first[1:P-1]` и `second[2:P]`. — *Прим. ред.*

Листинг 9.4. Игра "Жизнь"

```
chan exchange[1:n,1:n](int row, column, state);

process cell[i = 1 to n, j = 1 to n] {
  int state;   # инициализировать как мертвую или живую
  декларации других переменных;
  for [k = 1 to numGenerations] {
    # обмен состояниями с 8-ю соседями
    for [p = i-1 to i+1, q = j-1 to j+1]
      if (p != i or q != j)
        send exchange[p,q](i, j, state);
    for [p = 1 to 8] {
      receive exchange[i,j](row, column, value);
      запомнить состояние соседа;
    }
    обновить локальное состояние в соответствии с правилами;
  }
}
```

9.3. Конвейерные алгоритмы

Напомним, что процесс-фильтр получает данные из входного порта, обрабатывает их и отсылает результаты в выходной порт. *Конвейер* — это линейно упорядоченный набор процессов-фильтров. Данная концепция уже рассматривалась в виде каналов Unix (раздел 1.6), сортирующей сети (раздел 7.2), а также как способ циркуляции значений между процессами (раздел 7.4). Здесь показано, что эта парадигма полезна и в синхронных параллельных вычислениях.

В решении задач параллельных вычислений обычно используется несколько рабочих процессов. Иногда их можно программировать в виде фильтров и соединять в конвейер параллельных вычислений. Есть три базовые структуры таких конвейеров (рис. 9.1): открытая, закрытая и циклическая (круговая). Рабочие процессы обозначены символами от W_1 до W_n. В *открытом* конвейере входной источник и выходной адресат не определены. Такой конвейер можно включить в любую цепь, для которой он подходит. *Закрытый* конвейер — это открытый конвейер, соединенный с *управляющим процессом*, который производит входные данные для первого рабочего процесса и потребляет результаты, вырабатываемые последним рабочим процессом. Пример открытого конвейера — команда Unix "`grep pattern file | wc`", которую можно поместить в самые разные места. Выполняясь в командной строке, эта команда становится частью закрытого конвейера с пользователем в качестве управляющего процесса. Конвейер называется *циклическим* (круговым), если его концы соединены; в этой ситуации данные циркулируют между рабочими процессами.

В разделе 1.8 были представлены две распределенные реализации умножения матриц `a × b = c`, где `a`, `b` и `c` — плотные матрицы размерами `n × n`. В первом решении работа просто делилась между `n` рабочими процессами, по одному на строку матриц `a` и `c`, но каждый процесс должен был хранить всю матрицу `b`. Во втором решении также использовались `n` рабочих процессов, но каждый из них должен был хранить только один столбец матрицы `b`. В этом решении в действительности применялся круговой конвейер, в котором между рабочими процессами циркулировали столбцы матрицы `b`.

Здесь будут рассмотрены еще две распределенные реализации умножения плотных матриц. В первом решении используется закрытый конвейер, во втором — сеть циклических конвейеров. Оба решения имеют интересные особенности по сравнению с рассмотренными ранее алгоритмами, они также демонстрируют шаблоны, применимые к другим задачам.

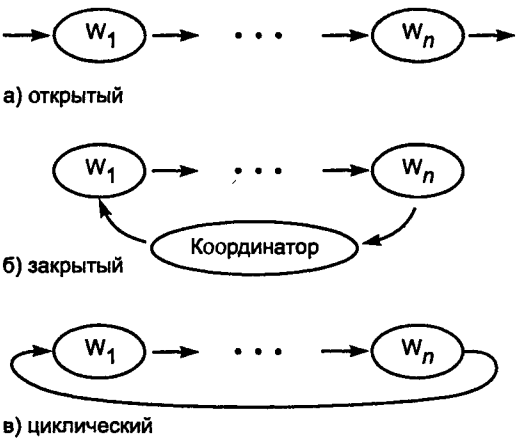


Рис. 9.1. Структуры конвейеров для параллельных вычислений

9.3.1. Конвейер для распределенного умножения матриц

Для простоты снова возьмем матрицы размером n × n и n рабочих процессов, каждый из которых будет вычислять одну строку матрицы c. Но в начале работы у рабочих процессов не будет никаких данных из матриц a и b. Все рабочие процессы объединим в закрытый конвейер, через который они будут получать необходимые данные и возвращать результаты вычислений. Для этого управляющий процесс отсылает каждую строку матрицы a и каждый столбец b по конвейеру первому рабочему и в конце концов получает от последнего рабочего все строки матрицы c.

Действия управляющего процесса показаны в листинге 9.5, *а*. Он отправляет строки от `a[0,*]` до `a[n-1,*]` в канал `vector[0]`, который является входным для рабочего процесса `0`. Затем управляющий процесс отправляет рабочему `0` столбцы от `b[*,0]` до `b[*,n-1]`. Наконец, он получает строки результирующей матрицы от рабочего процесса `n-1`. Однако результаты приходят в порядке от `c[n-1,*]` до `c[0,*]` по причинам, которые объясняются ниже.

Листинг 9.5, *а*. Конвейер для умножения матриц: управляющий процесс

```
chan vector[n](double v[n]);# сообщения для рабочих процессов
chan result(double v[n]);   # строки матрицы c для управляющего процесса

process Coordinator {
  double a[n,n], b[n,n], c[n,n];
  инициализировать матрицы a и b;
  for [i = 0 to n-1]        # отослать все строки матрицы a
    send vector[0](a[i,*]);
  for [i = 0 to n-1]        # отослать все столбцы матрицы b
    send vector[0](b[*,i]);
  for [i = n-1 to 0]          # получить строки матрицы c
    receive result(c[i,*]);   # в обратном порядке
}
```

Каждый рабочий процесс имеет три фазы выполнения. В первой фазе он получает строки матрицы a, сохраняя первую и передавая остальные дальше. На этой фазе строки матрицы a распределяются между рабочими процессами, причем процесс `Working[i]` сохраняет зна-

чения a[i,*]. Во второй фазе рабочие процессы получают столбцы матрицы b, сразу передают их следующему рабочему процессу и вычисляют одно промежуточное произведение. Эту фазу каждый рабочий процесс повторяет n раз, получая в результате значения c[i,*]. В третьей фазе каждый рабочий процесс отсылает свою строку матрицы c следующему рабочему процессу, затем получает и передает далее строки матрицы c от предшествующих процессов конвейера. Последний рабочий процесс передает свою и остальные полученные строки матрицы c управляющему. При этом строки передаются в порядке от c[n-1,*] до c[0,*], поскольку в этом порядке их получает последний рабочий процесс конвейера. При таком порядке передачи снижаются задержки взаимодействия, а последнему рабочему процессу не нужна локальная память для хранения всей матрицы c.

Действия рабочих процессов показаны в листинге 9.5, *б*. Три фазы работы процессов отмечены комментариями. Учтены отличия последнего рабочего процесса от остальных.

Листинг 9.5, *б*. Конвейер для умножения матриц: рабочие процессы

```
process Worker[w = 0 to n-1] {
  double a[n], b[n], c[n];  # "мои" строка или столбец каждой матрицы
  double temp[n];  # для передачи векторов
  double total;    # для вычисления произведения
  # получить строки матрицы a;
  # оставить себе первую и передать остальные дальше
  receive vector[w](a);
  for [i = w+1 to n-1] {
    receive vector[w](temp); send vector[w+1](temp);
  }
  # получить столбцы и вычислить промежуточные произведения
  for [j = 0 to n-1] {
    receive vector[w](b);  # получить столбец матрицы b
    if (w < n-1) # если не последний рабочий, передать дальше
      send vector[w+1](b);
    total = 0.0;
    for [k = 0 to n-1]  # вычислить одно произведение
      total += a[k] * b[k];
    c[j] = total;       # поместить total в матрицу c
  }
  # отослать "мою" строку матрицы c
  #следующему рабочему или управляющему процессу
  if (w < n-1)
    send vector[w+1](c);
  else
    send result(c);
  # получить и передать дальше предыдущие строки матрицы c
  for [i = 0 to w-1] {
    receive vector[w](temp);
    if (w < n-1)
      send vector[w+1](temp);
    else
      send result(temp);
  }
}
```

Это решение имеет несколько интересных свойств. Во-первых, все сообщения следуют по конвейеру одно за другим: сначала строки матрицы a, потом столбцы матрицы b и, наконец, строки матрицы c. Между моментом, когда рабочий процесс получает сообщение, и моментом, когда он передает сообщение дальше, практически нет паузы. Таким образом, сообще-

ния идут непрерывно. Вычисляя промежуточное произведение, рабочий процесс уже передал используемый столбец, поэтому следующий процесс может его получить, передать дальше и начать вычисление своего собственного промежуточного произведения.

Во-вторых, чтобы первый рабочий процесс получил все строки матрицы a и передал их далее, нужно n циклов передачи сообщений. Еще n-1 циклов нужно, чтобы заполнить конвейер, т.е. чтобы каждый рабочий процесс получил свою строку матрицы a. Однако после заполнения конвейера промежуточные произведения вычисляются почти с той же скоростью, с какой могут приходить сообщения. Причина, как уже отмечалось, в том, что столбцы матрицы b следуют сразу за строками матрицы a и передаются рабочими процессами сразу после получения. Если вычисление промежуточного произведения занимает больше времени, чем передача и прием сообщения, то после заполнения конвейера определяющим фактором станет время выполнения вычислений. Оставляем читателю решение интересных задач по выводу уравнений производительности и проведение опытов с пропускной способностью конвейера.

Еще одно интересное свойство рассматриваемого решения — возможность легко изменять число столбцов матрицы b. Для этого достаточно изменить верхние пределы в циклах обработки столбцов. Фактически такой же код можно использовать для умножения матрицы a на любой поток векторов, чтобы получить в результате поток векторов. Например, матрица a может представлять набор коэффициентов линейных уравнений, а поток векторов — различные комбинации значений переменных.

Конвейер также можно "сократить", чтобы использовать меньше рабочих процессов. Для этого каждый рабочий процесс должен хранить полосу строк матрицы a. Можно точно так же передавать по конвейеру столбцы матрицы b и строки c или, уменьшив количество сообщений, сделать их длиннее.

Закрытый конвейер, показанный в листинге 9.5, можно открыть и поместить его рабочие процессы в цепочку другого конвейера. Например, вместо управляющего процесса для создания исходных векторов можно использовать еще один конвейер умножения матриц, а для получения результатов — еще один процесс. Однако, чтобы придать конвейеру наиболее общий вид, через него нужно передавать все векторы (даже строки матрицы a) — тогда на выходе из конвейера эти данные будут доступны какому-нибудь другому процессу.

9.3.2. Блочное умножение матриц

Производительность предыдущего алгоритма определяется длиной конвейера и временем, необходимым для передачи и приема сообщений. Сеть связи некоторых высокопроизводительных машин организована в виде двухмерной сетки или структуры, которая называется *гиперкубом*. Эти виды сетей связи позволяют одновременно передавать сообщения между различными парами соседствующих процессов. Кроме того, они уменьшают расстояние между процессорами по сравнению с их линейным упорядочением, что сокращает время передачи сообщения.

Для эффективного умножения матриц на машинах с такими структурами сети связи нужно делить матрицы на прямоугольные блоки и для обработки каждого блока использовать отдельный рабочий процесс. Таким образом, рабочие процессы и данные распределяются по процессорам в виде двухмерной сетки. У каждого рабочего процесса есть по четыре соседа: сверху, снизу, слева и справа. Соседями считаются рабочие процессы в верхнем и нижнем рядах сетки, а также в ее левом и правом столбцах.

Вернемся к задаче вычисления произведения двух матриц a и b с размерами n × n и сохранения результата в матрице c. Чтобы упростить код, используем отдельный рабочий процесс для каждого элемента матрицы и пронумеруем строки и столбцы от 1 до n. (В конце раздела описано использование блоков значений.) Пусть массив Worker[1:n,1:n] — это матрица рабочих процессов. Матрицы a и b вначале распределены так, что у каждого процесса Worker[i,j] есть соответствующие элементы матриц a и b.

Для вычисления c[i,j] рабочему процессу Worker[i,j] нужно умножить каждый элемент строки i матрицы a на соответствующий элемент столбца j матрицы b и сложить результаты. Однако порядок выполнения операций умножения на результат не влияет! Вопрос в том, как организовать циркуляцию данных между рабочими процессами, чтобы каждый из них получил все необходимые пары чисел.

Для начала рассмотрим процесс Worker[1,1]. Для вычисления значения c[1,1] этому процессу нужны все элементы строки 1 матрицы a и столбца 1 матрицы b. Вначале у процесса есть a[1,1] и b[1,1], поэтому их можно сразу перемножить. Если теперь переместиться на 1 вправо по строке матрицы a и вниз по столбцу матрицы b, то процесс Worker[1,1] получит значения элементов a[1,2] и b[2,1], которые можно умножить и прибавить к значению c[1,1]. Если эти действия повторить еще n-2 раза, перемещаясь вправо по строке матрицы a и вниз по столбцу матрицы b, то процесс Worker[1,1] получит все необходимые ему данные.

К сожалению, такая последовательность сдвигов и умножений годится только для процесса, обрабатывающего диагональные элементы матрицы. Другие рабочие процессы тоже увидят необходимые им значения элементов матриц, но в неправильной последовательности. Однако перед началом умножений и перемещений элементы матриц a и b можно переупорядочить. Для этого нужно сначала циклически сдвинуть строку i матрицы a влево на i столбцов, а столбец j матрицы b вверх на j строк. (Причины, по которым такое перемещение элементов работает, не очевидны; этот порядок перемещения элементов был получен после исследований, проведенных для небольших матриц, и обобщения результатов.) Ниже показан результат предварительной перестановки значений матриц a и b с размерами 4 × 4.

$a_{1,2}$, $b_{2,1}$	$a_{1,3}$, $b_{3,2}$	$a_{1,4}$, $b_{4,3}$	$a_{1,1}$, $b_{1,4}$
$a_{2,3}$, $b_{3,1}$	$a_{2,4}$, $b_{4,2}$	$a_{2,1}$, $b_{1,3}$	$a_{2,2}$, $b_{2,4}$
$a_{3,4}$, $b_{4,1}$	$a_{3,1}$, $b_{1,2}$	$a_{3,2}$, $b_{2,3}$	$a_{3,3}$, $b_{3,4}$
$a_{4,1}$, $b_{1,1}$	$a_{4,2}$, $b_{2,2}$	$a_{4,3}$, $b_{3,3}$	$a_{4,4}$, $b_{4,4}$

После предварительной перестановки значений каждый рабочий процесс имеет два значения, которые он записывает в локальные переменные aij и bij. Затем рабочий процесс инициализирует переменную cij значением aij*bij и выполняет n-1 циклов сдвига и умножения. В каждом цикле значения aij передаются на один столбец влево, а значения bij — на строку выше; процесс получает новые значения, перемножает их и прибавляет произведение к текущему значению переменной cij. Когда рабочие процессы завершаются, произведение матриц хранится в переменных cij всех рабочих процессов.

В листинге 9.6 показан код, реализующий этот алгоритм умножения матриц. Рабочие процессы совместно используют n2 каналов для циркуляции данных влево и еще n2 каналов для циркуляции данных вверх. Из каналов формируются 2n пересекающихся циклических конвейеров. Рабочие процессы одной строки связаны в циклический конвейер, через который данные перемещаются влево; рабочие процессы одного столбца связаны в циклический конвейер, по которому данные идут вверх. Константы LEFT1, UP1, LEFTI и UPJ в каждом рабочем процессе инициализируются соответствующими значениями и используются в операторах send для индексации массивов каналов.

Листинг 9.6. Блочное умножение матриц

```
chan left[1:n,1:n](double);   # для циркуляции a влево
chan up[1:n,1:n](double);     # для циркуляции b вверх

process Worker[i = 1 to n, j = 1 to n] {
  double aij, bij, cij;
  int LEFT1, UP1, LEFTI, UPJ;
  инициализировать указанные выше значения;
  # циклический сдвиг значений aij влево на i столбцов
  send left[i,LEFTI](aij); receive left[i,j](aij);
```

```
  # циклический сдвиг значений bij вверх на j строк
  send up[UPJ,j](bij); receive up[i,j](bij);
  cij = aij * bij;
  for [k = 1 to n-1] {
  # сдвинуть aij влево на 1, bij вверх на 1,
  # перемножить и сложить
    send left[i,LEFT1](aij); receive left[i,j](aij);
    send up[UP1,j](bij); receive up[i,j](bij);
    cij = cij + aij*bij;
  }
}
```

Программа в листинге 9.6 явно неэффективна (если только она не реализована аппаратно). В ней используется слишком много процессов и сообщений, а каждый процесс производит слишком мало вычислений. Но этот алгоритм легко обобщается для использования квадратных или прямоугольных блоков. Каждый рабочий процесс назначается для блоков матриц a и b. Рабочие процессы сначала сдвигают свои блоки матрицы a влево на i блоков столбцов, а блоки матрицы b — вверх на j блоков строк. Затем каждый рабочий процесс инициализирует свой блок результирующей матрицы c промежуточными произведениями своих новых блоков матриц a и b. Затем рабочие процессы выполняют n-1 циклов сдвига матрицы a на блок влево и сдвига матрицы b на блок вверх, вычисляют новые промежуточные произведения и прибавляют их к c. Подробности этого процесса читатель может выяснить самостоятельно (см. упражнения в конце главы).

Дополнительный способ повысить эффективность кода в листинге 9.6 — выполнять при сдвиге данных оба оператора send до выполнения операторов receive. Изменим последовательность операторов

```
send/receive/send/receive
```

на

```
send/send/receive/receive.
```

Это снижает вероятность того, что оператор receive заблокирует работу программы, и делает возможной параллельную передачу сообщений (если она обеспечена сетью связи).

9.4. Алгоритмы типа "зонд-эхо"

Во многих приложениях, таких как Web-поиск, базы данных, игры и экспертные системы, используются деревья и графы. Особое значение они имеют для распределенных вычислений, многие из которых имеют структуру графа с узлами-процессами и ребрами-каналами.

Поиск в глубину (Depth-first search — DFS) — один из классических алгоритмов последовательного программирования для обхода всех узлов дерева или графа. Стратегия DFS в дереве — для каждого узла дерева посетить его узлы-сыновья и после этого вернуться к родительскому узлу. Этот вид поиска называется "поиск в глубину", поскольку каждый путь поиска сначала доходит вниз до узла-листа и лишь затем поворачивает; например, первым будет пройден путь от корня дерева к его крайнему слева листу. В графе общего вида, у которого могут быть циклы, используется тот же подход, нужно только помечать уже пройденные узлы, чтобы проходить по ребрам, выходящим из узла, только по одному разу.

В этом разделе описана парадигма (модель) "зонд-эхо" для распределенных вычислений в графах. *Зонд* — это сообщение, передаваемое узлом своему преемнику; *эхо* — последующий ответ. Поскольку процессы выполняются параллельно, зонды передаются всем преемникам также параллельно. Модель "зонд-эхо", таким образом, является параллельным аналогом модели DFS. Сначала модель зонда будет проиллюстрирована на примере рассылки информации всем узлам сети. Затем при разработке алгоритма построения топологии сети будет добавлено понятие эха.

9.4.1. Рассылка сообщений в сети

Предположим, что есть сеть узлов (процессоров), связанных двунаправленными каналами связи. Узлы могут напрямую связываться со своими соседями. Сеть, таким образом, имеет структуру неориентированного графа.

Предположим, что один узел-источник S должен *разослать* сообщение всем узлам сети. (Точнее, процессу, выполняемому на узле S, нужно разослать сообщение процессам, выполняемым на всех остальных узлах.) Например, на узле S может выполняться сетевой управляющий процесс, которой должен передать новую информацию о состоянии всем остальным узлам.

Если все остальные узлы являются соседями S, рассылка сигнала реализуется тривиально: узел S должен просто напрямую отправить сообщение каждому узлу. Однако в больших сетях узел обычно имеет лишь небольшое количество соседей. Узел S может послать сообщение соседям, а они в свою очередь должны будут передать его своим соседям и так далее. Итак, нам нужен способ рассылки зонда всем узлам.

Предположим, что узел S имеет локальную копию топологии сети. (Позже будет показано, как ее вычислить.) Топология представлена симметричной матрицей логических значений, ее элемент `topology[i,j]` имеет значение "истина", если узлы `i` и `j` соединены, и "ложь" — в противном случае.

Для эффективной рассылки сообщения узел S должен сначала создать *остовное дерево* сети с собой в качестве корня этого дерева. Остовное дерево графа — это дерево, в которое входят все узлы графа, а ребра образуют подмножество ребер графа. На рис. 9.2 показан пример такого дерева; узел S находится слева. Сплошными линиями обозначены ребра остовного дерева, а пунктирными — остальные ребра графа.

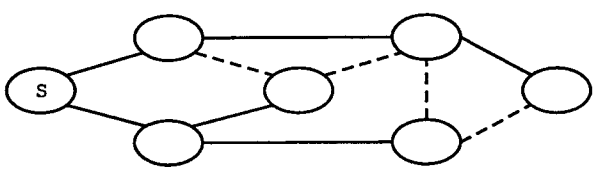


Рис. 9.2. Остовное дерево сети узлов

По данному остовному дереву t узел S может разослать сообщение m, передав его вместе с t всем своим сыновним узлам. Получив сообщение, каждый узел просматривает дерево t, чтобы определить свои сыновние узлы, после чего передает им всем сообщение m и дерево t. Остовное дерево передается вместе с сообщением m, поскольку иначе все узлы, кроме S, не будут знать, какое дерево использовать. Полный алгоритм приведен в листинге 9.7. Поскольку t — остовное дерево, в конце концов сообщение попадет во все узлы. Кроме того, каждый узел получит сообщение только один раз от своего родительского узла в дереве t. Для запуска рассылки сообщения используется отдельный процесс `Initiator` в узле S, благодаря чему процессы `Node` на всех узлах идентичны.

Листинг 9.7. Рассылка сообщения в сети с использованием остовного дерева

```
type graph = bool [n,n];
chan probe[n](graph spanningTree; message m);

process Node[p = 0 to n-1] {
  graph t; message m;
  receive probe[p](t, m);
  for [q = 0 to n-1 st q является сыновним для p в графе t]
```

```
    send probe[q](t, m);
}

process Initiator {  # выполняется на узле-источнике S
  graph topology = топология сети;
  graph t = остовное дерево для topology;
  message m = рассылаемое сообщение;
  send probe[S](t, m);
}
```

В алгоритме рассылки (листинг 9.7) предполагается, что инициирующий процесс заранее знает всю топологию сети и использует ее для построения остовного дерева, управляющего рассылкой. Предположим, что вместо этого каждый узел знает только своих соседей. Тогда можно разослать сообщение m следующим образом. Сначала узел S передает m всем своим соседям. Получив сообщение от одного соседа, узел передает его *всем остальным* своим соседям. Если связи, определенные множествами соседей, образуют дерево с корневым узлом S, то результат будет тем же, что и раньше. Однако в общем случае в сети будут циклы, поэтому некоторые узлы получат сообщение от нескольких соседей. Фактически два соседних узла могут одновременно отослать сообщение друг другу.

Все, что нужно сделать в общем случае, — это проигнорировать копии сообщения m, которые может получить узел. Однако это может привести к "замусориванию" сообщениями или взаимоблокировке. Впервые получив сообщение m и передав его, узел не знает, сколько еще раз можно ждать получения m от соседних узлов. Если узел их не ждет, то такие сообщения могут остаться в буферах каналов probe. Если узел выполняет ожидание заданное число раз, он может остаться заблокированным, не получив необходимого количества сообщений, но их может оказаться и больше.

Проблему необрабатываемых сообщений можно решить, используя полностью симметричный алгоритм. Например, после первого получения сообщения m узел отсылает его всем своим соседям, включая и тот узел, от которого сообщение было получено. После этого узел получает лишние копии сообщения от всех остальных соседей, которые игнорирует. Этот алгоритм представлен в листинге 9.8.

Листинг 9.8. Рассылка сообщений с использованием множеств соседей

```
chan probe[n](message m);

process Node[p = 1 to n] {
  bool links[n] = соседи узла p;
  int num = число соседей;
  message m;
  receive probe[p](m);
  # передать m всем соседям
  for [q = 0 to n-1 st links[q]]
    send probe[q](m);
  # получить num-1 лишних копий m
  for [q = 1 to num-1]
    receive probe[p](m);
}

process Initiator {  # выполняется на узле-источнике S
  message m = рассылаемое сообщение;
  send probe[S](m);
}
```

По алгоритму рассылки с помощью остовного дерева передается n-1 сообщений — по одному на каждое ребро между родительским и сыновним узлами остовного дерева. По алгоритму, использующему множества соседей, через каждую связь сети нужно передать два сообщения, по одному в каждом направлении. Точное число сообщений зависит от топологии сети, но в общем случае оно будет намного больше, чем n-1. Например, если топология сети представляет собой дерево, в корне которого находится узел-источник, будут переданы 2(n-1) сообщений. Для полного графа, в котором есть связи между всеми узлами, потребуется n(n-1) сообщений. Однако в алгоритме с множествами соседей узлу-источнику не нужно знать топологию сети или строить остовное дерево. По существу, остовное дерево строится динамически, оно состоит из связей, по которым проходят первые копии сообщения m. Кроме того, в этом алгоритме сообщения короче, поскольку в каждом из них не нужно передавать остовное дерево.

Оба алгоритма рассылки сообщений предполагают, что топология сети не меняется. Однако они не будут работать правильно, если во время их выполнения даст сбой один из процессоров или одна из связей. Если поврежден узел, он не сможет получить рассылаемое сообщение, а если — линия связи, могут стать недоступными связанные с ней узлы. Работы, в которых рассматриваются проблемы реализации отказоустойчивой рассылки сообщений, описаны в исторической справке в конце главы.

9.4.2. Построение топологии сети

Для применения эффективного алгоритма рассылки сообщений (см. листинг 9.7) необходимо заранее знать топологию сети. Здесь показано, как ее построить. Вначале каждому узлу известна лишь его локальная топология, т.е. связи с соседями. Задача в том, чтобы собрать воедино все локальные топологии, поскольку их объединение является общей топологией сети.

Топология собирается в две фазы. Сначала каждый узел посылает зонд своим соседям, как это происходило в листинге 9.8. Затем каждый узел отсылает эхо, содержащее информацию о локальной топологии, тому узлу, от которого он получил первый зонд. В конце концов инициирующий узел собирает все ответы-эхо и, следовательно, всю топологию. Затем он может, например, построить остовное дерево и разослать топологию всем остальным узлам.

Вначале предположим, что топология сети ациклична. Сеть является неориентированным графом, поэтому ее структура — дерево. Пусть узел S — это корень дерева и инициирующий узел. Тогда топологию можно собрать следующим образом. Сначала узел S передает зонды всем своим сыновним узлам. Когда эти узлы получают зонд, они передают его своим сыновним узлам и т.д. Таким образом, зонды распространяются по всему дереву и в конце концов достигают его листьев. Поскольку у листьев нет сыновних узлов, начинается фаза эха. Каждый лист отсылает эхо, содержащее множество соседних узлов, своему родительскому узлу. После получения эха от всех сыновей узел объединяет эти ответы со своим собственным множеством соседей и передает полученные данные своему родительскому узлу. В конце концов корневой узел получит эхо от каждого из своих сыновей. Объединение этих данных будет содержать всю топологию сети, поскольку начальный сигнал достигнет каждого узла, а каждый эхо-ответ содержит множество, состоящее из отвечающего узла, всех его соседей и их потомков.

Полный алгоритм "зонд-эхо" для сбора топологии сети в дереве приведен в листинге 9.9. Фаза зонда, по существу, является алгоритмом рассылки сообщения из листинга 9.8, за исключением того, что сообщения-зонды идентифицируют отправителя. Фаза эха возвращает информацию о локальной топологии вверх по дереву. Алгоритмы узлов не вполне симметричны, поскольку экземпляр процесса Node[p], выполняемый в узле S, должен знать, что нужно отослать эхо процессу-инициатору.

Листинг 9.9. Алгоритм "зонд-эхо" для сбора топологии дерева

```
type graph = bool [n,n];
chan probe[n](int sender);
chan echo[n](graph topology)     # фрагменты топологии
```

```
chan finalecho(graph topology)  # вся топология

process Node[p = 0 to n-1] {
  bool links[n] = соседи узла p;
  graph newtop, localtop = ([n*n] false);
  int parent;   # узел, от которого получен зонд
  localtop[p,0:n-1] = links;   # вначале локальные связи
  receive probe[p](parent);
  # отослать зонд сыновьям узла p
  for [q = 0 to n-1 st (links[q] and q != parent)]
    send probe[q](p);
  # получить эхо-ответы и объединить их в localtop
  for [q = 0 to n-1 st (links[q] and q != parent)] {
    receive echo[p](newtop);
    localtop = localtop or newtop;  # логическое или
  }
  if (p == S)
    send finalecho(localtop);    # узел S - корень
  else
    send echo[parent](localtop);
}

process Initiator {
  graph topology;
  send probe[S](S)   # запустить зонд в своем узле
  receive finalecho(topology);
}
```

Для построения топологии сети с циклами рассмотренный алгоритм обобщается следующим образом. Получив зонд, узел передает его остальным своим соседям и ждет от них эха. Однако, поскольку в сети есть циклы, а узлы работают параллельно, два соседа могут отослать зонды друг другу почти одновременно. На все зонды, кроме первого, эхо может быть отправлено немедленно. Если узел получает последующие зонды во время ожидания эха, он сразу отсылает эхо с пустой топологией (этого достаточно, поскольку локальные связи узла будут содержаться в эхе-ответе на первый зонд). В конце концов узел получит эхо в ответ на каждый зонд и передает эхо узлу, от которого получил первый зонд. Эхо содержит объединение данных о связях узла вместе со всеми остальными полученными данными.

Обобщенный алгоритм "зонд-эхо" построения топологии сети показан в листинге 9.10. Поскольку узел может получать последующие зонды во время ожидания эха, в один канал объединяются два типа сообщений. (Если бы они приходили по разным каналам, узлу нужно было использовать оператор `empty` и проводить опрос, чтобы решить, какой тип сообщения принять. Можно было бы использовать рандеву, выделив зонды и эхо в отдельные операции.)

Корректность алгоритма вытекает из следующих фактов. Поскольку сеть является связной, каждый узел рано или поздно получит зонд. Взаимоблокировка не возникает, поскольку на каждый зонд посылается эхо-ответ (на первый зонд — перед завершением процесса `Node`, на остальные — сразу после их получения). Это позволяет избежать буферизации исходящих сообщений в каналах `probe_echo`. Последнее эхо, переданное узлом, содержит локальный набор соседей. Следовательно, объединение множеств соседей в конце концов достигает процесса `Node[S]`, передающего топологию процессу `Initiator`. Как и в листинге 9.8, связи, по которым проходят первые зонды, образуют динамически создаваемое остовное дерево. Топология сети возвращается вверх по остовному дереву; эхо от каждого узла содержит топологию поддерева, корнем которого является этот узел.

Листинг 9.10. Алгоритм "зонд-эхо" для построения топологии графа

```
type graph = bool [n,n];
type kind = (PROBE, ECHO);
chan probe_echo[n](kind k; int sender; graph topology);
chan finalecho(graph topology);

process Node[p = 0 to n-1] {
  bool links[n] = соседи узла p;
  graph newtop, localtop = ([n*n] false);
  int first, sender; kind k;
  int need_echo = число соседей - 1;
  localtop[p,0:n-1] = links;   # вначале локальные связи
  receive probe_echo[p](k, first, newtop);  # получить зонд
  # отослать зонд всем соседям
  for [q = 0 to n-1 st (links[q] and q != first)]
    send probe_echo[q](PROBE, p, ∅);
  while (need_echo > 0) {
    # получить эхо или лишние зонды от соседей
    receive probe_echo[p](k, sender, newtop);
    if (k == PROBE)
      send probe_echo[sender](ECHO, p, ∅);
    else  # k == ECHO {
      localtop = localtop or newtop;  # логическое или
      need_echo = need_echo-1;
    }
  }
  if (p == S)
    send finalecho(localtop);
  else
    send probe_echo[first](ECHO, p, localtop);
}

process Initiator {
  graph topology;   # топология сети
  send probe_echo[source](PROBE, source, ∅);
  receive finalecho(topology);
}
```

9.5. Алгоритмы рассылки

В предыдущем разделе мы показали, как рассылать информацию по сети, имеющей структуру графа. В большинстве локальных сетей процессоры разделяют такой канал взаимодействия, как Ethernet или эстафетное кольцо (token ring). Каждый процессор напрямую связан со всеми остальными. Такие сети связи часто поддерживают специальный сетевой примитив — операцию рассылки `broadcast`, которая передает сообщение от одного процессора всем остальным. Независимо от того, поддерживается ли рассылка сообщений аппаратно, она обеспечивает полезную технику программирования.

Пусть `T[n]` — массив процессов, а `ch[n]` — массив каналов (по одному на процесс). Процесс `T[i]` рассылает сообщение `m`, выполняя оператор

```
broadcast ch(m);
```

При выполнении `broadcast` в каждый канал `ch[i]`, включая канал процесса `T[i]`, помещается копия сообщения `m`. Получается тот же результат, что и при выполнении кода

```
co [i = 1 to n]
  send ch[i](m);
```

Процессы получают рассылаемые и передаваемые напрямую сообщения, используя примитив `receive`.

Сообщения, рассылаемые одним и тем же процессом, помещаются в очереди каналов в порядке их рассылки. Однако операция `broadcast` не является неделимой. Например, сообщения, разосланные двумя процессами A и B, могут быть получены другими процессами в разных порядках. (Реализация неделимой рассылки сообщений описана в статьях, указанных в исторической справке.)

Примитив `broadcast` можно использовать для рассылки информации, например, для обмена данными о состоянии процессоров в локальных сетях. Также с его помощью можно решать задачи распределенной синхронизации. В этом разделе разработан алгоритм рассылки для поддержки распределенной реализации семафоров. Основой распределенных семафоров, как и многих других децентрализованных протоколов синхронизации, является полное упорядочение событий взаимодействия. Итак, вначале представим реализацию логических часов и их использование для упорядочения событий.

9.5.1. Логические часы и упорядочение событий

Действия процессов в распределенной программе можно разделить на локальные (чтение и запись переменных) и операции взаимодействия (передача и прием сообщений). Локальные операции не оказывают прямого влияния на другие процессы, а операции взаимодействия — оказывают, передавая информацию и синхронизируясь. Операции взаимодействия, таким образом, в распределенной программе являются важными *событиями*. Термин *событие* далее в тексте указывает на выполнение операторов `send`, `broadcast` или `receive`.

Если два процесса A и B выполняют локальные операции, то отследить относительный порядок выполнения их операций нельзя. Но если A передает (или рассылает) сообщение процессу B, то передача в A должна произойти перед соответствующим приемом сообщения в B. Если B затем передает сообщение процессу C, то передача в B должна произойти раньше, чем соответствующий прием в C. Кроме того, поскольку в B прием предшествует передаче, четыре события взаимодействия вполне упорядочены: передача сообщения из A, его прием в B, передача из B и, наконец, прием в C. Таким образом, *происходит-перед* является транзитивным отношением между причинно связанными событиями.

Хотя причинно связанные события вполне упорядочены, все множество событий в распределенной программе упорядочено лишь частично. Причина в том, что одна последовательность событий, не связанная с другой (например, операции взаимодействия между различными множествами процессов), может происходить после нее, перед ней или одновременно с ней.

Если бы существовали единые центральные часы, то события взаимодействия можно было бы полностью упорядочить, назначив каждому уникальную *метку времени*. Передавая сообщение, процесс мог бы считывать с часов время и присоединять значение времени к сообщению. Получая сообщение, процесс также мог бы прочитать значение времени и записать, когда было получено сообщение. При условии, что точность часов позволяет различать время любой передачи и соответствующего приема сообщения, у события, произошедшего перед другим событием, значение метки времени будет меньше. Кроме того, если процессы имеют уникальные идентификаторы, с их помощью можно упорядочить даже несвязанные события в разных процессах, имеющие одинаковые метки времени. (Например, упорядочить события по возрастанию идентификаторов процессов.)

К сожалению, использовать единые центральные часы практически невозможно. В локальной сети, например, у каждого процессора есть свои часы. Если бы они были точно синхронизированы, их можно было бы использовать для создания меток времени, но совершен-

ная синхронизация невозможна. Существуют алгоритмы синхронизации часов (см. историческую справку) для поддержания "достаточно хорошего", но не абсолютного их согласования. Итак, нам нужен способ имитации физических часов.

Логические часы — это простой целочисленный счетчик, который увеличивается при возникновении события. Предположим, что отдельные логические часы с нулевым начальным значением есть у каждого процесса. Допустим также, что в каждом сообщении есть специальное поле — *метка времени*. Значения логических часов увеличиваются в соответствии со следующими правилами.

Правила изменения значения логических часов. Пусть `A` — процесс с логическими часами `lc`. Процесс `A` обновляет значение `lc` так:

1. передавая или рассылая сообщение, `A` присваивает его метке времени текущее значение переменной `lc` и увеличивает `lc` на `1`;
2. получая сообщение с меткой времени `ts`, `A` присваивает переменной `lc` максимальное из значений `lc` и `ts+1` и затем увеличивает `lc` на `1`.

Поскольку `A` увеличивает `lc` после каждого события, у всех сообщений, передаваемых этим процессом, будут разные, возрастающие метки времени. Поскольку событие получения придает `lc` значение, которое больше метки времени в полученном сообщении, у любого сообщения, посылаемого процессом `A` в дальнейшем, будет большая метка времени.

Используя логические часы, с каждым событием можно связать их значение следующим образом. Значение часов для события передачи сообщения — это метка времени в сообщении, т.е. локальное значение переменной `lc` в начале передачи. Для события получения — это значение `lc` после того, как оно установлено равным максимальному из значений `lc` и `ts+1`, но до того, как оно будет увеличено получающим процессом.

Применение указанных выше правил гарантирует, что, если событие *a* происходит перед событием *b*, то значение часов, связанное с *a*, будет меньше, чем значение, связанное с *b*. Это определяет частичный порядок на множестве причинно связанных событий программы. Если каждый процесс имеет уникальный идентификатор, то полностью упорядочить все события можно, используя для событий с одинаковыми метками времени меньший идентификатор процесса, в котором происходит одно из них.

9.5.2. Распределенные семафоры

Обычно семафоры реализуются с помощью разделяемых переменных. Но их можно реализовать и на основе обмена сообщениями, используя серверный процесс (активный монитор), как показано в разделе 7.3. Их можно также реализовать децентрализовано, т.е. без центрального управляющего. Покажем, как это сделать.

Семафор `s` обычно представляется неотрицательным целым числом. Выполнение операции `P(s)` задерживается, пока значение `s` не станет положительным, а затем оно уменьшается. Выполнение операции `V(s)` увеличивает значение семафора. Таким образом, число завершенных операций `P` в любой момент времени не больше, чем число завершенных операций `V` плюс начальное значение `s`. Поэтому для реализации семафора необходимы способы подсчета операций `P` и `V` и задержки операций `P`. Кроме того, процессы, "разделяющие" семафор, должны взаимодействовать так, чтобы поддерживать инвариант семафора `s >= 0`, даже если состояние программы является распределенным.

Эти требования можно соблюсти, если процессы будут рассылать сообщения о своем желании выполнить операции `P` или `V` и по полученным сообщениям определять, когда можно продолжать. Для этого у каждого процесса должна быть локальная очередь сообщений `mq` и логические часы `lc`, значение которых изменяется в соответствии с представленными выше

правилами. Для имитации выполнения операций P и V процесс рассылает сообщение всем пользовательским процессам, в том числе и себе. Сообщение содержит идентификатор процесса, дескриптор типа операции (POP или VOP) и метку времени. Меткой времени каждой копии сообщения является текущее значение часов lc.

Получив сообщение POP или VOP, процесс сохраняет его в своей очереди mq. Эта очередь поддерживается отсортированной в порядке возрастания меток времени сообщений; сообщения с одинаковыми метками сортируются по идентификаторам отославших их процессов. Допустим пока, что каждый процесс получает все сообщения в порядке их рассылки и возрастания их меток времени. Тогда каждый процесс будет точно знать порядок передачи сообщений POP и VOP, сможет подсчитать количество соответствующих операций P и V и поддерживать истинным инвариант семафора.

К сожалению, операция broadcast не является неделимой. Сообщения, разосланные двумя разными процессами, могут быть получены другими процессами в разных порядках. Более того, сообщение с меньшей меткой времени может быть получено после сообщения с большей меткой. Однако разные сообщения, разосланные одним и тем же процессом, будут получены другими процессами в порядке их рассылки этим процессом, и у сообщений будут возрастающие метки времени. Эти свойства следуют из таких фактов: 1) выполнение операции broadcast — это то же, что параллельное выполнение операций send, которое, как мы считаем, обеспечивает упорядоченную и надежную доставку сообщения, 2) процесс увеличивает значение своих логических часов после каждого события взаимодействия.

То, что последовательные сообщения имеют возрастающие метки времени, дает нам способ принятия синхронизирующих решений. Предположим, что очередь сообщений процесса mq содержит сообщение m с меткой времени ts. Тогда, как только процесс получит сообщение с большей меткой времени от любого другого процесса, он гарантированно уже *никогда* не увидит сообщения с меньшей меткой времени. В этот момент сообщение m становится *полностью подтвержденным*. Кроме того, если сообщение m полностью подтверждено, то и все сообщения, находящиеся перед ним в очереди mq, тоже полностью подтверждены, поскольку их метки времени еще меньше. Поэтому часть очереди mq, содержащая полностью подтвержденные сообщения, является *стабильным префиксом*: в нее никогда не будут вставлены новые сообщения.

При каждом получении сообщения POP или VOP процесс должен рассылать подтверждающее сообщение (ACK), чтобы его получили все процессы. Сообщения ACK имеют обычные метки времени, но не добавляются в очереди сообщений процессов. Их используют просто для того, чтобы определить момент полного подтверждения обычного сообщения из очереди mq. (Если не использовать сообщений ACK, процесс не сможет определить, что сообщение полностью подтверждено, пока не получит более поздних сообщений POP или VOP от всех остальных процессов. Это замедлит работу алгоритма и приведет к блокировке, если какой-нибудь пользователь не захочет выполнить операции P или V.)

Чтобы реализация распределенных семафоров была завершенной, каждый процесс использует локальную переменную s для представления значения семафора. Получая сообщение ACK, процесс обновляет стабильный префикс своей очереди сообщений mq. Для каждого сообщения VOP процесс увеличивает значение s и удаляет это сообщение. Затем процесс просматривает сообщения POP в порядке возрастания меток времени. Если s > 0, процесс уменьшает значение s и удаляет сообщение POP. Таким образом, каждый процесс поддерживает истинность следующего предиката, который является инвариантом цикла процесса.

DSEM: s >= 0 $\wedge$ mq *упорядочена по меткам времени в сообщениях*

Сообщения POP обрабатываются в порядке их появления в стабильном префиксе, чтобы все процессы принимали одинаковые решения о порядке завершения операций P. Хотя процессы могут находиться на разных стадиях обработки сообщений POP и VOP, все они обрабатывают полностью подтвержденные сообщения в одном и том же порядке.

Алгоритм распределенных семафоров представлен в листинге 9.11. Пользовательские процессы — это обычные прикладные процессы. У каждого пользователя есть один вспомогательный процесс; вспомогательные процессы взаимодействуют друг с другом для реализации операций P и V. Пользовательский процесс инициирует операцию P или V, связываясь со своим вспомогательным процессом (помощником). Выполняя операцию P, пользователь ждет, пока его помощник не разрешит ему продолжать. Каждый помощник рассылает сообщения POP, VOP и ACK другим помощникам и управляет локальной очередью сообщений по описанному выше алгоритму. Все сообщения для помощников передаются или рассылаются по массиву каналов semop. Для добавления метки времени к сообщениям все процессы поддерживают локальные часы.

Листинг 9.11. Реализация распределенных семафоров с помощью алгоритма рассылки

```
type kind = enum(reqP, reqV, VOP, POP, ACK);
chan semop[n](int sender; kind k; int timestamp);
chan go[n](int timestamp);

process User[i = 0 to n-1] {
  int lc = 0, ts;
  ...
  # попросить своего помощника выполнить V(s)
  send semop[i](i, reqV, lc); lc = lc+1;
  ...
  # попросить своего помощника выполнить P(s),
  # а затем ждать разрешения
  send semop[i](i, reqP, lc); lc = lc+1;
  receive go[i](ts); lc = max(lc, ts+1); lc = lc+1;
}

process Helper[i = 0 to n-1] {
  queue mq = new queue(int, kind, int);   # очередь сообщений
  int lc = 0, s = 0;       # логические часы и семафор
  int sender, ts; kind k; # значения в полученных сообщениях
  while (true) {   # инвариант цикла DSEM
    receive semop[i](sender, k, ts);
    lc = max(lc, ts+1); lc = lc+1;
    if (k == reqP)
      { broadcast semop(i, POP, lc); lc = lc+1; }
    else if (k == reqV)
      { broadcast semop(i, VOP, lc); lc = lc+1; }
    else if (k == POP or k == VOP) {
      вставить(sender, k, ts) в соответствующую позицию mq;
      broadcast semop(i, ACK, lc); lc = lc+1;
    }
    else {  # k == ACK
      записать, что получен еще один ACK;
      for (все полностью подтвержденные сообщения VOP в mq)
        { удалить сообщение из mq; s = s+1; }
      for (все полностью подтвержденные сообщения POP в mq st s > 0){
        удалить сообщение из mq; s = s-1;
        if (sender == i)     # запрос на P моего пользователя
          { send go[i](lc); lc = lc+1; }
      }
    }
  }
}
```

Распределенные семафоры можно использовать для синхронизации процессов в распределенных программах точно так же, как и обычные семафоры в программах с разделяемыми переменными (глава 4). Например, их можно использовать для решения таких задач взаимного исключения, как блокировка файлов или записей базы данных. Можно использовать тот же базовый подход (рассылка сообщений и упорядоченные очереди) для решения других задач; некоторые из них описаны в исторической справке и упражнениях.

Если алгоритмы рассылки используются для принятия синхронизирующих решений, в этом должны участвовать все процессы. Например, процесс должен "услышать" все остальные процессы, чтобы определить, когда сообщение полностью подтверждается. Это значит, что алгоритмы рассылки не очень хорошо масштабируются для взаимодействия большого числа процессов и их нужно модифицировать, чтобы сделать устойчивыми к ошибкам.

9.6. Алгоритмы передачи маркера

В этом разделе описывается передача маркера — еще одна модель взаимодействия процессов. *Маркер* — это особый тип сообщений, который можно использовать для передачи разрешения или сбора информации о глобальном состоянии. Использование маркера для передачи разрешения иллюстрируется простым распределенным решением задачи критической секции. Сбор информации о состоянии демонстрируется в процессе разработки двух алгоритмов, позволяющих определить, завершились ли распределенные вычисления. Еще один пример приводится в следующем разделе (см. также историческую справку и упражнения).

9.6.1. Распределенное взаимное исключение

Задача о критической секции (КС) возникла в программах с разделяемыми переменными. Однако она встречается и в распределенных программах, если какой-нибудь разделяемый ресурс используется в них с исключительным доступом. Это может быть, например, линия связи со спутником. Кроме того, задача КС обычно является частью большей задачи, такой как обеспечение согласованности распределенного файла или системы баз данных.

Первый способ решения задачи КС — использовать активный монитор, разрешающий доступ к КС. Для многих задач вроде реализации блокировки файлов это самый простой и эффективный метод. Второй способ решения этой задачи — использовать распределенные семафоры, реализованные так, как описано в предыдущем разделе. Этот способ приводит к децентрализованному решению, в котором все процессы одинаковы, но для каждой операции с семафором необходим обмен большим количеством сообщений, поскольку операция `broadcast` должна быть подтверждена.

Здесь задача решается третьим способом — с помощью кольца передачи маркера (token ring). Это децентрализированное и справедливое решение, как и решение с использованием распределенных семафоров, но оно требует обмена намного меньшим количеством сообщений. Кроме того, базовый метод можно обобщить для решения других задач синхронизации.

Пусть `User[1:n]` — это набор прикладных процессов, содержащих критические и некритические секции. Как всегда, нужно разработать протоколы входа и выхода, которые выполняются перед КС и после нее. Протоколы должны обеспечивать взаимное исключение, отсутствие блокировок и ненужных задержек, а также возможность входа (справедливость).

Поскольку у пользовательских процессов есть своя работа, нам не нужно, чтобы они занимались передачей маркера. Используем набор дополнительных процессов `Helper[1:n]`, по одному для каждого пользовательского процесса. Вспомогательные процессы образуют кольцо (рис. 9.3). Маркер циркулирует от процесса `Helper[1]` к процессу `Helper[2]` и так далее до процесса `Helper[n]`, который передает его процессу `Helper[1]`. Получив маркер,

Helper[i] проверяет, не собирается ли входить в КС его клиент User[i]. Если нет, Helper[i] передает маркер. Иначе Helper[i] сообщает процессу User[i], что он может войти в КС, и ждет, пока процесс User[i] не выйдет из нее. После этого Helper[i] передает маркер. Таким образом, вспомогательные процессы работают совместно, чтобы обеспечить постоянное выполнение следующего предиката.

DMUTEX: User[i] *находится в своей КС* $\Rightarrow$
существует ровно один $\wedge$ *маркер маркер у* Helper[i]

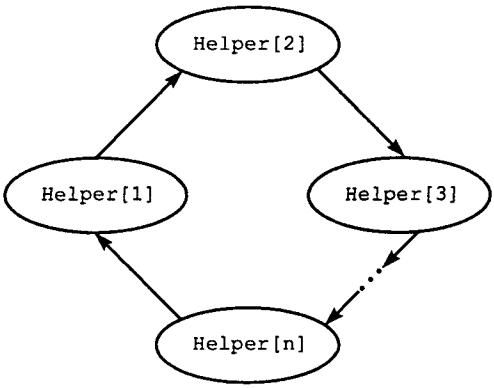


Рис. 9.3. Кольцо с передачей маркера для вспомогательных процессов

Соответствующая программа показана в листинге 9.12. Кольцо передачи маркера представлено массивом каналов token, по одному каналу для каждого вспомогательного процесса. В этой задаче сам маркер не имеет никаких данных, поэтому он представлен пустым сообщением. Другие каналы используются для взаимодействия пользовательских и их вспомогательных процессов. Имея маркер, вспомогательный процесс с помощью оператора empty проверяет, хочет ли его пользовательский процесс войти в КС. Если да, вспомогательный процесс передает пользовательскому сообщение go и ждет получения сообщения exit.

Листинг 9.12. Взаимное исключение с передачей маркера

```
chan token[1:n](), enter[1:n](), go[1:n](), exit[1:n]();

process Helper[i = 1 to n] {
  while (true) {   # инвариант цикла DMUTEX
    receive token[i]();          # ждать маркер
    if (not empty(enter[i])) { # хочет ли пользователь войти в КС?
      receive enter[i]();        # принять сообщение enter
      send go[i]();              # дать разрешение
      receive exit[i]();         # ждать выхода
    }
    send token[i%n + 1]();       # передать маркер
  }
}

process User[i = 1 to n] {
  while (true) {
    send enter[i]();             # протокол входа
```

```
    receive go[i]();
    критическая секция;
    send exit[i]();              # протокол выхода
    некритическая секция;
  }
}
```

Решение в листинге 9.12 является справедливым (при условии, что процессы когда-нибудь выходят из критических секций). Причина в том, что маркер циркулирует непрерывно, и как только он оказывается у процесса `Helper[i]`, процесс `User[i]` получает разрешение войти в КС (если хочет). Фактически то же самое происходит и в физической сети с передачей маркера. Однако при программной передаче маркера, вероятно, лучше добавить некоторую задержку во вспомогательных процессах, чтобы он двигался по кольцу медленней. (В разделе 9.7 представлен еще один алгоритм исключения на основе передачи маркера. Там маркеры не циркулируют непрерывно.)

Данный алгоритм предполагает отсутствие сбоев и потерь маркера. Поскольку управление распределено, алгоритм можно изменить, чтобы он стал устойчивым к сбоям. Алгоритмы восстановления потерянного маркера и использования двух маркеров, циркулирующих в противоположных направлениях, описаны в исторической справке.

9.6.2. Как определить окончание работы в кольце

Окончание работы последовательной программы обнаружить легко. Так же легко определить, что параллельная программа завершилась на одном процессоре — каждый процесс завершен или заблокирован и нет ожидающих операций ввода-вывода. Но не просто обнаружить, что закончилась распределенная программа, поскольку ее глобальное состояние не видно ни одному процессору. Кроме того, даже если все процессоры бездействуют, могут существовать сообщения, передаваемые между ними.

Существует несколько способов обнаружить завершение распределенных вычислений. В этом разделе разрабатывается алгоритм, основанный на передаче маркера при условии, что все взаимодействие между процессами происходит по кольцу. В следующем разделе этот алгоритм обобщается для полного графа связей. Другие подходы описаны в исторической справке и упражнениях.

Пусть в распределенных вычислениях есть процессы (задачи) `T[1:n]` и массив каналов взаимодействия `ch[1:n]`. Пока предположим, что процессы образуют кольцо, по которому проходит их взаимодействие. Процесс `T[i]` получает сообщения только из своего канала `ch[i]` и передает их только в следующий канал `ch[i%n+1]`. Таким образом, `T[1]` передает сообщения только `T[2]`, `T[2]` — только `T[3]` и так далее до `T[n]`, передающего сообщения `T[1]`. Как обычно, предполагается, что сообщения от каждого процесса его сосед по кольцу получает в порядке их передачи.

В любой момент времени процесс может быть активным или простаивать (бездействовать). Вначале активны все процессы. Процесс бездействует, если он завершен или приостановлен оператором получения сообщения. (Если процесс временно приостанавливается, ожидая окончания операции ввода-вывода, он считается активным, поскольку он еще не завершен и через некоторое время снова будет запущен.) Получив сообщение, бездействующий процесс становится активным. Таким образом, распределенные вычисления закончены, если выполняются следующие два условия:

DTERM: *все процессы бездействуют* $\wedge$ *нет передаваемых сообщений*

Сообщение передается (*находится в пути*), если оно уже отправлено, но еще не доставлено в канал назначения. Второе условие обязательно, поскольку доставленное сообщение может запустить приостановленный процесс.

Наша задача — перенести алгоритм определения окончания работы на любые распределенные вычисления, основываясь только на предположении о том, что процессы в вычислении взаимодействуют по кольцу. Окончание — это свойство глобального состояния, объединяющего состояния отдельных процессов и содержание каналов передачи сообщений. Таким образом, чтобы определить момент окончания вычислений, процессы должны взаимодействовать.

Пусть один маркер передается по кольцу в специальных сообщениях, не являющихся частью вычислений. Процесс, удерживающий маркер, передает его дальше и становится бездействующим. (Если процесс закончил свои вычисления, он бездействует, но продолжает участвовать в алгоритме обнаружения завершения работы.)

Процессы передают маркер, используя то же кольцо каналов связи, что и в самих вычислениях. Получая маркер, процесс знает, что отправитель в момент передачи маркера бездействовал. Кроме того, получая маркер, процесс бездействует, поскольку был задержан получением сообщения из канала, и не станет активным до тех пор, пока не получит обычное сообщение, являющееся частью распределенных вычислений. Таким образом, получая маркер, процесс передает его соседу и ждет получения еще одного сообщения из своего канала.

Вопрос в том, как определить, что вычисления завершились. Когда маркер совершает полный круг по кольцу взаимодействия, нам точно известно, что в этот момент все процессы бездействуют. Но как владелец маркера может определить, что все остальные процессы по-прежнему бездействуют и ни одно сообщение не находится в пути к получателю?

Допустим, что вначале маркер находится у процесса `T[1]`. Когда `T[1]` становится бездействующим, он инициирует алгоритм обнаружения окончания работы, передавая маркер процессу `T[2]`. Возвращение маркера к `T[1]` означает, что вычисления закончены, если `T[1]` был *непрерывно бездействующим* с момента передачи им маркера процессу `T[2]`. Дело в том, что маркер проходит по тому же кольцу, по которому идут обычные сообщения, а все сообщения доставляются в порядке их отправки. Таким образом, если маркер возвращается к `T[1]`, значит, ни одного обычного сообщения уже не может быть нигде в очереди или в пути. По сути, маркер "очищает" каналы, проталкивая перед собой все обычные сообщения.

Уточним алгоритм. Во-первых, с каждым процессом свяжем цвет: синий (холодный), если процесс бездействует, и красный (горячий), если он активен. Вначале все процессы активны, поэтому окрашены красным. Получая маркер, процесс бездействует, поэтому становится синим, передает маркер дальше и ждет следующее сообщение. Получив позже обычное сообщение, процесс станет красным. Таким образом, процесс, окрашенный в синий цвет, передал маркер дальше и остался бездействующим.

Во-вторых, с маркером свяжем значение, которое показывает количество пустых каналов, если `T[1]` остается бездействующим. Пусть это значение хранится в переменной `token`. Становясь бездействующим, `T[1]` окрашивается в синий цвет, присваивает `token` значение `0` и передает маркер процессу `T[2]`. Получая маркер, `T[2]` бездействует (при этом канал `ch[2]` может быть пустым). Поэтому `T[2]` становится синим, увеличивает значение `token` до `1` и передает маркер процессу `T[3]`. Все процессы `T[i]` по очереди становятся синими и увеличивают `token` перед тем, как передать дальше.

Описанные правила передачи маркера перечислены в листинге 9.13. Их выполнение гарантирует, что условие *RING* является глобальным инвариантом. Инвариантность *RING* следует из того, что, если процесс `T[1]` окрашен в синий цвет, значит, с момента передачи маркера никаких обычных сообщений он не передавал, и, следовательно, ни в одном из каналов, пройденных маркером, нет обычных сообщений. Кроме того, все соответствующие процессы остались бездействующими с тех пор, как получили маркер. Таким образом, если `T[1]` остался синим, снова получив маркер, то все остальные процессы тоже окрашены в синий цвет, а каналы — пусты. Следовательно, `T[1]` может объявить, что вычисления закончены.

Листинг 9.13. Определение окончания программы в кольце

Глобальный инвариант RING:

```
T[1] — синий ⇒ (T[1], ..., T[token+1] — синие ∧
                ch[2], ..., ch[token%n + 1] пусты)
```

Действия `T[1]`, *когда он впервые становится бездействующим:*

```
color[1] = blue; token = 0; send ch[2](token);
```

Действия `T[2], ..., T[n]` *при получении обычного сообщения:*

```
color[i] = red;
```

Действия `T[2], ..., T[n]` *при получении маркера:*

```
color[i] = blue; token++; send ch[i%n + 1](token);
```

Действия `T[1]` *при получении маркера:*

```
if (color[1] == blue)
  объявить о завершении и остановиться;
color[1] = blue; token = 0; send ch[2](token);
```

9.6.3. Определение окончания работы в графе

В предыдущем разделе предполагалось, что все взаимодействия происходят по кольцу. В общем случае структура взаимодействия распределенной программы образует произвольный ориентированный граф. Узлы графа представляют процессы в вычислениях, а дуги — пути взаимодействия. Два процесса связаны дугой, если первый из них передает данные в канал, из которого их считывает второй.

Предположим, что граф связей является *полным*, т.е. между любыми двумя процессами есть одна дуга. Как и ранее, есть n процессов `T[1:n]` и каналов `ch[1:n]`, и каждый `T[i]` получает данные из собственного канала `ch[i]`. Однако теперь каждый процесс может посылать сообщения только в канал `ch[i]`.

Учитывая вышесказанное, расширим предыдущий алгоритм определения окончания работы. Полученный в результате алгоритм можно использовать в любой сети, имеющей прямые линии взаимодействия между любыми двумя процессорами. Его легко распространить на произвольные графы взаимодействия и многоканальные связи (см. упражнения).

Определить окончание работы в полном графе сложнее, чем в кольце, поскольку сообщения могут идти по любой дуге. Рассмотрим, например, полный граф из трех процессов (рис. 9.4). Пусть процессы передают маркер только от `T[1]` к `T[2]`, затем к `T[3]` и обратно к `T[1]`. Допустим, процесс `T[1]` получает маркер и становится бездействующим; следовательно, он передает маркер процессу `T[2]`. Становясь бездействующим, `T[2]` передает маркер `T[3]`. Но перед получением маркера `T[3]` может передать процессу `T[2]` обычное сообщение. Таким образом, когда маркер возвращается к `T[1]`, нельзя сделать вывод о том, что вычисления завершены, даже если `T[1]` оставался непрерывно бездействующим.

Основным в кольцевом алгоритме (см. листинг 9.13) является то, что *все* взаимодействия проходят по кольцу, поэтому маркер "проталкивает" обычные сообщения, проходя все дуги кольца. Этот алгоритм можно обобщить для полного графа, обеспечив проход маркера по каждой дуге (маркер должен многократно посетить каждый процесс). Если *каждый* процесс остается непрерывно бездействующим с момента предыдущего получения маркера, то можно прийти к выводу, что вычисления завершены.

Как и раньше, процессы окрашиваются в красный или синий цвет (вначале — в красный). Получая обычное сообщение, процесс становится красным, а получая маркер, — блокируется в ожидании следующего сообщения из своего входного канала. Поэтому процесс окрашивается в синий цвет (если он еще не был синим) и передает маркер. (Как и раньше, заканчивая свои обычные вычисления, процесс продолжает обрабатывать сообщения с маркером.)

Любой полный граф содержит цикл, в который входят все его дуги (некоторые узлы могут включаться несколько раз). Пусть C — цикл графа взаимодействия, а nc — его длина. Каждый процесс отслеживает порядок, в котором исходящие из него дуги встречаются в цикле C. Получив маркер по одной дуге цикла C, процесс передает его по следующей. Это гарантирует, что маркер пройдет по каждой дуге графа взаимодействия.

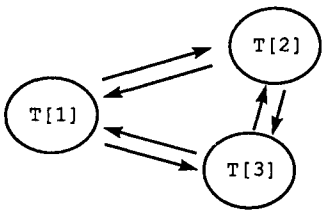


Рис. 9.4. Полный граф взаимодействия

Как и раньше, маркер имеет значение, говорящее о количестве пройденных бездействующих процессов и, следовательно, каналов, которые могут быть пустыми. Но из приведенного выше примера видно, что в полном графе бездействующий процесс снова может стать активным, даже если бездействующим остается процесс `T[1]`. Таким образом, чтобы делать вывод об окончании вычислений, нужен другой набор правил передачи маркера и другой глобальный инвариант.

Маркер начинает движение с любого процесса и имеет начальное значение 0. Когда этот процесс впервые становится бездействующим, он окрашивается в синий цвет и передает маркер по первому ребру цикла C. Получив маркер, процесс совершает действия, представленные в листинге 9.14. Если при получении маркера процесс окрашен в красный цвет (с момента последнего получения маркера он был активным), он становится синим и присваивает маркеру `token` значение 0 перед тем, как передать его по следующему ребру цикла C. Таким образом, алгоритм обнаружения окончания программы перезапускается. Но если при получении маркера процесс окрашен в синий цвет, т.е. с момента последнего получения маркера непрерывно бездействовал, то перед передачей маркера процесс увеличивает его значение.

Листинг 9.14. Определение окончания работы в полном графе

Глобальный инвариант GRAPH:

```
    token имеет значение V ⇒
    ( последние V каналов в цикле C пусты ∧
      последние V процессов, получивших маркер, имели цвет blue )
```

действия процессов `T[i]` *при получении обычного сообщения:*

```
    color[i] = red;
```

действия процессов `T[i]` *при получении маркера:*

```
    if (token == nc)
      объявить о завершении и остановиться;
     if (color[i] == red)
       { color[i] = blue; token = 0; }
     else
       token++;
     присвоить j индекс канала следующего ребра цикла C;
     send ch[j](token);
```

Правила передачи маркера обеспечивают, что *GRAPH* является глобальным инвариантом. Как только значение маркера `token` станет равным `nc` (длине цикла C), станет ясно, что вы-

числения закончены. В этот момент известно, что последние nc каналов, пройденные маркером, были пустыми. Поскольку бездействующий процесс только передает маркер, а значение маркера увеличивает, только если бездействовал с момента прошлого получения маркера, можно наверняка сказать, что все каналы пусты и все процессы бездействуют. В действительности все вычисления завершились уже к тому моменту, когда маркер начал свой последний проход по графу. Но ни один процесс не может это установить до тех пор, пока маркер не сделает еще один полный цикл по графу, чтобы убедиться в том, что все процессы остались бездействующими и все каналы — пустыми. Таким образом, после завершения всех вычислений маркер должен пройти по циклу как минимум дважды: на первом проходе процессы становятся синими, а на втором — проверяется, не поменяли ли они цвет.

9.7. Дублируемые серверы

Опишем последнюю парадигму взаимодействия процессов — дублируемые серверы. Как уже говорилось, сервер — это процесс, управляющий некоторым ресурсом. Сервер можно дублировать, если есть несколько отдельных экземпляров ресурса; каждый сервер тогда управляет одним из экземпляров. Дублирование также можно использовать, чтобы дать клиентам иллюзию уникальности ресурса, когда в действительности экземпляров ресурса несколько. Подобный пример был представлен при реализации дублирования файлов (см. раздел 8.4).

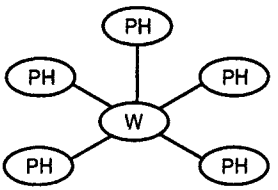
В этом разделе строятся два дополнительных решения задачи об обедающих философах и иллюстрируются два применения дублируемых серверов. Как обычно, в задаче есть пять философов и пять вилок, и для еды философу нужны две вилки. В виде распределенной программы эту задачу можно решить тремя способами. Пусть *PH* — это процесс-философ, *W* — процесс-официант. В первом способе всеми пятью вилками управляет один процесс-официант (эта *централизованная* структура показана на рис. 9.5, *а*). Второй способ — распределить вилки, чтобы одной вилкой управлял один официант (*распределенная* структура на рис. 9.5, *б*). Третий способ — прикрепить официанта к каждому философу (*децентрализованная* структура на рис. 9.5, *в*). Централизованное решение было представлено в листинге 8.6, а распределенное и децентрализованное решения разрабатываются здесь.

9.7.1. Распределенное решение задачи об обедающих философах

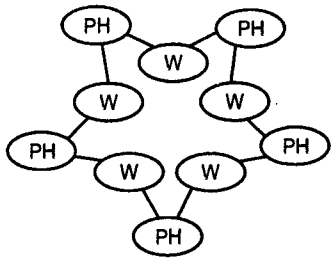
Централизованное решение задачи об обедающих философах (см. листинг 8.6) свободно от блокировок, но не справедливо. Кроме того, узким местом программы может стать процесс-официант, поскольку с ним должны взаимодействовать все процессы-философы. Распределенное решение можно сделать свободным от блокировок, справедливым и не имеющим узкого места, но платой за это будет более сложный клиентский интерфейс и увеличение числа сообщений.

В листинге 9.15 приведено распределенное решение, в котором использована составная нотация из раздела 8.3. (Это приводит к короткой программе, хотя ее легко переписать с помощью только передачи сообщений или только рандеву.) Есть пять процессов-официантов, каждый из которых управляет одной вилкой. Официант постоянно ожидает, пока философ возьмет вилку, а потом отдаст ее. Каждый философ, чтобы получить вилки, взаимодействует с двумя официантами. Чтобы не возникали блокировки, философы не должны выполнять одну и ту же программу. Вместо этого каждый из первых четырех философов берет левую, а затем правую вилки, а последний философ — сначала правую, а потом левую. Это решение очень похоже на решение с семафорами в листинге 4.6.

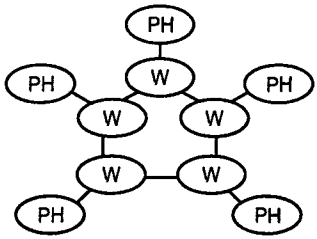
Распределенное решение в листинге 9.15 является справедливым, поскольку вилки запрашиваются по одной, и вызовы операции `getforks` обслуживаются в порядке вызова. Таким образом, все вызовы операции `getforks` в конце концов будут обслужены при условии, что философы когда-нибудь отдадут полученные вилки.



а) централизованная структура



б) распределенная структура



в) децентрализованная структура

Рис. 9.5. Структуры решений задачи об обедающих философах

Листинг 9.15. Распределенное решение задачи об обедающих философах

```
module Waiter[5]
  op getforks(), relforks();
body
  process the_waiter {
    while (true) {
      receive getforks();
      receive relforks();
    }
  }
end Waiter

process Philosopher[i = 0 to 4] {
  int first = i, second = i+1;
  if (i == 4) {
    first = 0; second = 4; }
  while (true) {
    call Waiter[first].getforks();
    call Waiter[second].getforks();
    поесть;
    send Waiter[first].relforks();
    send Waiter[second].relforks();
    подумать;
  }
}
```

9.7.2. Децентрализованное решение задачи об обедающих философах

Построим децентрализованное решение, в котором у каждого философа есть свой официант. Схема взаимодействия процессов здесь похожа на использованную в задаче о дублируемых файлах (см. листинги 8.1 и 8.14). Алгоритм работы процессов-официантов — это еще один пример передачи маркера (маркерами являются пять вилок). Представленное здесь решение можно адаптировать для управления доступом к дублируемым файлам или получить из него эффективное решение задачи распределенного взаимного исключения (см. упражнения).

Каждая вилка — это маркер, который находится у одного из двух официантов или в пути между ними. Когда философ хочет есть, он просит у своего официанта две вилки. Если у официанта в данный момент нет обеих вилок, он просит у соседних официантов. После этого официант контролирует вилки, пока философ ест.

Ключ к правильному решению — управлять вилками, избегая взаимных блокировок. Желательно, чтобы решение было справедливым. В данной задаче блокировка может возникнуть, если официанту нужны две вилки, но он не может их получить. Официант обязательно должен удерживать обе вилки, пока его философ ест, а если философ не ест, официант должен быть готовым отдать вилки по первому требованию. Однако необходимо избегать ситуации, когда вилка передается от одного официанта другому и обратно без использования.

Основная идея избавления от взаимных блокировок — официант должен отдать использованную вилку, но удерживать вилку, которую только что получил. Для этого, когда философ начинает есть, официант помечает обе вилки как "грязные". Если вилка нужна другому официанту, и она уже грязная и не используется, первый официант моет вилку и отдает. Но грязную вилку можно использовать повторно, пока она не понадобится другому официанту.

Этот децентрализованный алгоритм приведен в листинге 9.16. (Из-за мытья вилок его часто называют "алгоритмом гигиеничных философов".) Решение запрограммировано с помощью составной нотации (см. раздел 8.3), поскольку его легче построить, используя и удаленные вызовы процедур, и рандеву, и передачу сообщений.

Желая поесть, философ вызывает операцию `getforks`, экспортируемую модулем. Операция `getforks` реализована процедурой, чтобы скрыть, что получение вилок требует отправки сообщения `hungry` и получения сообщения `eat`. Получая сообщение `hungry`, процесс-официант проверяет состояние двух вилок. Если обе вилки у него, философ получает разрешение поесть, а официант ждет, пока философ отдаст вилки.

Если у процесса-официанта нет двух нужных вилок, он должен их взять, используя для этого операции `needL`, `needR`, `passL` и `passR`. Когда философ голоден и его официанту нужна вилка, официант передает сообщение другому официанту, у которого есть эта вилка. Другой официант получает это сообщение, когда вилка уже грязная и не используется, и передает вилку первому официанту. Операции `needR` и `needL` вызываются асинхронным вызовом `send`, а не синхронным `call`, поскольку одновременный вызов операций `call` двумя официантами может привести ко взаимной блокировке.

Для хранения состояния вилок каждый официант использует четыре переменные: `haveL`, `haveR`, `dirtyL` и `dirtyR`. Эти переменные инициализируются, когда в процессе `Main` вызываются операции `forks` модулей `Waiter`. Вначале официант `0` держит две грязные вилки, официанты `1-3` — по одной (правой) грязной вилке, а у официанта `4` вилок нет.

Во избежание взаимной блокировки вилки вначале распределяются принудительно, причем асимметрично, *и* все они грязные. Если бы, например, у каждого официанта вначале было по одной вилке, и все философы захотели есть, то каждый официант мог бы отдать свою вилку и затем удерживать полученную (взаимная блокировка!). Если бы какая-то вилка вначале была чистой, то официант не отдал бы эту удерживаемую вилку, пока философ не закончит есть; если процесс-философ завершился или философ никогда не захочет есть, то другой философ будет без конца ждать эту вилку.

Листинг 9.16. Децентрализованное решение задачи об обедающих философах

```
module Waiter[t = 0 to 4]
  op getforks(int), relforks(int);  # для философов
  op needL(), needR(),              # для официантов
     passL(), passR();
  op forks(bool,bool,bool,bool);  # для инициализации
body
  op hungry(), eat();       # локальные операции
  bool haveL, dirtyL, haveR, dirtyR;  # состояние вилок
  int left = (t-1) % 5 ;        # сосед слева
  int right = (t+1) % 5;        # сосед справа

  proc getforks() {
    send hungry();  # сообщить официанту, что философ голоден
    receive eat();  # ждать разрешения поесть
  }

  process the_waiter {
    receive forks(haveL, dirtyL, haveR, dirtyR);
    while (true) {
      in hungry() ->
          # попросить вилки, которых у меня нет
          if (!haveR) send Waiter[right].needL();
          if (!haveL) send Waiter[left].needR();
          # ждать получения обеих вилок
          while (!haveL or !haveR)
            in passR() ->
                haveR = true; dirtyR = false;
            [] passL() ->
                haveL = true; dirtyL = false;
            [] needR() st dirtyR ->
                haveR = false; dirtyR = false;
                send Waiter[right].passL();
                send Waiter[right].needL()
            [] needL() st dirtyL ->
                haveL = false; dirtyL = false;
                send Waiter[left].passR();
                send Waiter[left].needR();
            ni
          # разрешить философу поесть, затем ждать освобождения вилок
          send eat(); dirtyL = true; dirtyR = true;
          receive relforks();
      [] needR() ->
          # соседу нужна моя правая вилка (его левая)
          haveR = false; dirtyR = false;
          send Waiter[right].passL();
      [] needL() ->
          # соседу нужна моя левая вилка (его правая)
          haveL = false; dirtyL = false;
          send Waiter[left].passR();
      ni
    }
  }
end Waiter

process Philosopher[i = 0 to 4] {
  while (true) {
```

```
    call Waiter[i].getforks();
    поесть;
    call Waiter[i].relforks();
    подумать;
  }
}

process Main { # инициализировать вилки у официантов
  send Waiter[0].forks(true, true, true, true);
  send Waiter[1].forks(false, false, true, true);
  send Waiter[2].forks(false, false, true, true);
  send Waiter[3].forks(false, false, true, true);
  send Waiter[4].forks(false, false, false, false);
}
```

Философы не будут голодать, поскольку официанты всегда отдают грязные вилки. Если одному официанту нужна вилка, которую держит второй, он в конце концов ее получит. Если вилка грязная и не используется, второй официант немедленно отдаст ее первому. Если вилка грязная и используется, то второй философ когда-нибудь закончит есть, и его официант отдаст вилку первому. Чистая вилка означает, что другой философ захотел есть, его официант только что взял вилку или ждет вторую. По тем же причинам второй официант когда-нибудь обязательно получит вторую вилку, поскольку нет состояния, в котором каждый официант держит чистую вилку и просит еще одну. (Это еще одна причина, по которой нужна асимметричная инициализация официантов.)

Историческая справка

Все парадигмы, описанные в этой главе, были разработаны между серединой 1970-х годов и серединой 1980-х. В течение этих десяти лет интенсивно исследовались многие модели, а затем больше внимания стало уделяться их применению. Многие задачи можно решить несколькими способами, используя разные парадигмы. Некоторые из них описаны ниже; дополнительные примеры приводятся в упражнениях и в главе 11.

Парадигма "управляющий-рабочие" была представлена в статье [Gentleman, 1981] и названа парадигмой "администратор-рабочий". В других работах ([Carriero et al., 1986], [Carriero and Gelernter, 1989]) эта же идея называлась распределенным портфелем задач. В них были представлены решения нескольких задач, запрограммированные с помощью примитивов Linda (см. раздел 7.7). В статье [Finkel and Manber, 1987] использовался распределенный портфель задач для реализации алгоритмов с возвратом (backtracking). Модель "управляющий-рабочие" широко используется в параллельных вычислениях, где эту технику иногда называют рабочим пулом, процессорной или рабочей фермой. Независимо от названия идея всегда одна и та же: несколько рабочих процессов динамически делят набор задач.

Алгоритмы пульсации широко используются в распределенных параллельных вычислениях, а особенно часто — в сеточных вычислениях (см. раздел 11.1). Автор этой книги ввел термин *алгоритм пульсации* в конце 1980-х; это словосочетание показалось ему точно характеризующим действия процессов: закачка (передача), сокращение (прием), подготовка к следующему циклу (вычисления) и повторение этого цикла. Бринч Хансен [Brinch Hansen, 1995] назвал это парадигмой клеточных автоматов, хотя этот термин больше подходит для описания типа приложения, а не стиля программирования. В любом случае ставший каноническим стиль программирования "передать-принять-вычислить" многие никак не называют, а просто говорят, что процессы обмениваются информацией.

В разделе 9.2 были представлены алгоритмы пульсации для задачи выделения областей в изображении и для игры "Жизнь", которую придумал математик Джон Конвей (John Conway)

в 1960-х. Обработка изображений, клеточные автоматы и связанные с этим вопросы в генетических алгоритмах достаточно подробно описаны в книге [Wilkinson and Allen, 1999]. В книге [Brinch Hansen, 1995] также представлен клеточный автомат, а в [Fox et al., 1988] — широкий круг приложений и алгоритмов, многие из которых запрограммированы в стиле пульсации.

Идея программного конвейера возникла с появлением операционной системы Unix в начале 1970-х годов. Понятие аппаратного конвейера восходит еще к векторным процессорам (1960-е годы). Однако в качестве общей модели параллельного программирования конвейеры стали использоваться сравнительно недавно. Конвейер и кольцо (закрытый конвейер) используются для решения многих задач. Примеры можно найти в большинстве книг по параллельным вычислениям. Вот некоторые из них: [Brinch Hansen, 1995], [Fox et al., 1988], [Quinn, 1994], [Wilkinson and Allen, 1999]. Аппаратные конвейеры подробно рассмотрены в монографии [Hwang, 1993]. Поведение аппаратных и программных конвейеров аналогично, поэтому их производительность можно оценивать одними и теми же способами.

Идея парадигмы "зонд-эхо" возникла у нескольких исследователей одновременно. Наиболее полной из первых работ является [Chang, 1982]; в ней представлены алгоритмы решения нескольких задач с графами, включая сортировку, вычисление двухсвязных компонентов и обнаружение узлов (взаимоблокировок). (Они были названы алгоритмами эха; здесь используется термин "зонд-эхо", чтобы выделить существование двух отдельных фаз.) Эта же модель, хотя и без названия, использована в статьях [Dijkstra and Scholten, 1980] и [Frances, 1980], чтобы обнаружить окончание распределенной программы.

Для иллюстрации применения алгоритма "зонд-эхо" в данной книге использовалась задача о топологии сети. Впервые эта задача была поставлена в статье [Lamport, 1982]. В ней показано, как из алгоритма, использующего разделяемые переменные, систематическим образом можно вывести распределенный алгоритм. Там же описана работа с динамической сетью, в которой процессоры и связи могут отказывать и восстанавливаться. В статье [McCurley and Schneider, 1986] для решения этой же задачи выведен алгоритм пульсации.

Несколько ученых исследовали задачу обеспечения надежной или отказоустойчивой рассылки данных, состоящей в том, что каждый функционирующий и достижимый процессор должен получить рассылаемое сообщение. Например, в статье [Schneider et al., 1984] представлен алгоритм отказоустойчивой рассылки в дереве при условии, что отказавший процессор перестает работать и сбои можно обнаружить. В работе [Lamport et al., 1982] показано, как справляться со сбоями, которые могут привести к неопределенному поведению (так называемые византийские сбои).

Логические часы разработал Лампорт. В ставшей классической работе [Lamport, 1978] он показал, как упорядочивать события в распределенных системах. (В статье [Marzullo and Owicki, 1983] описана задача синхронизации физических часов.) В статье [Schneider, 1982] представлена реализация распределенных семафоров, аналогичная приведенной в листинге 9.11. В статье также показано, как изменить этот алгоритм для обработки сбоев. Такой же базовый подход (рассылка сообщений и упорядоченные очереди) можно использовать в решении многих других задач. Например, в [Lamport, 1978] представлен алгоритм для распределенного взаимного исключения, а в [Schneider, 1982] — распределенная реализация защищенных команд ввода-вывода в языке CSP (см. раздел 7.6). В алгоритмах из этих работ не предполагается, что сообщения вполне упорядочены. Однако это предположение упрощает решение многих задач, если операция рассылки неделима (все процессы получают рассылаемые сообщения в одной и той же последовательности). Две основные работы по использованию и реализации надежных и неделимых примитивов взаимодействия — это [Birman and Joseph, 1987] и [Birman et al., 1991].

В разделе 9.6 показано, как использовать передачу маркера для реализации распределенного взаимного исключения и определения окончания программы, а в разделе 8.4 — как использовать маркеры для синхронизации доступа к дублируемым файлам. Передача маркера для справедливого разрешения конфликтов использована в работе [Chandy and Misra, 1984], а для определения глобальных состояний в распределенных вычислениях — в [Chandy and Lamport, 1985].

Решение задачи распределенного взаимного исключения на основе передачи маркера (см. листинг 9.12) было представлено в работе [LeLann, 1977]. Там также было показано, как пропус-

тить вершину кольца, если она дала сбой, и как регенерировать маркер после его потери. Для метода Ле Ланна нужно знать максимальные задержки взаимодействия и идентификаторы процессов. Позже был разработан алгоритм ([Misra, 1983]), у которого этих ограничений нет благодаря использованию двух маркеров, циркулирующих по кольцу в противоположных направлениях.

Задачу распределенного взаимного исключения можно решить, используя алгоритм рассылки. Один из способов — использовать семафоры. Но для этого нужен обмен большим количеством сообщений, поскольку каждое сообщение должно подтверждаться процессом. Более эффективные алгоритмы, основанные на рассылке сообщений, были построены Лампортом, Райкартом и Агравалой (Ricart and Agrawala), Маекавой (Maekawa), Сузуки (Suzuki) и Касами (Kasami). Их описание и сравнение дается в книгах [Raynal, 1986] и [Maekawa et al., 1987].

Алгоритм с передачей маркера для определения окончания работы в кольце основан на алгоритме, описанном в [Dijkstra et al., 1983]. Алгоритм для полного графа связей впервые появился в работе [Misra, 1983]; там также было описано, как преобразовать этот алгоритм для определения окончания работы в произвольном графе. Эту задачу можно решить и на основе других моделей. Например, в [Dijkstra and Scholten, 1980] и [Frances, 1980] для разных вариантов этой задачи представлены алгоритмы "зонд-эхо", а в [Rana, 1983] и [Morgan, 1985] показано, как использовать логические часы и метки времени. В [Chandrasekaran and Vemkatesan, 1990] представлен алгоритм, оптимальный по числу сообщений, в котором сочетаются "зонд-эхо" и передача маркера. Описание и сравнение этих и других алгоритмов есть в [Raynal, 1988].

Выявление взаимной блокировки в распределенной системе аналогично обнаружению окончания работы программы. Для решения этой задачи были разработаны алгоритмы "зонд-эхо" и передачи маркера. Например, в статье [Chandy et al., 1983] представлен алгоритм "зонд-эхо", а в [Knapp, 1987] дан обзор алгоритмов выявления взаимных блокировок в распределенных системах баз данных.

В разделе 8.4 описано, как реализовать дублируемые файлы с помощью маркеров или взвешенного голосования. Можно также использовать распределенные семафоры, как описано в работе [Schneider, 1980]. Представленный там подход можно сделать устойчивым к ошибкам, используя методы из [Schneider, 1982]. Чтобы сделать алгоритмы с маркером или блокировкой устойчивыми к сбоям, можно регенерировать маркер или блокировку (работы [LeLann, 1977] и [Misra, 1983]). Когда есть несколько маркеров и сервер отказывает, другие серверы должны выбрать, кто из них получит потерянные маркеры. В книге [Raynal, 1988] описано несколько алгоритмов выбора.

Здесь несколько раз упоминался вопрос копирования со сбоями, а в некоторых из уже перечисленных работ показано, как сделать определенные алгоритмы устойчивыми к сбоям. Однако полное рассмотрение отказоустойчивого программирования выходит за рамки этой книги. В работе [Schneider and Lamport, 1985] дан прекрасный обзор отказоустойчивого программирования и описаны некоторые общие модели решений. Отказоустойчивости в распределенных системах посвящена монография [Jalote, 1994].

В книгах [Bacon, 1998] и [Bernstein and Lewis, 1993] несколько глав уделено распределенным системам и вопросам отказоустойчивости. В этих книгах описаны транзакции и восстановление в распределенных базах данных (одном из основных приложений, в которых отказоустойчивость является необходимым требованием). Отличный источник информации по всем вопросам распределенных систем (по состоянию на 1993 год) — это сборник статей [Mullender, 1993], написанных ведущими специалистами в этой области.

Литература

Bacon, J. 1998. *Concurrent Systems: Operating Systems, Database and Distributed Systems: An Integrated Approach*, 2nd ed. Reading, MA: Addison-Wesley.

Bernstein, A. J., and P. M. Lewis. 1993. *Concurrency in Programming and Database Systems.* Boston: Jones and Bartlett.

Birman, K. P., and T. A. Joseph. 1987. Reliable communication in the presence of failures. *ACM Trans. on Computer Systems* 5, 1 (February): 47–76.

Birman, K. P., A. Schiper, and A. Stephenson. 1991. Lightweight, causal, and atomic group multicast. *ACM Trans. on Computing Systems* 9, 3 (August): 272–314.

Brinch Hansen, P. 1995. *Studies in Computational Science.* Englewood Cliffs, NJ: Prentice-Hall.

Carriero, N., and D. Gelernter. 1989. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys* 21, 3 (September): 323–358.

Carriero, N., D. Gelernter, and J. Leichter. 1986. Distributed data structures in Linda. *Thirteenth ACM Symp. on Principles of Prog. Langs.*, January, pp. 236–242.

Chandrasekaran, S., and S. Venkatesan. 1990. A message-optimal algorithm for distributed termination detection. *Journal of Parallel and Distributed Computing* 8, 245–292.

Chandy, K. M., L. M. Haas, and J. Misra. 1983. Distributed deadlock detection. *ACM Trans. on Computer Systems* 1, 2 (May): 144–156.

Chandy, K. M., and Lamport, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems* 3, 1 (February): 63–75.

Chandy, K. M., and Misra, J. 1984. The drinking philosophers problem. *ACM Trans. on Prog. Languages and Systems* 6, 4 (October): 632–646.

Chang, E. J.-H. 1982. Echo algorithms: depth parallel operations on general graphs. *IEEE Trans. on Software Engr.* 8, 4 (July): 391–401.

Dijkstra, E. W., W. H. J. Feijen, and A. J. M. Van Gasteren. 1983. Derivation of a termination detection algorithm for distributed computation. *Information Processing Letters* 16, 5 (June): 217–219.

Dijkstra, E. W., and C. S. Scholten. 1980. Termination detection in diffusing computations. *Information Processing Letters* 11, 1 (August): 1–4.

Finkel, R., and U. Manber. 1987. DIB — distributed implementation of backtracking. *ACM Trans. on Prog. Languages and Systems* 9, 2 (April): 235–256.

Fox, G. C., M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. 1988. *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems.* Englewood Cliffs, NJ: Prentice-Hall.

Francez, N. 1980. Distributed termination. *ACM Trans. on Prog. Languages and Systems* 2, 1 (January): 42–55.

Gentleman, W. M. 1981. Message passing between sequential processes: the reply primitive and the administrator concept. *Software — Practice and Experience* 11, 435–466.

Hwang, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* New York: McGraw-Hill.

Jalote, P. 1994. *Fault Tolerance in Distributed Systems.* Englewood Cliffs, NJ: Prentice-Hall.

Knapp, E. 1987. Deadlock detection in distributed databases. *ACM Computing Surveys* 19, 4 (December): 303–328.

Lamport, L. 1978. Time, clocks, and the ordering of events in distributed systems. *Comm. ACM* 21, 7 (July): 558–565.

Lamport, L. 1982. An assertional correctness proof of a distributed algorithm. *Science of Computer Prog.* 2, 3 (December): 175–206.

Lamport, L., R. Shostak, and M. Pease. 1982. The Byzantine generals problem. *ACM Trans. on Prog. Languages and Systems* 3, 3 (July): 382–401.

LeLann, G. 1977. Distributed systems: Towards a formal approach. *Proc. Information Processing 77*, North-Holland, Amsterdam, pp. 155–160.

Maekawa, M., A. E. Oldehoeft, and R. R. Oldehoeft. 1987. *Operating Systems: Advanced Concepts.* Menlo Park, CA: Benjamin/Cummings.

Marzullo, K., and S. S. Owicki. 1983. Maintaining the time in a distributed system. *Proc. Second ACM Symp. on Principles of Distr. Computing*, August, pp. 295–305.

McCurley, E. R., and F. B. Schneider. 1986. Derivation of a distributed algorithm for finding paths in directed networks. *Science of Computer Prog.* 6, 1 (January): 1–9.

Misra, J. 1983. Detecting termination of distributed computations using markers. *Proc. Second ACM Symp. on Principles of Distr. Computing*, August, pp. 290–294.

Morgan, C. 1985. Global and logical time in distributed algorithms. *Information Processing Letters* 20, 4 (May): 189–194.

Mullender, S., ed. 1993. *Distributed Systems*, 2nd ed. Reading, MA: ACM Press and Addison-Wesley.

Quinn, M. J. 1994. *Parallel Computing: Theory and Practice*. New York: McGraw-Hill.

Rana, S. P. 1983. A distributed solution of the distributed termination problem. *Information Processing Letters* 17, 1 (July): 43–46.

Raynal, M. 1986. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press.

Raynal, M. 1988. *Distributed Algorithms and Protocols*. New York: Wiley.

Schneider, F. B. 1980. Ensuring consistency in a distributed database system by use of distributed semaphores. *Proc. of Int. Symp. on Distributed Databases*, March, pp. 183–189.

Schneider, F. B. 1982. Synchronization in distributed programs. *ACM Trans. on Prog. Languages and Systems* 4, 2 (April): 125–148.

Schneider, F. B., D. Gries, and R. D. Schlichting. 1984. Fault-tolerant broadcasts. *Science of Computer Prog.* 4: 1–15.

Schneider, F. B., and L. Lamport. 1985. Paradigms for distributed programs. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, Lecture Notes in Computer Science, vol. 190. Berlin: Springer-Verlag, pp. 431–480.

Wilkinson, B., and M. Allen. 1999. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Englewood Cliffs, NJ: Prentice-Hall.

Упражнения

9.1. В разделе 9.1 описан способ представления разреженных матриц и их умножения с помощью распределенного набора задач:

   а) пусть матрица a представлена строками, как было описано. Постройте код для транспонирования a;

   б) реализуйте программу в листинге 9.1 и поэкспериментируйте с ней. Сначала входные матрицы составьте вручную, чтобы для них был известен результат. Затем напишите небольшую программу порождения больших матриц, возможно, с помощью генератора случайных чисел. Определите производительность программы для больших матриц. Напишите краткий отчет о проведенных тестах и полученных результатах.

9.2. В разделе 3.6 представлена программа в стиле "портфель задач" для умножения плотных матриц:

   а) постройте распределенную версию этой программы. В частности, используйте модель "управляющий-рабочие";

   б) реализуйте ответ к пункту *а* и программу в листинге 9.1. Сравните производительность программ, используя в качестве тестовых большие разреженные матрицы. (Программы, конечно, будут представлять матрицы по-разному.) Напишите краткий отчет, объяснив результаты тестирования и сравнив затраты времени и памяти обеих программ.

9.3. В программе адаптивной квадратуры (см. листинг 9.2) используется фиксированное количество задач (интервал от `a` до `b` делится на заданное количество отрезков). Алгоритм в листинге 3.18 полностью адаптивный; он начинает работу с одной задачей и создает любое нужное их количество:

а) измените программу в листинге 9.2, чтобы использовать полностью адаптивный подход. Это значит, что рабочие процессы не должны вычислять площадь, используя рекурсивную процедуру. Вам необходимо разобраться, как обнаружить завершение работы программы!

б) измените ответ к пункту *б*, чтобы в нем использовался параметр *порог* `T`, задающий максимальный размер портфеля задач. После того как будет создано это число задач, управляющий процесс должен указать рабочему, что выполнять вычисления нужно рекурсивно, не генерируя дополнительные задачи;

в) реализуйте программу в листинге 9.2 и ответы к пунктам *а* и *б*. Сравните производительность трех полученных программ. Проведите серию экспериментов, напишите краткий отчет о проведенных экспериментах и полученных результатах.

9.4. Быстрая сортировка — это рекурсивный метод сортировки, при котором массив делится на меньшие части, которые затем объединяются. Разработайте программу, реализующую быструю сортировку с помощью модели "управляющий-рабочие". Массив целых чисел `a[1:n]` является локальным по отношению к управляющему процессу. Для сортировки используйте `w` рабочих процессов. После завершения программы результат должен сохраняться в массиве `a`. Не используйте разделяемые переменные. Объясните свое решение и обоснуйте его структуру.

9.5. *Задача о восьми ферзях.* Нужно расставить на шахматной доске восемь ферзей так, чтобы они не атаковали друг друга. Разработайте программу, которая строит все 92 решения этой задачи. Используйте модель "управляющий-рабочие". Пусть управляющий процесс помещает восемь начальных позиций ферзей в разделяемый портфель. Для расширения частичных решений используйте `w` рабочих процессов. Обнаружив полное решение, рабочий процесс должен передать его управляющему. Программа должна найти все решения и завершить работу. Не используйте разделяемые переменные.

9.6. Рассмотрим задачу подсчета числа слов в словаре, в которых ни одна буква не повторяется (буквы уникальны). Строчные и прописные буквы не различаются. (В большинстве систем Unix есть один или несколько словарей, например в файле `/usr/dict/words`.)

Напишите распределенную параллельную программу для решения этой задачи. Используйте модель "управляющий-рабочие" и `w` рабочих процессов. В конце программы управляющий процесс должен вывести число слов, состоящих из уникальных букв, и вывести все самые длинные слова. Если ваши рабочие процессы имеют доступ к разделяемой памяти, можете использовать разделяемую копию файла словаря, но если они выполняются на отдельных машинах, то каждый должен использовать собственную копию.

9.7. *Задача коммивояжера (КОМ).* Эта классическая комбинаторная задача имеет практическое применение, поскольку является основой, например, планирования полетов и работы персонала в авиакомпаниях.

Дано `n` городов и симметричная матрица `dist[1:n,1:n]`. Значением элемента `dist[i,j]` является расстояние между городами `i` и `j`, например, в воздушных милях.[18] Коммивояжер начинает путь в городе `1` и должен посетить по одному разу каждый город, закончив путь снова в городе `1`. Найти путь, минимизирующий расстояние, кото-

[18] Неявно предполагается, что расстояния определены для всех пар городов (граф городов полон), причем расстояния могут не удовлетворять евклидовой метрике. — *Прим. ред.*

рое коммивояжеру придется проехать, а результат сохранить в векторе `bestpath[1..n]`. Значением `bestpath` должна быть такая перестановка целых чисел от 1 до n, при которой сумма расстояний между соседними городами и расстояния от последнего города к первому будет наименьшей:

а) разработайте распределенную параллельную программу для решения поставленной задачи, используя парадигму "управляющий-рабочие". Выберите, какая часть работы будет образовывать одну задачу. Задач не должно быть слишком много или слишком мало. Нужно также отбрасывать задачи, которые не могут привести к лучшему результату, чем уже вычисленный;

б) в точном решении КОМ необходимо рассмотреть все возможные пути, но всего их `n!`. Поэтому были разработаны различные эвристические правила, одно из которых называется *алгоритмом ближайшего соседа*. Начиная с города 1, сначала посещают город, скажем, c, который ближе всего к 1. Затем — город, ближайший к c. Продолжая двигаться таким образом, посещают все города и возвращаются в город 1. Напишите программу, реализующую этот алгоритм;

в) еще одно эвристическое правило называется *алгоритмом ближайшей вставки*. Сначала находят пару городов с наименьшим расстоянием. Затем находят непосещенный город, который ближе всех к одному из этих городов, и вставляют между ними. Продолжают искать города, ближайшие к одному из городов частичного маршрута, и вставлять их между городами так, чтобы получить минимальное увеличение общей длины пути. Напишите программу, реализующую этот алгоритм;

г) сравните производительность и точность программ к пунктам *а–в*. Каково время их выполнения? Насколько хороши или плохи полученные приближенные значения? Насколько большую задачу можно решить с помощью приближенных алгоритмов по сравнению с точным алгоритмом? Проведите эксперименты с несколькими наборами исходных данных различных размеров. Напишите отчет о проведенных тестах и полученных результатах;

д) есть другие эвристические алгоритмы и методы локальной оптимизации для решения КОМ. Например, есть методы, которые называются разрезанием плоскостей (cutting planes) и имитацией отжига (simulated annealing). Начните с поиска хороших ссылок на КОМ. Выберите один или несколько хороших алгоритмов, запрограммируйте их и проведите серию экспериментов, чтобы оценить их качество (по времени выполнения и точности полученных решений).

9.8. В листинге 9.3 приведена программа выделения областей в изображении:

а) реализуйте программу, используя библиотеку MPI;

б) измените ответ к пункту *а*, чтобы избавиться от отдельного управляющего процесса. *Указание*. Используйте глобальные примитивы взаимодействия из библиотеки MPI;

в) сравните производительность этих программ. Создайте тестовый набор изображений, определите производительность каждой программы для них при разных количествах рабочих процессов. Напишите отчет с объяснением и анализом полученных результатов.

9.9. В разделе 9.2 описана задача выделения областей. Рассмотрим другую задачу обработки изображений, которая называется *сглаживанием*. Пусть изображение представлено матрицей пикселей, как в разделе 9.2. Нужно его сгладить, убрав "пики" и "зазубрины". Начав с исходного изображения, затемните (сделайте равными 0) все светлые пиксели, у которых *как минимум* `d` *соседей не освещены*. Каждый пиксель исходного изображения нужно рассматривать независимо, поэтому потребуется память для матрицы нового изображения. Затем возьмите полученное изображение и повторите обработку, затем-

няя все (новые) пиксели, у которых не освещены хотя бы d соседей. Продолжайте, *пока обработка будет изменять изображение* (таким образом, необходимое количество повторений алгоритма заранее неизвестно):

а) напишите параллельную программу для решения этой задачи; используйте алгоритм пульсации. В программе нужно использовать w рабочих процессов. Разделите изображение на w полос одинакового размера и назначьте для каждой полосы отдельный рабочий процесс. Можно предположить, что n кратно w;

б) проверьте свою программу для разных изображений, количеств рабочих процессов и значений параметра d. Напишите краткий отчет, объясняющий тесты и их результаты.

9.10. В листинге 9.4 приведена программа для игры "Жизнь". Измените программу, чтобы использовались w рабочих процессов, и каждый рабочий процесс работал с полосой или блоком клеток. Реализуйте программу, используя язык распределенного программирования или последовательного с библиотекой функций, например MPI. Отобразите выход программы на графический дисплей. Проведите эксперименты, составьте краткий отчет, объяснив проведенные эксперименты и полученные результаты. Сходится ли игра?

9.11. Игра "Жизнь" имеет очень простые организмы и правила. Разработайте более сложную игру, которую можно моделировать с помощью клеточных автоматов, но не делайте ее слишком сложной! Например, промоделируйте взаимодействие акул и рыб, кроликов и лисиц или возгорание деревьев и кустов при лесном пожаре.

Выберите задачу для моделирования и постройте первоначальный набор правил; можно использовать случайные числа, чтобы сделать взаимодействие не фиксированным, а вероятностным. Реализуйте модель в виде клеточного автомата, поэкспериментируйте с ней, модифицируйте правила, чтобы результаты не были тривиальными. Например, можно стремиться к сохранению популяции. Напишите отчет с описанием игры и полученными результатами.

9.12. В разделе 9.4 описан алгоритм "зонд-эхо" для вычисления топологии сети. Эту задачу можно решить, используя алгоритм пульсации. Предположим (как и в разделе 9.4), что процесс может взаимодействовать только с соседями, и что вначале процессу известны только его соседи. Разработайте алгоритм пульсации, по которому процессы многократно обмениваются информацией с соседями. После завершения программы *каждый* процесс должен знать топологию всей сети. Вам придется продумать, как процессы должны обмениваться данными и определять, что нужно заканчивать работу.

9.13. Пусть есть n2 процессов, расположенных в виде квадратной сетки. Каждый процесс может взаимодействовать только с соседями слева, справа, сверху и снизу. (У процессов в углах есть только по два соседа, а у остальных на границах сетки — по три.) У каждого процесса есть локальное целочисленное значение v. Напишите алгоритм пульсации для вычисления суммы всех n2 значений. После завершения программы сумма должна быть известна всем процессам.

9.14. В разделе 9.3 описаны два алгоритма умножения распределенных матриц:

а) измените алгоритмы, чтобы в них использовались по W рабочих процессов, где W намного меньше n (выберите W и n, упрощающие арифметические выкладки);

б) сравните теоретическую производительность ответов к пункту *а*. Каким будет необходимое число сообщений в каждой программе для заданных значений W и n? Некоторые сообщения могут находиться в пути одновременно. В каких случаях самая длинная цепочка сообщений, которые не могут перекрываться, для каждой программы будет кратчайшей? Каковы размеры сообщений в программах? Какой объем локальной памяти нужен для каждой из программ?

в) реализуйте ответы к пункту *а*, используя язык программирования или библиотеку процедур типа MPI. Сравните производительность программ для матриц разного размера и при разном количестве рабочих процессов. (Простой способ проверить правильность программ — взять обе исходные матрицы из единиц. В результате должна получиться матрица, у которой все элементы имеют значение `n`.) Напишите отчет с объяснением проведенных экспериментов и полученных результатов.

9.15. В листинге 9.6 приведен алгоритм блочного умножения матриц:

а) укажите расположение значений `a` и `b` после первоначального переупорядочения для `n = 6` и для `n = 8`;

б) измените программу, чтобы использовать `W` рабочих процессов (`W` является четной степенью 2 и делит `n`). Например, если `n = 1024`, то `W` может иметь значения `4`, `16`, `64` или `256`. Каждый рабочий процесс отвечает за один блок матриц. Покажите *все* подробности кода программы;

в) реализуйте ответ к пункту *б*, например, на языке `C` с использованием библиотеки `MPI`. Проведите эксперименты для оценки производительности программы при разных значениях `n` и `W`. Напишите отчет о проведенных экспериментах и полученных результатах. (Матрицы `a` и `b` можно инициализировать единицами, тогда конечным значением всех элементов матрицы `c` будет `n`.)

9.16. Для сортировки `n` значений можно использовать конвейер следующим образом. (Это не очень эффективный алгоритм сортировки, но это всего лишь упражнение!) Даны `W` рабочих и один управляющий процесс. Можно считать, что `n` кратно `W` и каждый рабочий процесс может хранить не больше, чем `n/W + 1` значений одновременно. Процессы образуют закрытый конвейер. Вначале у управляющего процесса есть `n` неотсортированных значений. Он передает их по одному рабочему процессу `1`. Рабочий процесс `1` оставляет часть полученных значений себе, а остальные передает рабочему процессу `2`. Процесс `2` тоже оставляет себе часть полученных значений, а остальные передает процессу `3` и т.д. В конце концов рабочие процессы возвращают значения управляющему (они могут делать это напрямую):

а) разработайте код для управляющего и рабочих процессов, чтобы управляющему возвращались *отсортированные* значения. Каждый процесс может вставлять или удалять значение из списка, но не может использовать внутреннюю процедуру сортировки;

б) сколько сообщений используется в вашем алгоритме? Ответ на этот вопрос представьте в виде функции от `n` и `W`. Покажите, как получена эта зависимость.

9.17. Даны `n` процессов, каждый из них соответствует узлу связанного графа. Каждый узел может связываться со своими соседями. Остовное дерево графа — это дерево, включающее все узлы графа и подмножество его ребер.

Напишите программу для построения остовного дерева "на лету". Не нужно сначала строить топологию; вместо этого дерево строится "от земли" вверх с помощью процессов, взаимодействующих со своими соседями, чтобы решить, какие ребра нужно добавить в дерево, а какие — пропустить. Можно считать, что процессы имеют уникальные индексы.

9.18. Дополните алгоритм "зонд-эхо" для вычисления топологии сети (см. листинг 9.10) при условии, что во время вычислений линии связи могут отказывать, а позже восстанавливаться. (Сбой процессора моделируется отказом всех его связей.) Можно считать, что при отказе связь "молча выбрасывает" недоставленные сообщения.

Определите дополнительные примитивы, необходимые для выявления сбоя или восстановления связи, кратко опишите их реализацию. Может также понадобиться изменить примитив `receive`, чтобы при сбое в канале он возвращал код ошибки. Алгоритм должен завершаться при условии отсутствия сбоев и восстановлений линий связи в течение

интервала времени, достаточного для того, чтобы все вершины и ребра пришли в соответствие с полученной топологией.

9.19. Даны n процессов. Предположим, что примитив `broadcast` рассылает сообщения от одного процесса всем n процессам, что он надежен и сохраняет полное упорядочение сообщений, т.е. все процессы получают рассылаемые сообщения *в одном и том же порядке*:

а) используя примитив `broadcast` (и, конечно, примитив `receive`), разработайте справедливое решение задачи распределенного взаимного исключения. Постройте протоколы входа и выхода, выполняемые процессами перед критической секцией и после нее. Не используйте дополнительные процессы; все n процессов должны взаимодействовать между собой напрямую;

б) обдумайте, как реализовать неделимую рассылку сообщений. Какие при этом нужно решить задачи? Как их решить?

9.20. Рассмотрим следующие три процесса, которые взаимодействуют с помощью передачи сообщений.

```
chan chA(...), chB(...), chC(...);
process A { ...
  send chC(...); send chB(...); receive chA(...);
  send chC(...); receive chA(...); }
process B { ...
  send chC(...); receive chB(...);
  receive chB(...); send chA(...); }
process C { ...
  receive chC(...); receive chC(...);
  send chA(...); send chB(...); receive chC(...); }
```

Предположим, что у каждого процесса есть логические часы с нулевым начальным значением, и для добавления метки времени в сообщения и обновления часов при получении сообщений используются правила, описанные в разделе 9.5.

9.21. Какими будут конечные значения логических часов в каждом процессе? Возможны различные ответы.

Рассмотрим реализацию распределенных семафоров в листинге 9.11:

а) предположим, что есть четыре пользовательских процесса. Первый пользователь инициирует операцию `V`, когда его логические часы имеют значение `0`. Второй и третий пользователи инициируют операции `P`, когда их логические часы имеют значение `1`. Четвертый пользователь инициирует операцию `V`, когда его логические часы имеют значение `10`. Постройте трассу *всех* событий взаимодействия, происходящих при работе этого алгоритма, и укажите, какие значения логических часов и меток времени соответствуют каждому событию. Проследите содержание очередей сообщений во вспомогательных процессах;

б) в данном алгоритме логические часы есть и у пользовательских, и у вспомогательных процессов, но только вспомогательные процессы взаимодействуют между собой и принимают решения, основанные на полном упорядочении событий связи. Допустим, пользовательские процессы не имеют логических часов и не добавляют метки времени к своим сообщениям. Останется ли алгоритм правильным? Если да, объясните, почему, а если нет, покажите на примере, что может работать неправильно;

в) допустим, программу изменили так, чтобы пользовательские процессы, желая выполнить операцию `P` или `V`, рассылали сообщения всем вспомогательным процессам. Например, желая выполнить операцию `V`, пользовательский процесс должен выполнить операцию

```
broadcast semop(i, VOP, lc)
```

и затем обновить свои логические часы. (После рассылки сообщения POP пользователь все равно будет ждать разрешения от своего вспомогательного процесса.) В результате программа упростится, поскольку станут ненужными две первые ветви операторов `if/then/else` во вспомогательных процессах. К сожалению, полученная программа будет ошибочной. Покажите на примере, что будет работать неправильно.

9.22. В решении задачи распределенного взаимного исключения (см. листинг 9.12) используется передача одного маркера, который непрерывно циркулирует между процессами. Допустим, что вместо этого каждый процесс `Helper` может взаимодействовать со всеми вспомогательными процессами, т.е. граф связей полон. Разработайте решение, в котором не используются циркулирующие маркеры. Процессы `Helper` должны пробуждаться, только когда какой-нибудь из процессов `User[i]` пытается войти в критическую секцию или выйти из нее. Ваше решение должно быть справедливым и свободным от взаимных блокировок. Каждый процесс `Helper` должен выполнять один и тот же алгоритм, а процессы `User[i]` — выполнять тот же код, что и в листинге 9.12. (*Указание.* Свяжите маркер с каждой парой процессов, т.е. с каждым ребром графа связей.)

9.23. В листинге 9.13 представлен набор правил для определения окончания работы программы в кольце. Переменная `token` используется для подсчета числа бездействующих процессов. Действительно ли это значение необходимо, или для того, чтобы обнаружить окончание, достаточно знать только цвета процессов? Другими словами, дайте точное объяснение роли переменной `token`.

9.24. *Задача о пьющих философах* [Chandy and Misra, 1984]. Рассмотрим следующее обобщение задачи об обедающих философах. Дан неориентированный граф G; философы связаны с его узлами и могут общаться только с соседями. С каждым ребром графа связана бутылка. Каждый философ циклически проходит через три состояния: спокойный, жаждущий, пьющий. Спокойный философ может стать жаждущим. Перед тем как начать пить, философ должен взять бутылку, связанную с каждым ребром, инцидентным его узлу. Попив, философ снова становится спокойным.

Разработайте решение этой задачи, обеспечив справедливость и отсутствие взаимных блокировок. Все философы должны работать по одному и тому же алгоритму. Для представления бутылок используйте маркеры. Допускается, что спокойный философ отвечает на запросы, поступающие от соседей, о бутылках, которые у него могут быть.

9.25. Дан набор процессов, которые могут взаимодействовать между собой с помощью асинхронной передачи сообщений. *Рассеивающие вычисления* (diffusing computation) — это тип вычислений, при которых главный процесс начинает работу, отправляя сообщения одному или нескольким другим процессам [Dijkstra and Scholten, 1980]. Другой процесс сможет передавать сообщения после того, как сам получит сообщение.

Разработайте схему сигнализации, которая лучше всего подходит для такого типа вычислений и позволяет главному процессу обнаружить окончание вычислений. Используйте алгоритм "зонд-эхо" и другие идеи из раздела 9.5. (*Указание.* Ведите подсчет сообщений и сигналов.)

9.26. В правилах распределенного определения окончания работы (см. листинг 9.14) предполагалось, что граф полон и процесс получает сообщения только из одного канала. Решите следующие упражнения как отдельные задачи:

а) расширьте правила передачи маркера для работы с произвольным связным графом;

б) расширьте правила передачи маркера для работы в ситуациях, когда процесс получает сообщения из нескольких каналов (конечно же, по одному).

9.27. Рассмотрим следующий вариант правил из листинга 9.14 для того, чтобы обнаружить окончание работы в полном графе. Получая обычное сообщение, процесс должен совершить те же действия, что в листинге 9.14, а получая маркер, — следующие операции.

```
if (color[i] == blue)
  token = blue;
else
  token = red;
color[i] = blue;
переменной j присвоить индекс канала следующего ребра в цикле C;
send ch[j](token);
```

Другими словами, значение переменной `token` больше не изменяется и не проверяется. Можно ли с помощью этого нового правила обнаружить окончание работы? Если да, объясните, когда известно, что вычисления закончены. Если нет, объясните, почему этих правил недостаточно.

9.28. В листинге 8.14 показано, как реализовать дублируемые файлы, используя по одной блокировке на копию. В этой программе процесс, желая обновить файл, должен получить каждую блокировку. Допустим, что вместо этого используются `n` маркеров, и что маркер без необходимости не перемещается. Вначале у каждого файлового сервера есть один маркер:

а) измените программу в листинге 8.14, чтобы вместо блокировок в ней использовались маркеры. Реализуйте операцию `read` путем чтения локальной копии файла, а операцию `write` — обновления всех копий. Когда клиент открывает файл для чтения, его сервер должен получить один маркер; для записи ему нужны все `n` маркеров. Ваше решение должно быть справедливым и свободным от взаимных блокировок;

б) измените ответ к пункту *а*, чтобы использовать взвешенное голосование маркерами. Когда клиент открывает файл для чтения, его сервер должен получить любые `readWeight` маркеров. Можно считать, что значения `readWeight` и `writeWeight` удовлетворяют условиям, описанным в конце раздела 8.4. Покажите, как в программе обрабатываются метки времени файлов, а также, как определяется, какая копия является текущей, когда файл открывается для чтения;

в) сравните ответы к пунктам *а* и *б*. Сколькими сообщениями серверы должны обмениваться для разных клиентских операций в каждом из решений? Определите наилучший и наихудший случаи, рассмотрите их подробно.

Глава 10

Реализация языковых механизмов

В данной главе представлены способы реализации различных языковых механизмов, описанных в главах 7 и 8: асинхронной и синхронной передачи сообщений, удаленного вызова процедур (RPC) и рандеву. Сначала показано, как реализовать асинхронную передачу сообщений с помощью ядра. Затем асинхронная передача сообщений используется для реализации синхронной передачи сообщений и защищенного взаимодействия. Далее демонстрируется реализация RPC с помощью ядра, рандеву с помощью асинхронной передачи сообщений и, наконец, рандеву (и совместно используемые примитивы) в ядре.

Реализация синхронной передачи сообщений сложнее, чем асинхронной, поскольку операторы как приема, так и передачи являются блокирующими. Аналогично реализация рандеву сложнее, чем реализация RPC или асинхронной передачи сообщений, поскольку для рандеву нужны двусторонние взаимодействие и синхронизация.

Отправной точкой рассматриваемых реализаций является ядро с разделяемой памятью из главы 6. Таким образом, хотя программы, использующие передачу сообщений, RPC и рандеву, обычно пишутся для машин с распределенной памятью, их можно выполнять и на машинах с разделяемой памятью. Используя так называемую *распределенную разделяемую память*, можно выполнять программы с разделяемыми переменными на машинах с распределенной памятью, даже если они были написаны для работы на машинах с разделяемой памятью. Последний раздел главы посвящен реализации распределенной разделяемой памяти.

10.1. Асинхронная передача сообщений

В этом разделе представлены две реализации асинхронной передачи сообщений. В первой из них к ядру для разделяемой памяти из главы 6 добавлены каналы и примитивы передачи сообщений. Эта реализация подходит для работы на одном процессоре или на мультипроцессоре с разделяемой памятью. Во второй реализации ядро с разделяемой памятью дополнено до распределенного ядра, которое может работать в многопроцессорной системе или в сети из отдельных машин.

10.1.1. Ядро для разделяемой памяти

Каждый канал программы представлен в ядре *дескриптором канала*. Дескриптор канала содержит заголовки списков сообщений и заблокированных процессов. В списке сообщений находятся сообщения, поставленные в очередь; в списке блокированных процессов — процессы, ожидающие получения сообщений. Хотя бы один из этих списков всегда пуст, поскольку, если есть доступное сообщение, процесс не блокируется, а если есть заблокированный процесс, то сообщения не ставятся в очередь.

Дескриптор создается с помощью примитива ядра `createChan`, который вызывается по одному разу для каждой декларации `chan` в программе до создания процессов. Массив каналов создается либо вызовом примитива `createChan` для каждого элемента, либо одним вызовом примитива `createChan` с параметром, указывающим размер массива. Примитив `createChan` возвращает имя (индекс или адрес) дескриптора.

Оператор send реализован с помощью примитива sendChan. Сначала процесс-отправитель вычисляет выражения и собирает значения в единое сообщение, которое обычно записывает в стек выполнения процесса, передающего сообщение. Затем вызывается примитив sendChan; его аргументами являются имя канала (возвращенное из вызова createChan) и само сообщение. Примитив sendChan сначала находит дескриптор канала. Если в списке заблокированных процессов есть хотя бы один процесс, то оттуда удаляется самый старый процесс, а сообщение копируется в его адресное пространство. После этого дескриптор процесса помещается в список готовых к работе. Если заблокированных процессов нет, сообщение необходимо сохранить в списке сообщений дескриптора, поскольку передача является неблокирующей операцией, и, следовательно, отправителю нужно позволить продолжать выполнение.

Пространство для сохраненного сообщения можно выделять динамически из единого буферного пула, или с каждым каналом может быть связан отдельный коммуникационный буфер. Однако асинхронная передача сообщений поднимает важный вопрос реализации: что, если пространство ядра исчерпано? У ядра есть два выхода: либо остановить выполнение программы из-за переполнения буфера, либо заблокировать передающий процесс, пока не появится достаточно места.

Остановка программы — это решительный шаг, поскольку свободное пространство может вскоре и появиться, но программист сразу получает сигнал о том, что сообщения производятся быстрее, чем потребляются (это обычно говорит об ошибке). С другой стороны, блокировка передающего процесса нарушает неблокирующую семантику оператора send и усложняет ядро, создавая дополнительный источник блокировок. И здесь автор параллельной программы не может ничего предполагать о скорости и порядке выполнения процессов. Ядра операционных систем блокируют отправителей сообщений и при необходимости выгружают заблокированные процессы из памяти в файл подкачки, поскольку должны избегать отказов системы. Однако для языков программирования высокого уровня приемлемым выбором является остановка программы.

Оператор receive реализуется с помощью примитива receiveChan. Его аргументами являются имя канала и адрес буфера сообщений. Действия примитива receiveChan дуальны действиям примитива sendChan. Сначала ядро находит дескриптор, соответствующий выбранному каналу, затем проверяет его список сообщений. Если список не пуст, первое сообщение из него удаляется и копируется в буфер сообщений получателя. Если список сообщений пуст, процесс-получатель добавляется в список заблокированных процессов. Получив сообщение, процесс-адресат распаковывает сообщение из буфера в соответствующие переменные.

Четвертый примитив, emptyChan, используется для реализации функции empty(ch). Он просто находит дескриптор и проверяет, не пуст ли список сообщений. В действительности структуры данных ядра находятся не в защищенной области, и выполняемый процесс может сам проверять свой список сообщений. Критическая секция не нужна, поскольку процессу нужно просмотреть только заголовок списка сообщений.

В листинге 10.1 показаны схемы всех четырех примитивов. Эти примитивы добавлены к однопроцессорному ядру (см. листинг 6.1). Значением executing является адрес дескриптора процесса, выполняемого в данный момент, а dispatcher — это процедура, планирующая работу процессов на данном процессоре. Действия примитивов sendChan и receiveChan очень похожи на действия примитивов P и V в семафорном ядре (см. листинг 6.4). Основное отличие состоит в том, что дескриптор канала содержит список сообщений, тогда как дескриптор семафора — только его значение.

Ядро в листинге 10.1 можно изменить для работы на мультипроцессоре с разделяемой памятью, используя методику, описанную в разделе 6.2. Основное требование состоит в том, что структуры данных ядра нужно хранить в памяти, доступной всем процессорам, а для защиты критических секций кода ядра, дающих доступ к разделяемым данным, использовать блокировки.

Листинг 10.1. Асинхронная передача сообщений в однопроцессорном ядре

```
int createChan(int msgSize) {
  получить дескриптор пустого канала и инициализировать его;
  возвратить индекс или адрес дескриптора;
  dispatcher();
}

proc sendChan(int chan; byte msg[*]) {
  найти дескриптор канала chan;
  if (список заблокированных процессов пуст) {  # сохранить сообщение
    получить буфер и скопировать в него сообщение msg;
    вставить буфер в конец списка сообщений;
  }
  else {  # передать сообщение получателю
    удалить процесс из списка заблокированных;
    скопировать сообщение msg в адресное пространство процесса;
    вставить процесс в конец списка готовых к работе;
  }
  dispatcher();
}

proc receiveChan(int chan; result byte msg[*]) {
  найти дескриптор канала chan;
  if (список сообщений пуст) {  # заблокировать получатель
    вставить процесс executing в конец списка заблокированных;
    записать адрес сообщения msg в дескриптор executing;
    executing = 0;
  }
  else {  # передать сохраненное сообщение получателю
    удалить буфер из списка сообщений;
    скопировать содержимое буфера в сообщение msg;
  }
  dispatcher();
}

bool emptyChan(int chan) {
  bool r = false;
  найти дескриптор канала chan;
  if (список сообщений пуст)
    r = true;
  сохранить r в качестве возвращаемого значения;
  dispatcher();
}
```

10.1.2. Распределенное ядро

Покажем, как для поддержки распределенного выполнения расширить ядро с разделяемой памятью. Главная идея — дублировать ядро, помещая по одной его копии на каждую машину, и обеспечить взаимодействие копий с помощью сетевых примитивов.

В распределенной программе каждый канал хранится на отдельной машине. Предположим пока, что у канала может быть сколько угодно отправителей и только один получатель. Тогда дескриптор канала было бы логично поместить на ту же машину, на которой выполняется получатель. Процесс, выполняемый на этой машине, обращается к каналу так же, как

и при использовании ядра с разделяемой памятью. Но процесс, выполняемый на другой машине, не может обращаться к каналу напрямую; для этого должны взаимодействовать два ядра, выполняемых на этих машинах. Ниже будет описано, как изменить ядро с разделяемой памятью и как для реализации распределенной программы использовать сеть.

На рис. 10.1 показана структура распределенного ядра. Ядро, выполняемое на каждой машине, содержит дескрипторы каналов и процессы, расположенные на данной машине. Как и раньше, в каждом ядре есть обработчики локальных прерываний для вызовов супервизора (внутренние ловушки), таймеры и устройства ввода-вывода. Сеть связи является особым видом устройства ввода-вывода. Таким образом, в каждом ядре есть обработчики прерывания сети и процедуры, которые читают из сети и записывают в нее.

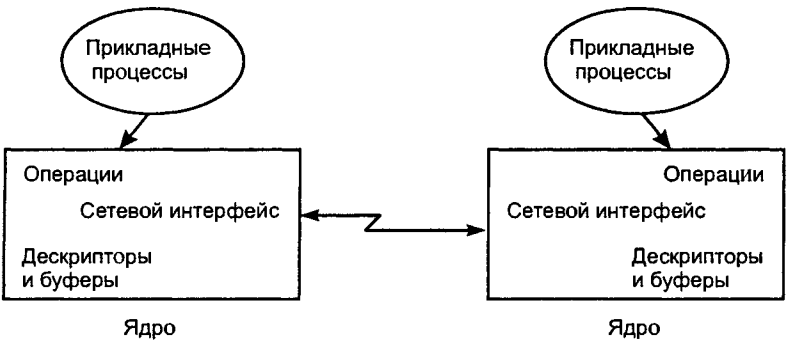


Рис. 10.1. Структура распределенного ядра и взаимодействие

В качестве конкретного примера рассмотрим типичный доступ в сети Ethernet. Контроллер Ethernet состоит из двух независимых частей (для записи и для чтения). С каждой из этих частей в ядре связан обработчик прерывания. Прерывание записи устанавливается, когда операция записи завершается; сам контроллер следит за доступом к сети. Прерывание чтения устанавливается на процессоре, получающем по сети сообщение.

Примитив ядра, выполняемый в результате вызова из прикладного процесса, при передаче сообщения на другую машину вызывает процедуру ядра `netWrite`. Она имеет три аргумента: процессор назначения, вид сообщения (см. ниже) и само сообщение. Сначала процедура `netWrite` получает буфер, форматирует сообщение и записывает его в буфер. Затем, если записывающая часть сетевого контроллера свободна, инициируется запись; в противном случае буфер добавляется в очередь запросов на запись. В обоих случаях происходит выход из `netWrite`. Позже при возникновении прерывания записи связанный с ним обработчик освобождает буфер сообщения, которое только что было записано. Если очередь записи не пуста, обработчик прерывания инициирует следующую сетевую запись.

Ввод из сети обычно обрабатывается в обратном порядке. Когда к ядру приходит сообщение, вызывается обработчик прерывания чтения из сети. Сначала он сохраняет состояние выполняющегося процесса, затем выделяет новый буфер для следующего входного сетевого сообщения. Наконец обработчик чтения распаковывает первое поле сообщения, чтобы определить его вид, и вызывает соответствующий виду примитив ядра.[19]

В листинге 10.2 схематически представлены процедуры сетевого интерфейса. К ним относятся обработчики сетевых прерываний и процедура `netWrite`. Обработчик `ne-`

[19] При другом подходе к обработке сетевого ввода используется процесс-демон, выполняемый вне ядра. Обработчик прерывания просто передает сообщение в канал, из которого демон постоянно выбирает сообщения. Использование демона уменьшает время выполнения собственно обработчика чтения, но увеличивает общее время, необходимое для обработки сетевого ввода. С другой стороны, упрощается ядро, освобождаясь от подробностей обработки сетевых сообщений.

tRead_handler обслуживает сообщения трех видов: SEND, CREATE_CHAN и CHAN_DONE. Эти сообщения передаются одним ядром, а обслуживаются другим, как описано ниже. Чтобы возобновить выполнение прерванного процесса, в конце обработчика netWrite_handler вызывается диспетчерская процедура ядра. Однако в конце обработчика netRead_handler вызывается не диспетчерская процедура, а примитив ядра (в зависимости от вида сообщения), а в этом примитиве, в свою очередь, вызывается диспетчер.

Листинг 10.2. Процедуры сетевого интерфейса

```
type mkind = enum(SEND, CREATE_CHAN, CHAN_DONE);
bool writing = false;       # состояние сетевой записи
другие переменные для очереди записи и буферов передачи;

proc netWrite(int dest; mkind kind; byte data[]) {
  получить буфер; форматировать сообщение и записать его в буфер;
  if (writing)
    вставить буфер сообщения в очередь записи;
  else {
    writing = true;
    начать передачу сообщения по сети;
  }
}

netWrite_handler: {  # вход с запрещенными прерываниями
  сохранить состояние процесса executing;
  освободить текущий буфер передачи;
  writing = false;
  if (очередь записи не пуста) { # начать следующую запись
    удалить из очереди первый буфер; writing = true;
    начать передачу сообщения по сети;
  }
  dispatcher();
}

netRead_handler: {   # вход с запрещенными прерываниями
  сохранить состояние процесса executing;
  получить новый буфер; подготовить контроллер сети к следующему чтению;
  распаковать первое поле сообщения для определения его вида kind;
  if (kind == SEND)
    remoteSend(имя канала, буфер);
  else if (kind == CREATE_CHAN)
    remoteCreate(остальная часть сообщения);
  else  # kind == CHAN_DONE
    chanDone(остальная часть сообщения);
}
```

Для простоты предполагается, что передача по сети происходит без ошибок, и, следовательно, не нужно подтверждать получение сообщений или передавать их заново. Также игнорируется проблема исчерпания области буфера для входящих или исходящих сообщений. На практике для ограничения числа сообщений в буфере используется *управление потоком*. Ссылки на литературу, в которой описаны эти темы, даны в исторической справке.

Канал может храниться локально или удаленно, поэтому его имя должно состоять из двух полей: номера машины и индекса или смещения. Номер машины указывает, где хранится де-

скриптор; индекс определяет положение дескриптора в ядре указанной машины. Примитив createChan также нужно дополнить аргументом, указывающим, на какой машине нужно создать канал. Выполняя примитив createChan, ядро сначала проверяет этот аргумент. Если канал находится на той же машине, ядро создает канал (как в листинге 10.1). В противном случае ядро блокирует выполняемый процесс и передает на удаленную машину сообщение CREATE_CHAN. Это сообщение содержит идентификатор выполняемого процесса. В конце концов локальное ядро получит сообщение CHAN_DONE, которое говорит о том, что на удаленной машине канал создан. Сообщение содержит имя канала и указывает процесс, для которого создан канал. Как показано в листинге 10.2, обработчик netRead_handler, получая это сообщение, вызывает еще один примитив ядра, chanDone, который снимает блокировку процесса, запросившего создание канала, и возвращает ему имя созданного канала.

Демон ядра на другой стороне сети, получив сообщение CREATE_CHAN, вызывает примитив remoteCreate. Этот примитив создает канал и возвращает сообщение CHAN_DONE первому ядру. Таким образом, при создании канала на удаленной машине выполняются следующие шаги.

- Прикладной процесс вызывает локальный примитив createChan.
- Локальное ядро передает сообщение CREATE_CHAN удаленному ядру.
- Обработчик прерывания чтения в удаленном ядре получает это сообщение и вызывает примитив remoteCreate удаленного ядра.
- Удаленное ядро создает канал и передает сообщение CHAN_DONE локальному ядру.
- Обработчик прерывания чтения в локальном ядре получает это сообщение и вызывает примитив chanDone, запускающий прикладной процесс.

В распределенном ядре нужно также изменить примитив sendChan. Примитив sendChan здесь будет намного проще, чем createChan, поскольку операция передачи send является асинхронной. В частности, если канал находится на локальной машине, примитив sendChan должен выполнить такие же операции, как в листинге 10.1. Если канал находится на удаленной машине, примитив sendChan передает на эту машину сообщение SEND. В этот момент выполняемый процесс может продолжить работу. Получив сообщение SEND, удаленное ядро вызывает примитив remoteSend, который, по существу, выполняет те же действия, что и (локальный) примитив sendChan. Его отличие состоит лишь в том, что входящее сообщение уже записано в буфер, поэтому ядру не нужно выделять для него новый буфер.

В листинге 10.3 схематически представлены примитивы распределенного ядра. Примитивы receiveChan и emptyChan по сравнению с листингом 10.1 не изменились, поскольку у каждого канала есть только один получатель, причем расположенный на той же машине, что и канал. Однако если это не так, то для взаимодействия машины, на которой был вызван примитив receiveChan или empty, и машины, на которой расположен канал, нужны дополнительные сообщения. Это взаимодействие аналогично взаимодействию при создании канала — локальное ядро передает сообщение удаленному ядру, которое выполняет примитив и возвращает результаты локальному ядру.

Листинг 10.3. Примитивы распределенного ядра

```
type chanName = rec(int machine, index);

chanName createChan(int machine) {
  chanName chan;
  if (machine локальная) {
    получить дескриптор пустого канала и инициализировать его;
    chan = chanName(номер локальной машины, адрес дескриптора);
  } else {
```

```
    netWrite(machine, CREATE_CHAN, executing);
    вставить дескриптор executing в список приостановленных;
    executing = 0;
  }
  dispatcher();
}

proc remoteCreate(int creator) {
  chanName chan;
  получить дескриптор пустого канала и инициализировать его;
  chan = chanName(номер локальной машины, адрес дескриптора);
  netWrite(creator, CHAN_DONE, chan);
  dispatcher();
}

proc chanDone(int creator; chanName chan) {
  удалить дескриптор процесса creator из списка приостановленных;
  сохранить chan как возвращаемое значение для процесса creator;
  вставить дескриптор процесса creator в список готовых к работе;
  dispatcher();
}

proc sendChan(chanName chan; byte msg[*]) {
  if (chan.machine локальная)
    те же действия, что и в примитиве sendChan в листинге 10.1;
  else
    netWrite(chan.machine, SEND, msg);
  dispatcher();
}

proc remoteSend(chanName chan; int buffer) {
  найти дескриптор канала chan;
  if (список заблокированных пуст)
    вставить buffer в список сообщений;
  else {
    удалить процесс из списка заблокированных;
    скопировать сообщение из buffer в адресное пространство процесса;
    вставить процесс в конец списка готовых к работе;
  }
  dispatcher();
}

proc receiveChan(int chan; result byte msg[*]) {
  те же действия, что и в примитиве receiveChan в листинге 10.1;
}

bool emptyChan(int chan) {
  те же действия, что и в примитиве emptyChan в листинге 10.1;
}
```

10.2. Синхронная передача сообщений

Напомним, что при синхронной передаче сообщений примитивы send и receive являются блокирующими: пытаясь взаимодействовать, процесс должен сначала подождать, пока

к этому не будет готов второй процесс. Это делает ненужными потенциально неограниченные очереди буферизованных сообщений, но требует, чтобы для установления синхронизации получатель и отправитель обменялись управляющими сигналами.

Ниже будет показано, как реализовать синхронную передачу сообщений с помощью асинхронной, а затем — как реализовать операторы ввода, вывода и защищенные операторы взаимодействия библиотеки CSP, используя специальный *учетный процесс* (clearinghouse process). Вторую реализацию можно адаптировать для реализации пространства кортежей Linda (см. раздел 7.7). В исторической справке в конце главы даны ссылки на децентрализованные реализации; см. также упражнения.

10.2.1. Прямое взаимодействие с использованием асинхронных сообщений

Пусть дан набор из n процессов, которые взаимодействуют между собой, используя асинхронную передачу сообщений. Передающая сторона называет нужный ей приемник сообщений, а принимающая сторона может получать сообщения от любого отправителя. Например, исходный процесс S передает сообщение процессу назначения D, выполняя операцию

```
synch_send(D, expressions);
```

Процесс назначения ждет получения сообщения из любого источника при выполнении оператора

```
synch_receive(source, variables);
```

Когда процессы доходят до выполнения этих операторов, идентификатор отправителя и значения выражений передаются в виде сообщения от процесса S процессу D. Затем эти данные записываются в переменные source и variables соответственно. Получатель, таким образом, узнает идентификатор отправителя сообщения.

Описанные примитивы можно реализовать с помощью асинхронной передачи сообщений, используя три массива каналов: sourceReady, destReady и transmit. Первые два массива используются для обмена управляющими сигналами, а третий — для передачи данных. Каналы используются, как показано в листинге 10.4. Процесс-получатель ждет сообщения из своего элемента массива sourceReady; сообщение идентифицирует отправителя. Затем получатель разрешает отправителю продолжить передачу, и передается само сообщение.

Код в листинге 10.4 обрабатывает отправку в указанное место назначения и прием сообщения из любого источника. Если обе стороны должны всегда называть друг друга, то в листинге 10.4 не нужны каналы sourceReady, а получатель может просто передавать отправителю сигнал о готовности к получению сообщения. Оставшихся операций передачи и приема вполне достаточно для синхронизации двух процессов. С другой стороны, если процесс-получатель может называть источник *или* принимать сообщения из любого источника, ситуация становится намного сложнее. (Такая возможность есть в библиотеке MPI.) Тогда либо нужно иметь отдельный канал для каждого пути взаимодействия и опрашивать каналы, либо получающий процесс должен проверять каждое сообщение и сохранять те из них, которые он еще не готов принять. Читателю предоставляется задача изменить реализацию, чтобы она обрабатывала описанную ситуацию (см. упражнения в конце главы).

Листинг 10.4. Синхронное взаимодействие с использованием асинхронных сообщений

разделяемые переменные:

```
chan sourceReady[n](int);          # готовность отправителя
chan destReady[n]();               # готовность получателя
chan transmit[n](byte msg[*]); # передача данных
```

Синхронная передача, выполняемая процессом-отправителем S:

```
собрать выражения в буфер сообщения b;
send sourceReady[D](S);    # сообщить D о готовности
receive destReady[S]();    # ждать готовности D
send transmit[D](b);       # передать сообщение
```

Синхронный прием, выполняемый процессом-получателем D:

```
int source; byte buffer[BUFSIZE];
receive sourceReady[D](source);# ждать любого отправителя
send destReady[source]();     # сообщить ему о готовности
receive transmit[D](buffer); # получить сообщение
распаковать буфер в переменные;
```

10.2.2. Реализация защищенного взаимодействия с помощью учетного процесса

Вновь предположим, что есть n процессов, но они взаимодействуют и синхронизируются с помощью операторов ввода и вывода языка CSP (см. раздел 7.6). Напомним, что они имеют такой вид.

```
Source?port(переменные);          # оператор ввода
Destination!port(выражения);      # оператор вывода
```

Эти операторы *согласуются*, когда процесс Destination выполняет оператор ввода, а процесс Source — оператор вывода, имена портов одинаковы, переменных и выражений поровну, и их типы совпадают.

В языке CSP также представлено защищенное взаимодействие с недетерминированным порядком. Напомним, что операторы защищенного взаимодействия имеют следующий вид.

```
B; C -> S;
```

Здесь B — необязательное логическое выражение (защита), C — оператор ввода или вывода, а S — список операторов. Операторы защищенного взаимодействия используются внутри операторов if или do для выбора из нескольких возможных взаимодействий.

Основное в реализации операторов ввода, вывода и защищенных операторов — объединить в пары процессы, желающие выполнить согласованные операторы взаимодействия. Для подбора пар используется специальный "учетный процесс" CH ("clearinghouse"). Пусть обычный процесс P_i собирается выполнить оператор вывода, в котором процессом назначения является P_j, а процесс P_j — операцию ввода с P_i в качестве источника. Предположим, что имя порта и типы сообщений совпадают. Эти процессы взаимодействуют с учетным процессом и между собой, как показано на рис. 10.2. Каждый из процессов P_i и P_j передает учетному процессу CH сообщение, описывающее желаемое взаимодействие. Процесс CH сначала сохраняет первое из этих сообщений. Получив второе, он возвращается к первому и определяет, согласуются ли операторы двух процессов. Затем CH передает обоим процессам ответ. Получив ответ, процесс P_i отсылает выражения своего оператора вывода процессу P_j, получающему их в переменные своего оператора ввода. В этот момент каждый процесс начинает выполнять код, следующий за оператором взаимодействия.

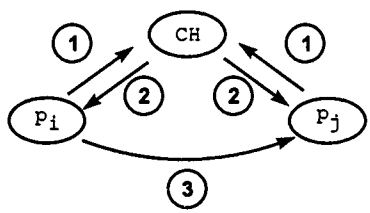


Рис. 10.2. Модель взаимодействия с учетным процессом

Чтобы уточнить программную структуру на рис. 10.2, нужен канал для каждого пути взаимодействия. Один канал используется для сообщений от обычных процессов к учетному. Эти сообщения содержат шаблоны, описывающие возможные варианты согласованных операторов. Каждому обычному процессу для возвращения сообщений от учетного процесса нужен канал ответа. Наконец, нужен один канал данных для каждого обычного процесса, содержащего операторы ввода; такие каналы используются другими обычными процессами.

Пусть у каждого обычного процесса есть уникальный идентификатор (целое число от 1 до n). Эти идентификаторы используются для индексирования каналов данных и каналов ответа. Сообщения-ответы от учетного процесса определяют направление взаимодействия и идентификатор другого процесса. Сообщения по каналу данных передаются в виде массива байтов. Предполагается, что сообщения описывают сами себя, т.е. содержат метки, позволяющие получателю определить типы данных в сообщении.

Доходя до выполнения операторов ввода, вывода или операторов защищенного взаимодействия, обычные процессы передают учетному *шаблоны*. Эти шаблоны используются для подбора соответствующих пар операторов. Каждый шаблон имеет четыре поля.

```
direction, source, destination, port
```

Для операторов вывода поле направления (`direction`) имеет значение `OUT`, для операторов ввода — `IN`. Источник (`source`) и приемник (`destination`) — это идентификаторы отправителя и желаемого получателя (для вывода) или желаемого отправителя и получателя (для ввода). Поле порт (`port`) содержит целое число, которое однозначно определяет порт и, следовательно, типы данных операторов ввода и вывода. Каждому типу порта в исходном тексте программы должен соответствовать определенный номер. Это значит, что каждому явному имени порта должно быть назначено уникальное целочисленное значение, как и для каждого безымянного порта. (Напомним, что имена портов используются в исходной программе, поэтому номера портов можно присвоить статически во время компиляции программы.)

Листинг 10.5 содержит объявления разделяемых типов данных и каналов взаимодействия, а также код, выполняемый обычными процессами при достижении операторов ввода и вывода. Выполняя незащищенный оператор взаимодействия, процесс передает учетному процессу один шаблон и ждет ответа. Найдя согласующийся вызов (как описано ниже), учетный процесс передает ответ. Получив его, процесс-отправитель передает выражения оператора вывода в процесс назначения, который записывает их в переменные своего оператора ввода.

Листинг 10.5. Протоколы для обычных процессов

```
type direction = enum(OUT, IN);
type template =
       rec(direction d; int source; int dest; int port);
type Templates = set of template;

chan match(Templates t);
chan reply[1:n](direction d; int who);
chan data[1:n](byte msg[*]);
```

незащищенный оператор вывода:

```
  Templates t = template(OUT, myid, destination, port);
  send match(t);
  receive reply[myid](direction, who);
  # направление будет OUT, адресат — who
  собрать выражения в буфер сообщения;
  send data[who](buffer);
```

незащищенный оператор ввода:

```
  Templates t = template(IN, source, myid, port);
```

```
send match(t);
receive reply[myid](direction, who);
# направление будет IN и адресат who будет myid
receive data[myid](buffer);
распаковать буфер в локальные переменные;
```

защищенный оператор ввода или вывода:

```
Templates t = ∅;    # множество возможных взаимодействий
for [ логические выражения в защитах, которые истинны ]
  вставить шаблон для оператора ввода или вывода в множество t;
send match(t);        # передать совпадения процессу учета
receive reply[myid](direction, who);
используя значения direction и who, определить, какой из
    защищенных операторов взаимодействия согласован;
if (direction == IN)
  { receive data[myid](buffer);
    распаковать буфер в локальные переменные; }
else      # direction == OUT
  { собрать выражения в буфер сообщения;
    send data[who](buffer); }
выполнить соответствующий защищенный оператор S;
```

Используя защищенный оператор взаимодействия, процесс сначала должен проверить каждую защиту. Для каждого истинного выражения защиты процесс создает шаблон и добавляет его в множество `t`. После вычисления всех выражений защиты процесс передает множество `t` учетному процессу и ждет ответа. (Если `t` пусто, процесс просто продолжает работу.) Полученный ответ указывает процесс, выбранный для взаимодействия, и направление этого взаимодействия. Если направление `OUT`, процесс отсылает сообщение другому процессу, иначе ждет получения данных. После этого процесс выбирает соответствующий защищенный оператор и выполняет его. (Предполагается, что полей `direction` и `who` достаточно, чтобы определить, какой из операторов защищенного взаимодействия был выбран учетным процессом в качестве согласованного. В общем случае для этого нужны также порт и типы данных.)

В листинге 10.6 представлен учетный процесс `CH`. Массив `pending` содержит по одному набору шаблонов для каждого обычного процесса. Если `pending[i]` не пусто, обычный процесс `i` блокируется в ожидании согласованного оператора взаимодействия. Получая новое множество `t`, процесс `CH` сначала просматривает один из шаблонов, чтобы определить, какой из процессов `s` передал его. (Если в шаблоне указано направление `OUT`, то источником является процесс `s`; если указано направление `IN`, то `s` —приемник.) Затем учетный процесс сравнивает элементы множества `t` с шаблонами в массиве `pending`, чтобы увидеть, есть ли согласование. По способу своего создания два шаблона являются согласованными, если их направления противоположны, а порты и источник с приемником одинаковы. Если `CH` находит соответствие с некоторым процессом `i`, он отсылает ответы процессам `s` и `i` (в ответах каждому процессу сообщаются идентификатор другого процесса и направление взаимодействия). В этом случае процесс `CH` очищает элемент `pending[i]`, поскольку процесс `i` больше не заблокирован. Не найдя соответствия ни для одного шаблона во множестве `t`, процесс `CH` сохраняет `t` в элемент `pending[s]`, где `s` — передающий процесс.

Листинг 10.6. Централизованный учетный процесс

```
# декларации глобальных типов и каналв, как в листинге 10.5

process CH {
```

```
  Templates t, pending[1:n] = ([n] ∅);
  ## if pending[i] != ∅, then процесс i заблокирован
  while (true) {
    receive match(t);  # получить новый набор шаблонов
    рассмотреть один из шаблонов в t, чтобы определить отправителя s;
    for [ каждый шаблон в t ] {
      if ( в некотором элементе pending[i] есть согласованная пара ) {
        if (s является отправителем) {
          send reply[s](OUT, i);
          send reply[i](IN, s);
        } else {    # s является адресатом
          send reply[s](IN, i);
          send reply[i](OUT, s);
        }
        pending[i] = ∅;
        break;   # выйти из цикла
      }
    }
    if (согласованные пары не найдены)
      pending[s] = t;
  }
}
```

Разберем работу этих протоколов с помощью конкретного примера. Пусть есть два процесса A и B, которые хотят обменяться данными с помощью следующих защищенных операторов.

```
process A {
  int a1, a2;
  if B!a1 -> B?a2;
  [] B?a2 -> B!a1;
  fi
}
process B {
  int b1, b2;
  if A!b1 -> A?b2;
  [] A?b2 -> A!b1;
  fi
}
```

Начиная выполнять оператор `if`, процесс A строит множество с двумя шаблонами.

```
{ (OUT, A, B, p2), (IN, B, A, p1) }
```

Здесь предполагается, что `p1` — это идентификатор порта в процессе A, а `p2` — в B. Затем A передает эти шаблоны учетному процессу.

Процесс B совершает аналогичные действия и передает учетному процессу такие шаблоны.

```
{ (OUT, B, A, p1), (IN, A, B, p2) }
```

Получив набор шаблонов, учетный процесс определяет, есть ли две возможные согласованные операции. Он выбирает одну из них, отсылает ответы процессам A и B и отбрасывает оба множества шаблонов. Процессы A и B выполняют пару согласованных операций взаимодействия, выбранных учетным процессом. Далее они выполняют операторы взаимодействия внутри тел выбранных защищенных операторов. Для этого каждый процесс передает шаблон учетному процессу, ждет ответа и затем взаимодействует с другим процессом.

Если поиск согласующихся операторов, проводимый учетным процессом в очереди `pending`, всегда идет в одном и том же порядке, то некоторые из заблокированных процессов могут никогда не продолжить свою работу. Однако справедливость можно обеспечить простой стратегией при ус-

ловии, что в программе не может быть взаимных блокировок. Пусть элемент, с которого начинается поиск, указывается значением целочисленной переменной `start`. Получая новый набор шаблонов, процесс СН сначала просматривает элемент `pending[start]`, затем `pending[start+1]` и т.д. Как только процесс `start` получает шанс взаимодействия, учетный процесс СН увеличивает значение переменной `start` до индекса следующего процесса с непустым множеством ожидания. Тогда значение переменной `start` будет циклически проходить по индексам процессов (при условии, что процесс `start` не блокируется навсегда). Таким образом, каждый процесс периодически будет получать шанс быть проверенным первым.

10.3. Удаленный вызов процедур и рандеву

В данном разделе показано, как реализовать RPC и совместно используемые примитивы (включая рандеву) в ядре, а рандеву — с помощью асинхронной передачи сообщений. Ядро, поддерживающее RPC, иллюстрирует, как управлять двусторонним взаимодействием в ядре. На примере реализации рандеву с помощью передачи сообщений представлены дополнительные операции взаимодействия, необходимые для поддержки синхронизации в стиле рандеву. Ядро, обеспечивающее совместно используемые примитивы, демонстрирует реализацию всех различных примитивов взаимодействия одним унифицированным способом.

10.3.1. Реализация RPC в ядре

RPC поддерживает только взаимодействие и не заботится о синхронизации, поэтому реализуется проще всего. Напомним, что программа, использующая RPC, состоит из набора модулей, которые содержат процедуры и процессы. Процедуры (операции), объявленные в части определений модуля, можно вызывать из процессов, которые выполняются в других модулях. Все части модуля располагаются на одной машине, но разные модули могут находиться на разных машинах. (Здесь не рассматривается, как программист указывает размещение модуля; один из таких механизмов был описан в разделе 8.7.)

Процессы, выполняемые в одном модуле, взаимодействуют с помощью разделяемых переменных и синхронизируются, используя семафоры. Предполагается, что на каждой машине есть локальное ядро, реализующее процессы и семафоры, как описано в главе 6, и что ядра содержат процедуры сетевого интерфейса (см. листинг 10.2). Задача состоит в том, чтобы дополнить ядро процедурами и примитивами для реализации RPC.

Между вызывающим процедуру процессом и процедурой возможны три типа отношений.

- Они находятся в одном модуле и, следовательно, на одной машине.
- Они находятся в разных модулях, но на одной машине.
- Они находятся на разных машинах.

В первой ситуации можно использовать обычный вызов процедуры. Нет необходимости использовать ядро, если на этапе компиляции известно, что процедура является локальной. Вызывающий процедуру процесс может просто поместить аргументы в стек и перейти к выполнению процедуры, а после выхода из нее — извлечь из стека ее результаты и продолжить работу.

Для межмодульных вызовов каждую процедуру можно однозначно идентифицировать парой (`machine`, `address`), где `machine` указывает место хранения тела процедуры, а `address` — точку входа в процедуру. Оператор вызова можно реализовать следующим образом.

```
if (machine локальная)
  выполнить обычный вызов по адресу address;
else
  rpc(machine, address, аргументы-значения);
```

Для использования обычного вызова процедура обязательно должна существовать. Это условие выполняется, если нельзя изменять идентификатор процедуры или динамически уничтожать модули. В противном случае, чтобы перед вызовом процедуры убедиться в ее существовании, понадобится входить в локальное ядро.

Чтобы выполнить удаленный вызов процедуры, процесс должен передать на удаленную машину аргументы-значения и приостановить работу до получения результатов. Когда удаленная машина получает сообщение CALL, она создает процесс для выполнения тела процедуры. Перед завершением этого процесса вызывается примитив удаленного ядра, который передает результаты на первую машину.

В листинге 10.7 приведены ядровые примитивы для реализации RPC, а также новый обработчик прерывания чтения. В этих подпрограммах используется процедура распределенного ядра для асинхронной передачи сообщений netWrite (см. листинг 10.2), которая, в свою очередь, взаимодействует с соответствующим обработчиком прерывания.

При обработке удаленного вызова происходят следующие события.

- Вызывающий процесс инициирует примитив rpc, который передает идентификатор вызывающего процесса, адрес процедуры и ее аргументы-значения на удаленную машину.
- Обработчик прерывания чтения в удаленном ядре получает сообщение и вызывает примитив handle_rpc, который создает процесс для обслуживания вызова.
- Серверный процесс выполняет тело процедуры и затем запускает примитив rpcReturn, чтобы возвратить результаты в ядро вызывающего процесса.
- Обработчик прерывания чтения в ядре вызывающего процесса получает возвращаемое сообщение и вызывает процедуру handleReturn, которая снимает блокировку вызывающей процедуры.

В примитиве handle_rpc предполагается, что существует список заранее созданных дескрипторов процессов, обслуживающих вызовы. Это ускоряет обработку удаленного вызова, поскольку не требует накладных расходов на динамическое выделение памяти и инициализацию дескрипторов. Также предполагается, что каждый серверный процесс запрограммирован так, что его первым действием является переход на соответствующую процедуру, а последним — вызов примитива ядра rpcReturn.

Листинг 10.7. Процедуры ядра для реализации RPC

объявления сетевых буферов, свободных дескрипторов, списка отложенных процессов

```
netRead_handler: {  # вход с запрещенными прерываниями
  сохранить состояние процесса executingP;
  выделить новый буфер;
  подготовить сетевой контроллер для следующей операции чтения;
  распаковать первое поле входного сообщения, чтобы определить его вид kind;
  if (kind == CALL)
    handleRPC(caller, address, аргументы-значения);
  else  # kind == RETURN
    handleReturn(caller, результаты);
}

proc rpc(int machine, address; byte args[*]) {
  netWrite(machine, CALL, (executing,address,args));
  вставить дескриптор процесса executing в список отложенных процессов;
  dispatcher();
}
```

```
proc handle_rpc(int caller, address; byte args[*]) {
  получить свободный дескриптор процесса;
  записать в него идентификатор caller;
  поместить address в регистр для процесса;
  распаковать аргументы args и поместить их в стек процесса;
  вставить дескриптор процесса в список готовых к работе;
  dispatcher();
}

proc rpcReturn(byte results[*]) {
  извлечь идентификатор caller вызывающего процесса из дескриптора executing;
  netWrite(машина вызывающего процесса, RETURN, (caller, результаты));
  возвратить дескриптор процесса executing в список свободных дескрипторов;
  dispatcher();
}

proc handleReturn(int caller; byte results[*]) {
  удалить дескриптор caller из списка отложенных процессов;
  поместить результаты в стек вызывающего процесса;
  вставить дескриптор caller в список готовых к работе;
  dispatcher();
}
```

10.3.2. Реализация рандеву с помощью асинхронной передачи сообщений

Рассмотрим, как реализовать рандеву, используя асинхронную передачу сообщений. Напомним, что в рандеву участвуют два партнера: вызывающий процесс, который инициирует операцию с помощью оператора вызова, и сервер, обслуживающий операцию, используя оператор ввода. В разделе 7.3 было показано, как имитировать вызывающий процесс (клиент) и сервер с помощью асинхронной передачи сообщений (см. листинги 7.3 и 7.4). Здесь эта имитация развивается для реализации рандеву.

Главное — реализовать операторы ввода. Напомним, что оператор ввода содержит одну или несколько защищенных операций. Выполнение оператора `in` приостанавливает процесс до появления приемлемого вызова — оператор ввода обслужит тот вызов, для которого выполняется условие синхронизации. Пока не будем обращать внимания на выражения планирования.

Операцию можно обслужить только тем процессом, в котором она объявлена, поэтому ожидающие вызовы можно сохранить в этом процессе. Существует два основных способа сохранения вызовов: с отдельной очередью для каждой операции или с отдельной очередью для каждого процесса. (В действительности существует еще один вариант, который используется в следующем разделе по причинам, объясняемым здесь.) Используем способ с отдельной очередью для каждого процесса, поскольку это приводит к более простой реализации. К тому же, во многих примерах главы 8 использовался один оператор ввода на серверный процесс. Однако у сервера может быть и несколько операторов ввода, обслуживающих разные операции. В таком случае появится необходимость просматривать вызовы, которые не могут быть выбраны в данном операторе ввода.

В листинге 10.8 иллюстрируется реализация рандеву с помощью асинхронной передачи сообщений. Каждый процесс `C`, выполняющий оператор вызова, имеет канал `reply`, из которого он получает результаты вызовов процедур. У каждого серверного процесса `S`, выполняющего оператор ввода, есть канал `invoke`, из которого он получает вызовы, и локальная очередь

pending, которая содержит еще не обслуженные вызовы. Сообщение вызова содержит идентификатор вызывающего процесса, название вызываемой операции и ее аргументы-значения.

Листинг 10.8. Реализация рандеву с помощью асинхронной передачи сообщений

```
разделяемые каналы:
  chan invoke[1:n](int caller, opid; byte values[*]);
  chan reply[1:n](byte results[*]);

оператор вызова в процессе C операции, обслуживаемой процессом S:
  send invoke[S](C, opid, аргументы-значения);
  receive reply[C](результирующие переменные);

оператор ввода в процессе S:
  queue pending;  # ожидающие вызовы
  просмотреть очередь ожидающих вызовов;
  if (некоторый вызов приемлем)
    удалить самый старый из ожидающих приемлемых вызовов;
  else   # получить и проверить другой вызов
    while (true) {
      receive invoke[S](caller, opid, values);
      if (этот вызов приемлем)
        break;
      else
        вставить (caller, opid, values) в pending;
    }
  выполнить соответствующую защищенную операцию;
  send reply[caller](результаты);
```

Чтобы реализовать оператор ввода, серверный процесс S сначала просматривает ожидающие вызовы. Если он находит приемлемый вызов (операцию, соответствующую оператору вызова, для которой истинно условие синхронизации), то S удаляет самый старый из таких вызовов в очереди pending. Иначе S получает новые вызовы до тех пор, пока не найдет приемлемый вызов (сохраняя при этом неприемлемые). Найдя приемлемый вызов, S выполняет тело защищенной операции и после этого передает ответ вызвавшему процессу.

Напомним, что выражение планирования влияет на выбор вызова, если приемлемых вызовов несколько. Выражения планирования можно реализовать, дополнив листинг 10.8 следующим образом. Во-первых, чтобы планировать обработку ожидающих вызовов, серверный процесс должен сначала узнать обо всех таких вызовах. Это вызовы из очереди pending и других очередей, которые могут собираться в канале invoke процесса. Таким образом, перед просмотром очереди pending сервер S должен выполнить такой код.

```
while (not empty(invoke[S])) {
  receive invoke[S](caller, opid, values);
  вставить (caller, opid, values) в очередь pending;
}
```

Во-вторых, если сервер находит приемлемый вызов в очереди pending, он должен просмотреть всю очередь в поисках приемлемого вызова этой же операции с минимальным значением выражения планирования. Если такой вызов найден, то серверный процесс удаляет его из pending и обслуживает вместо первого найденного вызова. Однако цикл в листинге 10.8 изменять не нужно. Если в очереди pending нет приемлемых вызовов, то первый полученный сервером вызов как раз и будет иметь наименьшее значение выражения планирования.

10.3.3. Реализация совместно используемых примитивов в ядре

Рассмотрим реализацию в ядре совместно используемых примитивов (см. раздел 8.3). В ней свойства распределенных ядер для передачи сообщений и RPC сочетаются со свойствами реализации рандеву с помощью асинхронной передачи сообщений. Она также иллюстрирует один из способов реализации рандеву в ядре.

Используя составную нотацию примитивов, операции можно вызывать двумя способами: с помощью синхронных операторов вызова (`call`) и асинхронных — передачи (`send`). Обслуживать операции можно тоже двумя способами, используя процедуры или операторы ввода (но не обоими одновременно). Таким образом, в ядре должно быть известно, какие методы используются для вызова и обслуживания каждой операции. Предположим, что ссылка на операцию имеет вид записи с тремя полями. Первое поле указывает способ обработки операции. Второе определяет машину, на которой должна быть обслужена операция. Для операции, обслуживаемой процедурой `proc`, третье поле записи указывает точку входа процедуры в ядре для RPC. Если же операция обслуживается операторами ввода, третье поле содержит адрес дескриптора операции (его содержимое уточняется через два абзаца).

В рандеву каждая операция обслуживается тем процессом, в котором она объявлена. Следовательно, в реализации рандеву используется по одному набору ожидающих вызовов на каждый серверный процесс. Однако при использовании составной нотации примитивов операцию можно обслуживать операторами ввода в нескольких процессах модуля, в котором она объявлена. Таким образом, серверным процессам одного модуля может понадобиться совместный доступ к ожидающим вызовам. Можно реализовать одно множество ожидающих вызовов для каждого модуля, но тогда все процессы модуля должны будут соперничать в доступе к этому множеству, даже если они обслуживают разные операции. Это приведет к задержкам, связанным с ожиданием доступа ко множеству и просмотром вызовов, которые наверняка не могут быть обслужены. Поэтому для ожидающих вызовов используем несколько множеств, по одному для каждого класса операций, как определено ниже.

Класс операции — это класс эквивалентности транзитивного замыкания отношения "обслуживаются одним и тем же оператором ввода". Например, если в одном операторе ввода есть операции `a` и `b`, то они принадлежат одному классу. Если в другом операторе ввода (в этом же модуле) есть операции `a` и `c`, то операция `c` также принадлежит классу, содержащему `a` и `b`. В худшем случае все операции модуля принадлежат одному классу. В лучшем — каждая находится в своем классе (например, если все они обслуживаются операторами `receive`).

Ссылка на операцию, которую обслуживают операторы ввода, содержит указатель на дескриптор операции. Дескриптор операции, в свою очередь, содержит указатель на дескриптор класса операции. Оба дескриптора хранятся на машине, обслуживающей эту операцию. Дескриптор класса содержит следующую информацию:

блокировка — используется для взаимоисключающего доступа;

список задержанных — задержанные вызовы операций данного класса;

список новых — вызовы, пришедшие, пока класс был заблокирован;

список доступа — процессы, ожидающие блокировки;

список ожидания — процессы, ожидающие появления новых вызовов.

Блокировка используется для того, чтобы в любой момент времени только один процесс мог просматривать ожидающие вызовы. Использование других полей описано ниже.

Операторы вызова и передачи реализуются следующим образом. Если операцию обслуживает процедура, которая находится на той же машине, то оператор вызова преобразуется в прямой вызов процедуры. Процесс может определить это, просмотрев поля описанной выше ссылки на операцию. Если операция относится к другой машине или обслуживается опе-

раторами ввода, то оператор вызова запускает на локальной машине примитив invoke. Независимо от того, как обслуживается операция, оператор send выполняет примитив invoke.

Листинг 10.9 содержит код примитива invoke и двух процедур ядра, которые он использует. Первый аргумент определяет вид вызова. При вызове CALL ядро блокирует выполняемый процесс до завершения обслуживания вызова. Затем ядро определяет, на локальной или удаленной машине должна быть обслужена операция. Если на удаленной, ядро отсылает сообщение INVOKE удаленному ядру, выполняющему затем процедуру ядра localInvoke.

Подпрограмма localInvoke проверяет, процедурой ли обслуживается операция. Если да, то она получает свободный дескриптор процесса и дает серверному процессу указание выполнить процедуру, как в ядре для RPC. Ядро также записывает в дескрипторе, как вызывалась операция. Эти данные используются позже для определения, есть ли вызвавший процедуру процесс, который нужно запустить после выхода из нее.

Листинг 10.9. Примитивы вызова

```
type howInvoked = enum(CALL, SEND);
type howServiced = enum(PROC, IN);
type opRef = rec(howServiced how; int machine, opid);

proc invoke(howInvoked how; opRef op; byte values[*]) {
  if (how == CALL)
    вставить executing в список задержанных вызовов call;
  if (op.machine локальная)
    localInvoke(executing, how, op, values);
  else {    # удаленная машина
    netWrite(machine, INVOKE, (executing,how,op,values));
    dispatcher();
  }
}

proc localInvoke(int caller; howInvoked inv;
                 opRef op; byte values[*]) {
  if (op.how == PROC) {
    получить свободный дескриптор процесса;
    if (inv == CALL)
      сохранить идентификатор процесса caller в дескрипторе;
    else    # inv == SEND
      записать, что вызывающего процесса нет (обнулить поле вызывающего процесса);
    присвоить счетчику команд процесса значение op.address;
    поместить values в стек процесса;
    вставить дескриптор процесса в список готовых к работе;
  }
  else {    # op.how == IN
    просмотреть дескриптор класса операции;
    if (inv == CALL)
      append(opclass, caller, op.opid, values);
    else    # inv == SEND
      append(opclass, 0, op.opid, values);
  }
  dispatcher();
}

proc append(int opclass, caller, opid; byte values[*]) {
  if (opclass заблокирован) {
```

```
    вставить (caller, opid, values) в список новых вызовов;
    переместить процессы (если есть) из списка ожидания в список доступа;
  }
  else {   # opclass не заблокирован
    вставить (caller, opid, values) в список задержанных;
    if (список ожидания не пуст) {
      переместить первый процесс в список готовых к работе;
      переместить остальные процессы в список доступа;
      установить блокировку;
    }
  }
}
```

Если операция обслуживается оператором ввода, процедура `localInvoke` проверяет дескриптор класса. Если класс заблокирован (процесс выполняет оператор ввода и проверяет задержанные вызовы), ядро сохраняет вызов в списке новых вызовов и перемещает все процессы, ожидающие новых вызовов, в список доступа к классу. Если класс не заблокирован, ядро добавляет вызов ко множеству задержанных вызовов и проверяет список ожидания. (Список доступа пуст, если класс не заблокирован.) Если есть процессы, ожидающие новых вызовов, то один из них запускается, устанавливается блокировка, а остальные ожидающие процессы перемещаются в список доступа.

Заканчивая выполнение процедуры, процесс вызывает примитив ядра `procDone` (листинг 10.10). Этот примитив освобождает дескриптор процесса и запускает вызвавший процедуру процесс (если такой есть). Примитив `awakenCaller` выполняется ядром на той машине, на которой размещен вызывающий процесс.

Листинг 10.10. Примитивы возвращения

```
proc procDone(byte results[*]) {
  вернуть executing в список свободных дескрипторов;
  найти идентификатор процесса caller (если есть) в дескрипторе процесса;
  if (caller локальный)
    awakenCaller(caller, results);
  else if (caller удаленный)
    netWrite(машина вызывающего процесса, RETURN, (caller, results));
  dispatcher();
}

proc awakenCaller(int caller; byte results[*]) {
  удалить дескриптор процесса caller из списка задержанных;
  поместить results в стек процесса, вызвавшего процедуру;
  вставить дескриптор процесса caller в список готовых к работе;
}
```

Для операторов ввода процесс выполняет код, приведенный в листинге 10.11. В этом коде вызываются примитивы оператора ввода (листинг 10.12). Процесс сначала получает исключительный доступ к дескриптору класса операции и затем ищет приемлемые задержанные вызовы. Если ни один из них принять нельзя, процесс вызывает процедуру `waitNew`, чтобы приостановить работу до прихода нового вызова. Этот примитив может закончиться немедленно, если новый вызов приходит во время поиска задержанных вызовов (и, следовательно, при заблокированном дескрипторе класса).

Найдя приемлемый вызов, процесс выполняет соответствующую защищенную операцию и вызывает процедуру `inDone`. Ядро запускает вызвавший процедуру процесс (если он есть)

и обновляет состояние дескриптора класса. Если во время выполнения оператора ввода прибывают новые вызовы, они перемещаются в список задержанных, а процессы, ожидающие доступа к классу, перемещаются в список доступа. Затем, если есть процессы, ожидающие доступ к классу, запускается один из них, иначе блокировка снимается.

Листинг 10.11. Код оператора ввода, выполняемый процессом

```
startIn(opclass);
while (true) {
  # повторять, пока не будет найден приемлемый вызов
  искать приемлемый вызов в списке задержанных вызовов для класса opclass;
  if (вызов найден)
    break;  # выход из цикла
  waitNew(opclass);
}
переместить вызов из списка задержанных для класса opclass;
выполнить соответствующую защищенную операцию;
inDone(opclass, caller, результирующие значения);
```

Листинг 10.12. Примитивы оператора ввода

```
proc startIn(int opclass) {
  if (opclass заблокирован)
    вставить процесс executing в список доступа класса opclass;
  else
    установить блокировку;
  dispatcher();
}

proc waitNew(int opclass) {
  искать дескриптор класса операций opclass;
  if (список новых вызовов не пуст)
    переместить новые вызовы в список задержанных;
  else {
  вставить процесс executing в список ожидающих процессов класса opclass;
    if (список доступа пуст)
      снять блокировку;
    else
      переместить первый процесс из списка доступа в список готовых к работе;
  }
  dispatcher();
}

proc inDone(int opclass, caller; byte results[*]) {
  if (caller локальный)
    call awakenCaller(caller, results);
  else if (caller удаленный)
    netWrite(машина вызывающего процесса, RETURN, (caller,results));
  if (список новых вызовов класса opclass не пуст) {
    переместить вызовы из списка новых в список задержанных;
    переместить процессы (если есть) из списка ожидающих в список доступа;
  }
  if (список доступа класса opclass не пуст)
```

```
    переместить первый дескриптор в список готовых к работе;
  else
    снять блокировку;
  dispatcher();
}
```

Эта реализация операторов ввода пригодна в самом общем случае. Ее можно оптимизировать для обработки следующих распространенных частных случаев.

- Если все операции класса обслуживаются одним процессом, то список доступа к классу не нужен, поскольку серверу никогда не придется ожидать получения блокировки. (Сама блокировка, однако, нужна, чтобы процедура append могла определить, в какой список, новых или задержанных вызовов, вставлять новый вызов.)
- Если в классе только одна операция, и она обслуживается операторами receive или операторами ввода без выражений планирования и условий синхронизации, то ожидающие вызовы обслуживаются в порядке FIFO, как при передаче сообщений. Для обработки такой ситуации можно добавить примитив ядра, чтобы приостановить работу сервера, пока не появится задержанный вызов, и затем возвратить его серверу. Серверу не нужно блокировать класс и искать в списке задержанных вызовов.
- Если операция действует как семафор (не имеет параметров или возвращаемого значения), вызывается оператором send и обслуживается оператором receive, ее можно реализовать с помощью семафора.

Описанные оптимизации значительно повышают производительность программы.

10.4. Распределенная разделяемая память

Как отмечалось во введении к данной главе, в программах для машин с разделяемой памятью обычно используются разделяемые переменные, а для машин с распределенной памятью — передача сообщений, RPC или рандеву. Но, как показано в разделе 10.1, на машинах с разделяемой памятью легко реализовать передачу сообщений. На машинах с распределенной памятью также можно, хотя и сложнее, реализовать разделяемые переменные. Здесь описано, как это сделать. В исторической справке и списке литературы есть ссылки на более подробные источники информации по этой теме.

Напомним, что в мультипроцессоре с разделяемой памятью каждый процессор имеет доступ к любой области памяти (см. раздел 1.2). Поскольку время доступа к памяти намного больше, чем длительность цикла процессоров, для повышения скорости работы мультипроцессоры используют кэширование. В каждом процессоре есть один или несколько уровней кэш-памяти. В ней хранятся копии областей памяти, к которым чаще всего обращается код программы, выполняемой на этом процессоре. Кэш-память организована в виде набора строк памяти, каждая из которых содержит одно или несколько непрерывных слов памяти. Для поддержания согласованности данных в кэш-памяти и в первичной памяти используются аппаратно реализованные протоколы.

Распределенная разделяемая память (РРП) является программной реализацией этих же идей на машине с распределенной памятью. РРП обеспечивает виртуальное адресное пространство, доступное каждому процессору. Это адресное пространство обычно организовано в виде набора страниц, распределенных в локальной памяти процессоров. Каждая страница может храниться в одной копии или в нескольких копиях на разных процессорах. Обращаясь к памяти удаленной машины, процесс должен получить копию страницы. Для управления перемещением страниц и их содержимого используется *протокол согласования страниц.*

Основная причина для обеспечения РРП состоит в том, что большинству программистов легче писать параллельные программы, используя разделяемые переменные, а не передачу

сообщений. Частично это связано с тем, что понятие разделяемых переменных близко последовательному программированию. Однако РРП требует дополнительных затрат на обработку ситуаций, связанных с отсутствием страниц в локальной памяти, а также на получение и передачу удаленных страниц.

Ниже описана реализация РРП, которая сама по себе является еще одним примером распределенной программы. Затем рассматриваются некоторые из общепринятых протоколов согласования страниц и то, как они поддерживают схемы доступа к данным в прикладных программах.

10.4.1. Обзор реализации

РРП — это программная прослойка между прикладными программами и операционной системой или специализированным ядром. Ее общая структура показана на рис. 10.3. Адресное пространство каждого процессора (узла) состоит из разделяемой и скрытой областей. Разделяемые переменные прикладной программы хранятся в разделяемой области, а код и скрытые данные — в скрытой. Таким образом, можно считать, что в разделяемой области памяти находится "куча" программы, а в скрытой — сегменты кода и стеки процессов. В скрытой области каждого узла также расположена копия программного обеспечения РРП и операционная система узла или ядро взаимодействия.

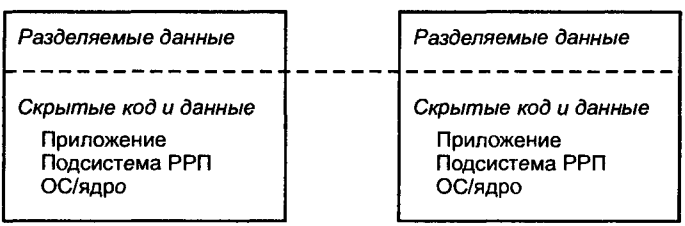


Рис. 10.3. Структура системы распределенной разделяемой памяти

РРП управляет разделяемой областью адресного пространства. Это линейный массив байтов, который теоретически дублируется на каждом узле. Разделяемая область памяти разбивается на модули, каждый из которых защищается отдельно и находится на одном или нескольких узлах. Обычно модули являются страницами фиксированного размера, хотя они могут быть и страницами переменного размера или отдельными объектами данных. Здесь предполагается, что размеры модулей зафиксированы. Управление страницами аналогично управлению страничной виртуальной памятью одиночного процессора. Страница может быть или резидентной (присутствующей) или нет. Резидентная страница может быть доступной только для чтения или для чтения и записи.

Каждая разделяемая переменная в приложении отображается в адрес в разделяемой области памяти и, следовательно, имеет один и тот же адрес на всех узлах. Вначале страницы разделяемой области некоторым образом распределены по узлам. Пока будем считать, что существует одна копия каждой страницы, и страница доступна для чтения и записи на узле ее расположения.

Обращаясь к разделяемой переменной в резидентной странице, процесс получает к ней прямой доступ. Но обращение к нерезидентной странице приводит к ошибке обращения к странице. Эту ошибку обрабатывает программное обеспечение РРП. Оно определяет расположение страницы и отправляет сообщение на запросивший ее (первый) узел. Второй узел (на котором была страница) помечает ее как нерезидентную и передает на первый узел. Первый узел, получив страницу, обновляет ее защиту и возвращается к прикладному процессу. Как и в системе виртуальной памяти, прикладной процесс повторно выполняет инструкцию, вызвавшую ошибку обращения к странице, и обращение к разделяемой переменной заканчивается успешно.

Для иллюстрации описанных выше действий рассмотрим следующий простой пример.

```
int x = 0, y = 0;
process P1 {
  x = 1;
}
process P2 {
  y = 1;
}
```

Даны два узла. На первом узле выполняется процесс P1, на втором — P2. Предположим, что разделяемые переменные хранятся в одной странице памяти, которая вначале находится на первом узле. Последовательность действий показана на рис. 10.4.

	Узел 1	Узел 2
начальное состояние	x = 0, y = 0	
1	запись x, ошибки нет	
2		запись y, ошибка обращения к странице
3		отправка запроса на узел 1
4	передача страницы на узел 2	
5		получение страницы
6		запись y, ошибки нет
конечное состояние		x = 1, y = 1

Рис. 10.4. Обработка ошибки обращения к странице в РРП

Процессы могли бы работать синхронно или в произвольном порядке, но предположим, что P1 выполняется первым и присваивает x. Поскольку страница с x в данный момент находится на узле 1, запись выполняется успешно и процесс завершается. Теперь начинается P2. Он пытается записать в y, но содержащая y страница нерезидентная, и возникает ошибка. Обработчик ошибки узла 2 передает запрос страницы на узел 1, который отсылает страницу узлу 2. Получив страницу, узел 2 запускает процесс P2; теперь запись в y завершается успешно. В заключительном состоянии страница является резидентной для узла 2 и обе переменные имеют новые значения.

Когда РРП реализуется на основе операционной системы Unix, защита разделяемых страниц устанавливается системным вызовом mprotect. Защита резидентной страницы устанавливается в состояние READ или READ и WRITE. Защита нерезидентных страниц устанавливается в состояние NONE. При запросе нерезидентной страницы вырабатывается сигнал о нарушении сегментации (SIGSEGV). Обработчик ошибок обращения к страницам в РРП получает этот сигнал и посылает сообщение о запросе страницы, используя примитивы взаимодействия Unix (или им подобные программы). По прибытии сообщения на узел генерируется сигнал ввода-вывода (SIGIO). Обработчик сигналов ввода-вывода определяет тип сообщения (запрос страницы или ответ) и предпринимает соответствующие действия. Обработчики сигналов должны выполняться в критических секциях, поскольку во время обработки одного сигнала может прийти другой, например во время обработки ошибки обращения к странице может прийти запрос на страницу.

РРП может быть однопоточной или многопоточной. Однопоточная РРП поддерживает только один прикладной процесс на каждом узле. Процесс, вызвавший ошибку обращения к странице, приостанавливается до разрешения ошибки. Многопоточная РРП поддерживает несколько приложений на каждом узле, поэтому, когда один процесс вызывает ошибку обращения к странице, другой может выполняться, пока первый ожидает разрешения ошибки. Однопоточную РРП реализовать проще, поскольку в ней меньше критических секций. Однако многопоточная РРП гораздо лучше скрывает задержки при обращениях к удаленной странице и, следовательно, может обеспечить более высокую производительность.

10.4.2. Протоколы согласования страниц

Производительность приложения, выполняемого на базе РРП, зависит от эффективности реализации самой РРП. Сюда входят и возможность маскирования ожидания при доступе к памяти, и эффективность обработчиков сигналов, и, в особенности, свойства протоколов взаимодействия. Производительность приложения также существенно зависит от методов управления страницами, а именно — от используемого *протокола согласования страниц*. Далее описаны три таких протокола: блуждания, или переноса (migratory), денонсирующей записи (write invalidate) и совместной записи (write shared).

В примере в листинге 10.4 предполагалось, что существует одна копия каждой страницы. Когда страница нужна другому узлу, копия перемещается. Это называется *протоколом переноса*. Содержание страницы всегда согласованно, поскольку есть только одна ее копия. Но что случится, когда два процесса на двух узлах просто захотят прочесть переменную на этой странице? Тогда страница будет прыгать между ними; этот процесс называется *замусориванием* (trashing).

Протокол денонсирующей записи позволяет дублировать страницы при чтении. У каждой страницы есть владелец. Пытаясь прочитать удаленную страницу, процесс получает от ее владельца неизменяемую копию (только для чтения). Копия владельца на это время тоже помечается как доступная только для чтения. Пытаясь записать в страницу, процесс получает копию (при необходимости), делает недействительными (денонсирует) остальные копии и записывает в полученную страницу. Обработчик ошибки обращения к странице узла, выполняющего запись, при этом совершает следующие действия: 1) связывается с владельцем, 2) получает страницу и владение ею, 3) отправляет денонсирующие сообщения узлам, на которых есть копии этой страницы (они устанавливают защиту своей копии страницы в состояние `NONE`), 4) устанавливает защиту своей копии в состояние `READ` и `WRITE`, 5) возобновляет прикладной процесс.

Протокол денонсирующей записи очень эффективен для страниц, которые доступны только для чтения (после их инициализации) или редко изменяются. Однако при появлении *ложного разделения* этот протокол приводит к замусориванию. Вернемся к рис. 10.4. Переменные `x` и `y` не разделяются процессами, но находятся на одной разделяемой странице. Следовательно, эта страница будет перемещаться между двумя узлами так же, как и при использовании протокола переноса.

Ложное разделение можно исключить, размещая такие переменные, как `x` и `y` в примере на разных страницах. Это можно делать статически во время компиляции программы или динамически во время выполнения. Вместе с тем, ложное разделение можно допустить, используя *протокол совместной записи*. Этот процесс позволяет выполнять несколько параллельных записей на одну страницу. Например, когда процесс `P2`, выполняемый на узле 2 (см. рис. 10.4), записывает в `y`, узел 2 получает копию страницы от узла 1, и оба узла получают разрешение записать на эту страницу. Ясно, что копии становятся несогласованными. В таких точках синхронизации, определенных в приложении, как барьеры, копии объединяются. При ложном разделении страницы объединенная копия будет правильной и согласованной. Однако при действительном разделении объединенная копия будет неопределенной комбинацией записанных значений. Для обработки таких ситуаций каждый узел поддерживает список изменений, сделанных им на странице с совместной записью. Эти списки используются при объединении копий для учета последовательности записей, сделанных разными процессами.

Историческая справка

Реализации примитивов взаимодействия существовали столько же, сколько и сами примитивы. В исторических справках к главам 7–9 упоминались статьи с описаниями новых примитивов; во многих из этих работ представлена и реализация примитивов. Два хороших

источника информации по данной теме — книги [Bacon, 1998] и [Tanenbaum, 1992]. В них описаны примитивы взаимодействия и вопросы их реализации в целом, рассмотрены примеры важных операционных систем.

В распределенном ядре (раздел 10.1) предполагалось, что передача по сети происходит без ошибок, и не учитывалось управление буферами и потоками. Решение этих вопросов представлено в книгах по компьютерным сетям, например [Tanenbaum, 1988] или [Paterson and Davie, 1996].

Централизованная реализация синхронной передачи сообщений (см. рис. 10.2, листинги 10.5 и 10.6) была разработана автором этой книги. Существуют и децентрализованные решения без центрального управляющего. В работе [Silberschatz, 1979] рассмотрены процессы, образующие кольцо, а в [Van de Snepscheut, 1981] — иерархические системы. В [Bernstein, 1980] представлена реализация, работающая при любой топологии связи, а в [Schneider, 1982] — алгоритм рассылки, который, по существу, повторяет множества ожидания нашего учетного процесса (листинг 10.6). Алгоритм Шнейдера прост и справедлив, но требует обмена большим количеством сообщений, поскольку каждый процесс должен подтверждать каждую передачу. В статье [Buckley and Silberschatz, 1983] предложен справедливый децентрализованный алгоритм, обобщающий алгоритм из [Bernstein, 1980]. Этот алгоритм эффективнее, чем алгоритм Шнейдера, но намного сложнее. (Все упомянутые алгоритмы описаны в книге [Raynal, 1988].) В работе [Bagrodia, 1989] представлен еще один алгоритм, более простой и эффективный, чем алгоритм из [Buckley and Silberschatz, 1983].

В разделе 10.3 были представлены реализации рандеву с использованием асинхронной передачи сообщений и ядра. Из-за сложности механизма рандеву его производительность в целом ниже, чем других механизмов синхронизации. Во многих программах рандеву можно заменить более простыми механизмами, такими как процедуры и семафоры. Многочисленные примеры таких преобразований (для языка Ада) приведены в статье [Roberts et al., 1981]. В [McNamee and Olsson, 1990] представлено больше преобразований и проведен анализ получаемого роста производительности, который в некоторых случаях достигал 95 процентов.

Концепцию распределенной разделяемой памяти (РРП) разработал Кай Ли (Kai Li) в свой докторской диссертации в Йельском университете под руководством Пола Хьюдака (Paul Hudak). В статье [Li and Hudak, 1989] представлен протокол денонсирующей записи для согласования страниц. Ли внес решающий вклад в развитие этой темы, поскольку до его публикации никто не верил, что с помощью передачи сообщений можно моделировать разделяемую память с приемлемой производительностью. Теперь РРП занимает прочные позиции и даже поддерживается многими производителями суперкомпьютеров.

Две важнейшие из последних РРП-систем — Munin и TreadMarks, разработанные в университете Райса (Rice). Реализация и производительность системы Munin, в которой появился протокол совместной записи, описана в статье [Carter, Bennett and Zwaenepoel, 1991], а система TreadMarks — в [Amza et al., 1996]. Производительность PVM и TreadMarks сравнивается в работе [Lu et al., 1997]. Книга [Tanenbaum, 1995] содержит отличный обзор по РРП, включая работу Ли, систему Munin и другие. За более новой информацией по вопросам РРП обращайтесь к специализированному изданию *Proceedings of IEEE* (март 1999).

Литература

Amza, C., A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. 1996. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer* 29, 2 (February): 18–28.

Bacon, J. 1998. *Concurrent Systems: Operating Systems, Database and Distributed Systems: An Integrated Approach*, 2nd ed. Reading, MA: Addison-Wesley.

Bagrodia, R. 1989. Synchronization of asynchronous processes in CSP. *ACM Trans. on Prog. Languages and Systems* 11, 4 (October): 585–597.

Bernstein, A. J. 1980. Output guards and non-determinism in CSP. *ACM Trans. on Prog. Languages and Systems* 2, 2 (April): 234–238.

Buckley, G. N., and A. Silberschatz. 1983. An effective implementation for the generalized input-output construct of CSP. *ACM Trans. on Prog. Languages and Systems* 5, 2 (April): 223–235.

Carter, J. B., J. K. Bennett, and W. Zwaenepoel. 1991. Implementation and performance of Munin. *Proc. 13th ACM Symposium on Operating Systems Principles*, October, pp. 152–164.

Li, K., and P. Hudak. 1989. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems* 7, 4 (November): 321–359.

Lu, H., S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. 1997. Quantifying the performance differences between PVM and TreadMarks. *Journal of Par. and Distr. Computation* 43, 2 (June): 65–78.

McNamee, C. M., and R. A. Olsson. 1990. Transformations for optimizing interprocess communication and synchronization mechanisms. *Int. Journal of Parallel Programming* 19, 5 (October): 357–387.

Peterson, L. L., and B. S. Davie. 1996. *Computer Networks: A Systems Approach*. San Francisco: Morgan Kaufmann.

Raynal, M. 1988. *Distributed Algorithms and Protocols*. New York: Wiley.

Roberts, E. S., A. Evans, C. R. Morgan, and E. M. Clarke. 1981. Task management in Ada — a critical evaluation for real-time multiprocessors. *Software — Practice and Experience* 11: 1019–1051.

Schneider, F. B. 1982. Synchronization in distributed programs. *ACM Trans. on Prog. Languages and Systems*, 2 (April): 125–148.

Silberschatz, A. 1979. Communication and synchronization in distributed programs. *IEEE Trans. on Software Engr.* SE–5, 6 (November): 542–546.

Tanenbaum, A. S. 1988. *Computer Networks*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall.

Tanenbaum, A. S. 1992. *Modern Operating Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Tanenbaum, A. S. 1995. *Distributed Operating Systems*. Englewood Cliffs, NJ: Prentice-Hall.

van de Snepscheut, J. L. A. 1981. Synchronous communication between asynchronous components. *Inf. Proc. Letters* 13, 3 (December): 127–130.

Упражнения

10.1. Рассмотрим распределенное ядро в листингах 10.2 и 10.3:

а) расширьте реализацию, чтобы у канала могло быть несколько получателей. В частности, измените примитивы `receiveChan` и `emptyChan`, чтобы процесс на одной машине мог получать доступ к каналу, расположенному на другой;

б) измените ядро, чтобы примитив `sendChan` стал *полусинхронным*, т.е., вызывая `sendChan`, процесс должен дождаться постановки сообщения в очередь канала (или передачи получателю), даже если канал расположен на другой машине;

в) добавьте в ядро код определения окончания программы. Можно не учитывать ожидающий ввод-вывод. Вычисления завершены, когда все списки готовых к работе процессов пусты и сеть бездействует.

10.2. Реализация синхронной передачи сообщений в листинге 10.4 предполагает, что и процесс-источник, и процесс-приемник именуют друг друга. Обычно же процесс-приемник должен либо указывать источник, либо принимать сообщения от любого ис-

точника. Предположим, что процессы пронумерованы от 1 до n и индекс 0 используется процессом-приемником для указания, что он готов принять сообщение от любого источника. В последнем случае оператор ввода присваивает параметру source идентификатор процесса, приславшего сообщение.

Измените протоколы взаимодействия в листинге 10.4, чтобы они обрабатывали описанную ситуацию. Как и в листинге, основой для реализации описанной выше формы синхронной передачи сообщений должна служить асинхронная передача сообщений.

10.3. Разработайте реализацию в ядре примитивов синхронной передачи сообщений synch_send и synch_receive, определенных в начале раздела 10.2. Сначала постройте однопроцессорное ядро. Затем разработайте распределенное ядро со структурой, показанной в листинге 10.1. Все необходимые процедуры можно брать из листингов 10.2 и 10.3.

10.4. Даны процессы P[1:n], каждый из которых содержит значение элемента a[i] массива из n значений. В следующей программе используется синхронная передача сообщений. Она написана в нотации CSP (см. раздел 7.6). В программе каждый процесс отправляет свое значение элемента массива всем остальным процессам. Когда программа завершается, у каждого процесса есть значения всех элементов массива.

```
process P[i = 1 to n] {
  int a[1:n];   # a[i] считается инициализированным
  bool sent[1:n] = ([n] false);
  int recvd = 0;
  do [j = 1 to n] (i != j and not sent[j]);
        P[j]!a[i] -> sent[j] = true;
  [] [j = 1 to n] (i != j and recvd < n-1);
        P[j]?a[j] -> recvd = recvd+1;
  od
}
```

Квантификаторы в ветвях оператора do показывают, что есть n копий каждой ветви, по одной для каждого значения счетчика j:

а) проведите трассировку одной из возможных последовательностей сообщений, отправляемых при реализации этой программы с централизованным учетным процессом (см. листинг 10.6). Пусть n = 3. Сколько сообщений отправят процессы и сколько — учетный процесс? Отследите содержимое множества шаблонов, получаемых учетным процессом;

б) каково *общее* число сообщений, отправленных *и* полученных учетным процессом в вашем ответе к пункту *а*? Выразите это количество в виде функции, зависящей от n.

10.5. Алгоритм в листинге 9.11 дает справедливую децентрализованную реализацию семафоров. В нем использованы примитивы broadcast, метки времени и вполне упорядоченные очереди сообщений:

а) используя подобный алгоритм, разработайте справедливую децентрализованную реализацию синхронной передачи сообщений. Считайте, что операторы ввода и вывода могут появляться в защите. *Указание.* Обобщите централизованную реализацию из раздела 10.2, продублировав ожидающее множество шаблонов учетного процесса;

б) проиллюстрируйте выполнение вашего алгоритма на примере программы с процессами A и B в конце раздела 10.2.

10.6. Ядро в листингах 10.9–10.12 реализует нотацию совместно используемых примитивов (см. раздел 8.3):

а) предположим, что язык программирования имеет только механизмы рандеву, определенные в разделе 8.2. Операции вызываются только оператором `call`, а обслуживаются только операторами `in`. Каждую операцию может обслуживать только тот процесс, в котором она объявлена. Сделайте ядро как можно более простым, чтобы оно реализовывало только этот набор механизмов;

б) в языке Ada аналог оператора `in` дополнительно ограничен: условия синхронизации не могут ссылаться на формальные параметры, а выражений планирования нет. Измените ответ к пункту *а*, чтобы реализовать эту ограниченную форму оператора `in`.

10.7. Рассмотрим примитивы Linda (см. раздел 7.7):

а) реализуйте эти примитивы, изменив централизованный учетный процесс из листинга 10.6. Укажите действия, которые должны выполнить обычные процессы для каждого из примитивов Linda (как в листинге 10.5);

б) разработайте реализацию примитивов Linda в распределенном ядре. По уровню детализации решение должно быть сравнимо с листингами 10.2 и 10.3.

10.8. На рис. 10.4 представлены действия РРП, использующей для согласования страниц протокол переноса. В этом примере было два процесса. Первый выполняется на узле `1` и записывает в переменную `x`, второй — на узле `2` и записывает в переменную `y`. Обе переменные хранятся на одной странице, которая вначале расположена на узле `1`:

а) представьте трассу действий РРП, использующей протокол денонсирующей записи. Считайте, что первым выполняется процесс, записывающий в `x`;

б) повторите пункт *а* при условии, что первым выполняется процесс, который записывает в `y`;

в) представьте трассу действий РРП, использующей протокол совместной записи (первым выполняется процесс, записывающий в `x`);

г) повторите пункт *в* при условии, что первым выполняется процесс, который записывает в `y`;

д) повторите пункт *в*, предполагая, что процессы выполняются одновременно;

е) при использовании протокола совместной записи страница становится несогласованной, если в нее записывают несколько процессов. Предположим, что страницы согласуются в точках барьерной синхронизации. Дополните ответ к пункту *д*, включив барьер после двух операций записи. Добавьте к трассе действия узлов по согласованию двух копий страницы.

Часть 3

Синхронное параллельное программирование

Термин *параллельная программа* относится к любой программе, имеющей несколько процессов. В части 1 говорилось, как писать параллельные программы, в которых процессы взаимодействуют с помощью разделяемых переменных и синхронизируются, используя активное ожидание, семафоры или мониторы. В части 2 речь шла о параллельных программах, в которых процессы взаимодействуют и синхронизируются с помощью обмена сообщениями, RPC или рандеву. Такие программы называются *распределенными*, поскольку процессы в них обычно распределяются между процессорами, не разделяющими память.

В части 3 представлен третий тип параллельных программ — *синхронные параллельные программы*. Их отличительное свойство определяется целью их создания — решать поставленную задачу быстрее, чем с помощью аналогичных последовательных программ, сокращая время работы. Параллельные[20] программы призваны, во-первых решать экземпляры задачи большего размера, а, во-вторых, большее их количество за одно и то же время. Например, программа для прогнозирования погоды должна заканчиваться за определенное время и в идеале давать как можно более точные результаты. Для повышения точности вычислений можно использовать более подробную модель погоды. Можно также данную модель запускать несколько раз с различными начальными условиями. В обоих случаях параллельная программа повысит шансы получить результаты вовремя.

Параллельную программу можно написать, используя или разделяемые переменные, или обмен сообщениями. Обычно выбор диктуется типом архитектуры вычислителя, на котором будет выполняться программа. В параллельной программе, выполняемой на мультипроцессоре с разделяемой памятью, обычно используются разделяемые переменные, а в программе для мультикомпьютера или сети машин — обмен сообщениями.

Мы уже приводили несколько примеров синхронных параллельных программ: умножение матриц, адаптивные квадратуры, алгоритмы, параллельные по данным, порождение простых чисел и обработка изображений. (В упражнениях в конце каждой главы описаны многие другие задачи.) Рассмотрим важную тему высокопроизводительных вычислений, связанную с применением параллельных машин к решению задач большого размера в науке и инженерии.

В главе 11 рассмотрены три основных класса научных приложений: 1) сеточные вычисления для приближенных решений дифференциальных уравнений в частных производных, 2) точечные вычисления для моделирования систем взаимодействующих тел и 3) матричные вычисления для решения систем линейных уравнений. Для каждого класса приведена типич-

[20] В дальнейшем, если речь не идет о других видах параллелизма, слово "синхронные" иногда опускается. — *Прим. ред.*

ная задача и разработаны параллельные программы, в которых используются разделяемые переменные или обмен сообщениями.

В программах использованы языковые механизмы и методы программирования, описанные в частях 1 и 2. В главе 12 рассмотрены дополнительные языки, компиляторы, библиотеки и инструментарий для высокопроизводительных параллельных вычислений. Отдельные темы посвящены библиотеке OpenMP для параллелизма с разделенной памятью, распараллеливающим компиляторам, параллельным по данным, и функциональным языкам, абстрактным моделям, языку программирования High-Performance Fortran (HPF) и инструментальной системе Глобус для научных метавычислений.

Прежде чем рассматривать отдельные приложения, определим несколько ключевых понятий, присущих параллельному программированию: ускорение, эффективность, источники накладных расходов и проблемы, которые нужно преодолевать, чтобы достичь хорошей производительности.

Ускорение и эффективность

Как уже отмечалось, главная цель параллельного программирования — решить задачу быстрее. Уточним эту цель. *Производительность* программы определяется общим временем ее выполнения (работы). Пусть для решения некоторой задачи с помощью последовательной программы, выполняемой на одном процессоре, нужно время T_1, а с помощью параллельной программы, выполняемой на p процессорах, — T_p. Тогда *ускорение* (*speedup* — S) параллельной программы определяется как $S = T_1/T_p$.

Например, пусть время работы последовательной программы равно 600 *с*, а время выполнения параллельной программы на 10 процессорах — 60 *с*. Тогда ускорение параллельной программы равно 10. Такое ускорение называется *линейным* (или идеальным), поскольку программа на 10 процессорах работает в 10 раз быстрее. Обычно ускорение программы, выполняемой на p процессорах, оказывается меньше p. Такое ускорение называется *менее, чем линейным* (sublinear). Иногда ускорение оказывается *более, чем линейным* (superlinear), т.е. больше p; так бывает, когда данных в программе так много, что они не умещаются в кэш одного процессора, но после разделения их можно разместить в кэшах p процессоров.

Двойником ускорения является *эффективность* (*Efficiency* — E) — мера того, насколько хорошо параллельная программа использует дополнительные процессоры. Она определяется следующим образом.

$$E = S/p = T_1/(p * T_p)$$

Если программа имеет линейное ускорение, ее эффективность равна 1,0. Эффективность меньше 1,0 означает, что ускорение менее, чем линейное, а больше — что ускорение более, чем линейное.

Ускорение и эффективность относительны. Они зависят от количества процессоров, размера задачи и используемого алгоритма. Например, часто эффективность параллельной программы с ростом числа процессоров снижается; например, при малом числе процессоров эффективность может быть близка к 1,0 и уменьшаться с ростом p. Аналогично параллельная программа может быть весьма эффективной при решении задач больших (но не малых) размеров. Говорят, что параллельная программа *масштабируема*, если ее эффективность постоянна в широком диапазоне количеств процессоров и размеров задач. Наконец, ускорение и эффективность зависят от используемого алгоритма — параллельная программа может оказаться эффективной для одного последовательного алгоритма и неэффективной для другого. Поэтому лучшей мерой будет *абсолютная эффективность* (или абсолютное ускорение), для которой T_1 — время работы программы по наилучшему из известных последовательных алгоритмов.

Ускорение и эффективность зависят от общего времени выполнения. Обычно программа имеет три фазы — ввод данных, вычисления, вывод данных. Предположим, что в последовательной программе фазы ввода и вывода данных занимают по 10% от времени выполнения, а фаза вычислений — остальные 80%. Предположим также, что фазам ввода и вывода прису-

ще последовательное выполнение, т.е. в параллельной программе их нельзя ускорить. Тогда максиальное ускорение, достижимое с помощью параллельной программы, *не более 5*! Например, если фазы ввода и вывода занимают по 10 *с* (а фаза вычислений — 80 *с*), то минимальное время работы *любой* параллельной программы будет больше 20 *с*, даже если длительность фазы вычислений сократить почти до 0 *с*. Таким образом, ускорение параллельной программы ограничено сверху 100/20, т.е. пятью.

Указанный предел ускорения следует из *закона Амдала* (его автора): максимальное улучшение, достигаемое при ускорении части вычислений, ограничено долей времени, занимаемого этой частью. Уточним. Пусть T_{old} обозначает общее время работы первоначальной программы, T_{new} — модифицированной программы, S_{tot} — общее ускорение. Тогда $S_{tot} = T_{old}/T_{new}$. Пусть F_{part} обозначает долю времени работы, которую в общем времени T_{old} занимает ускоряемая часть, а S_{part} — ее ускорение. Тогда

$$T_{new} = T_{old} - F_{part} \times T_{old} + F_{part} \times T_{old} / S_{part}.$$

Отсюда

$$S_{tot} = 1/(1 - F_{part} + F_{part}/S_{part}).$$

В нашем примере доля фазы вычислений, которую можно распараллелить, равна 80 %, или 0,8. Следовательно, если ускорение этой фазы бесконечно, то общее ускорение все равно останется равным 1/(1–0,8), т.е. 5. Однако общее ускорение на p процессорах в действительности меньше. Например, если ускорение вычислительной фазы на 10 процессорах равно 10, то общее ускорение — 1/(0,2+0,8/10), или приблизительно 3,57.

К счастью, во многих приложения фазы ввода и вывода данных занимают пренебрежимо малую часть общего времени выполнения. Более того, быстродействующие машины всегда обеспечивают аппаратную поддержку высокоскоростного параллельного ввода-вывода. Таким образом, общая производительность параллельных вычислений будет в целом определяться тем, насколько хорошо распараллеливается фаза вычислений.

Накладные расходы и дополнительные проблемы

Предположим, что дана последовательная программа решения некоторой задачи, и нужно написать параллельную программу, решающую эту же задачу. Сначала следует распараллелить различные части алгоритма. Необходимо решить, сколько использовать процессов, что каждый из них будет делать и как они будут взаимодействовать и синхронизироваться. Очевидно, что параллельная программа должна быть корректной (давать такие же результаты, как последовательная программа) и свободной от ошибок синхронизации, таких как состояние гонок и взаимные блокировки. Кроме того, желательно достичь как можно более высокой производительности, т.е. получить программу, ускорение которой не зависит (по возможности) от числа процессоров и размера задачи. Вряд ли нам удастся получить идеальное, масштабируемое ускорение, но желательно получить хотя бы "разумное" ускорение. (Неформально "разумное" означает, что рост производительности программы достаточен, чтобы вообще имело смысл использовать многопроцессорные системы.) Например, даже умеренное двухкратное ускорение на машине с четырьмя процессорами позволит решить вдвое большую по объему задачу за то же самое время.

В наибольшей степени на производительность влияет используемый алгоритм, поэтому для получения высокой производительности нужно начинать с хорошего алгоритма. Этот вопрос будет затрагиваться при изучении каждого из приложений в главе 11. Иногда наилучший параллельный алгоритм для решения задачи отличается от наилучшего последовательного алгоритма. Однако пока предположим, что нам дан хороший последовательный алгоритм, и его нужно распараллелить.

Общее время выполнения параллельной программы — это сумма времени самих вычислений и накладных расходов, вызванных параллелизмом, взаимодействием и синхронизацией. Любая параллельная программа должна выполнить такой же общий объем работы, что и последовательная программа, решающая ту же задачу. Следовательно, первая проблема со-

стоит в том, чтобы распараллелить вычисления и назначить процессам процессоры так, чтобы *сбалансировать вычислительную нагрузку*. Пусть T_{comp} — это время работы последовательной программы. Для достижения идеального ускорения на p процессорах нужно так сбалансировать нагрузку, чтобы время работы каждого процессора было близко к T_{comp}/p. Если один процессор загружен больше, чем другой, часть времени тот будет *простаивать*. Таким образом, общее время вычислений и производительность будут определяться временем работы наиболее загруженного процессора.

Кроме последовательных компонентов, у параллельной программы есть еще три источника накладных расходов: 1) создание процессов и их диспетчеризация, 2) взаимодействие и 3) синхронизация. Их нельзя исключить, поэтому вторая проблема — минимизировать их. К сожалению, накладные расходы взаимозависимы, и уменьшение одного может привести к увеличению других.

В параллельной программе есть много процессов, которые нужно создавать и планировать. Стандартный способ уменьшить эти расходы — на каждом процессоре создавать один процесс. Это дает наименьшее число процессов, использующих данное число процессоров. Кроме того, если на каждый процессор приходится по одному процессу, то отсутствуют расходы, связанные с переключением контекста при переходе процессора от выполнения одного процесса к другому. Однако приостановка процесса приводит к простою его процессора. Если бы в программе было больше процессов, мог бы выполняться один из них. Также, когда процессы выполняют различные объемы работы, вычислительную нагрузку сбалансировать намного легче, если процессов больше, чем процессоров. Наконец, некоторые приложения намного проще программируются с помощью рекурсивных или мелкомодульных процессов. Итак, параллельное программирование требует разрешения противоречий между числом процессов, балансировкой нагрузки и накладными расходами при создании и планировании процессов.

Второй источник дополнительных расходов — взаимодействие процессов. Это особенно актуально в программах, использующих обмен сообщениями, поскольку для отправки сообщений нужны определенные действия в ядре получателя и отправителя и собственно перемещение сообщений по сети. Расходы в ядре неустранимы, поэтому для их сокращения важно уменьшить число сообщений. Перемещения сообщений также не избежать, но его можно *замаскировать* (*скрыть*), если во время передачи сообщения у процессора есть другая работа.

Взаимодействие может приводить к накладным расходам даже в программах, которые используют разделяемые переменные и выполняются на машинах с разделяемой памятью. Дело в том, что на этих машинах применяются кэши и аппаратные протоколы для поддержания согласованности кэшей друг с другом и с первичной памятью. Кроме того, у машин с большой разделяемой памятью время доступа к памяти неоднородно. Для уменьшения расходов на доступ к памяти программисту приходится распределять данные так, чтобы каждый процессор работал со своей частью, и размещать каждую часть в модуле памяти, близком к месту ее использования, особенно при записи данных. Кроме того, следует использовать как можно меньше разделяемых переменных. Наконец, нужно избегать *ложного разделения данных*, когда две переменные хранятся в одной строке кэша и используются разными процессорами, причем хотя бы один из них записывает в "свою" переменную.

Синхронизация — это последний источник накладных расходов, как правило, неустранимый. Совместно решая задачу, процессы синхронизируются. В параллельных программах чаще всего используются следующие типы синхронизации: критические секции, барьеры и обмен сообщениями. Для сокращения накладных расходов на синхронизацию желательно ограничивать ее частоту, использовать эффективные протоколы и уменьшать задержки (приводящие к простоям). Например, вместо накопления глобального результата в одной разделяемой переменной, которую нужно защищать критической секцией, можно на каждом процессоре иметь по одной переменной, чтобы процессоры вычисляли для себя глобальное значение, например после барьера. Время дополнительных вычислений обычно будет намного меньше, чем расходы, вносимые критическими секциями. При другом подходе

(в программе с обменом сообщениями) сообщение отправляется как можно раньше, чтобы повысить вероятность того, что сообщение прибудет до того, как оно потребуется другому процессу. Это позволяет уменьшить и иногда даже исключить задержки в процессе-получателе. Описанные и подобные им методы иллюстрируются на конкретных примерах в главе 11.

Итак, при написании параллельной программы нужно начать с выбора хорошего алгоритма. Затем выбрать стратегию расспараллеливания, т.е. решить, сколько использовать процессов и как разделить данные между ними, чтобы сбалансировать вычислительную нагрузку. После этого нужно добавить взаимодействие и синхронизацию, чтобы процессы корректно работали вместе над решением задачи. Проектируя программу, не забывайте о перечисленных выше источниках дополнительных расходов. Наконец, после того, как программа будет написана и проверена на корректность, измерьте и отрегулируйте ее производительность. Под регулированием подразумевается оптимизация программы (вручную или с помощью оптимизаций компилятора), после которой уменьшается общее время выполнения и, следовательно, повышается производительность. В следующих двух главах дается несколько советов, как это сделать. Замечания в конце главы 12 описывают программные системы, используемые для измерения и визуализации производительности.

Глава 11

Научные вычисления

Существует два традиционных метода научных исследований — *теория* и *эксперимент*. Например, теоретическая физика занимается построением моделей, объясняющих физические явления, а экспериментальная физика — их изучением, часто для того, чтобы подтвердить или опровергнуть гипотезы. Теперь появился третий тип исследований — *численное моделирование*, в котором явления имитируются с помощью компьютеров, причем основной вопрос — "Что будет, если...?". Например, физик, применяющий численное моделирование, может написать программу, моделирующую эволюцию звезд или слияние ядер.

Численное моделирование стало возможным в 1960-х годах благодаря изобретению быстродействующих компьютеров. Наверное, правильней сказать, что быстродействующие компьютеры появились *для удовлетворения нужд* инженеров и ученых. Во всяком случае, самые быстродействующие машины с наибольшей памятью всегда используются в научных расчетах. Раньше у таких машин были векторные процессоры, сегодня это машины с массовым параллелизмом, имеющие десятки и сотни процессоров. Численное моделирование постоянно применяется во всех научных и инженерных областях — от разработки новых лекарств, моделирования климата, конструирования самолетов и автомобилей до определения, где сверлить нефтяную скважину, и изучения перемещения загрязнений в водоносных слоях.

Несмотря на множество научных компьютерных приложений и численных моделей, постоянно используются три основных метода: сеточные, точечные и матричные вычисления. Сеточные вычисления применяются в численном решении уравнений в частных производных и других приложениях (таких как обработка изображений), когда пространственная область разбивается на множество точек. Точечные вычисления используются в моделях, имитирующих взаимодействие отдельных частиц, таких как молекулы или звездные объекты. Матричные вычисления применяются всегда, когда нужно решить систему одновременно действующих уравнений.

В данной главе представлены примеры сеточных, точечных и матричных вычислений. Для каждого метода описаны возможные алгоритмы и разработана последовательная программа. Затем составлены параллельные программы, сначала с помощью разделяемых переменных, а затем — передачи сообщений. Также показано, как оптимизировать программы, чтобы повысить их производительность.

11.1. Сеточные вычисления

Дифференциальные уравнения в частных производных (partial differential equations — PDE) применяются для моделирования разнообразных физических систем: прогноза погоды, обтекания крыла потоком воздуха, турбулентности в жидкостях и т.д. Некоторые простые PDE можно решить прямыми методами, но обычно нужно найти приближенное решение уравнения на конечном множестве точек, применяя итерационные численные методы. В этом разделе показано, как решить одно конкретное PDE — двухмерное уравнение Лапласа — с помощью сеточных вычислений по так называемому *методу конечных разностей*. Но при решении других PDE и в других приложениях сеточных вычислений, например, при обработке изображений, используется такая же техника программирования.

11.1.1. Уравнение Лапласа

Уравнение Лапласа является примером так называемого эллиптического дифференциального уравнения в частных производных. В двухмерном варианте это уравнение имеет следующий вид.

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = 0$$

Функция Φ представляет собой некоторый неизвестный потенциал, например, теплоту или напряжение.

По данной области пространства и известным значениям в точках на границах этой области нужно аппроксимировать стационарное решение во внутренних точках области. Это можно сделать, покрыв область равномерной сеткой точек (рис. 11.1). Каждая внутренняя точка инициализируется некоторым значением. Затем с помощью повторяемых итераций вычисляются стационарные значения внутренних точек. На каждой итерации новое значение точки является комбинацией старых и/или новых значений соседних точек. Вычисления прекращаются либо после определенного количества итераций, либо тогда, когда разность между каждым новым и соответствующим предыдущим значением становится меньше заданной величины EPSILON.

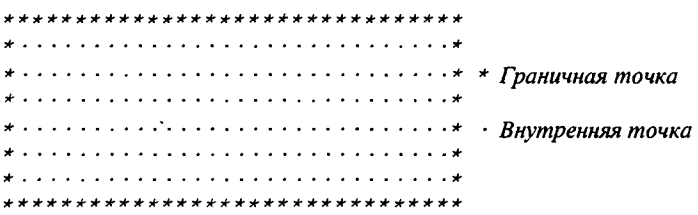


Рис. 11.1. Аппроксимация уравнения Лапласа с помощью сетки

Для решения уравнения Лапласа существует несколько итерационных методов: Якоби, Гаусса–Зейделя, последовательная сверхрелаксация (successive over-relaxation — SOR) и многосеточный. Вначале будет показано, как запрограммировать метод итераций Якоби (с помощью разделяемых переменных и передачи сообщений), поскольку он наиболее прост и легко распараллеливается. Затем покажем, как программируются другие методы, сходящиеся гораздо быстрее. Их алгоритмы сложнее, но схемы взаимодействия и синхронизации в параллельных программах для них аналогичны.

11.1.2. Метод последовательных итераций Якоби

В методе итераций Якоби новое значение в каждой точке сетки равно среднему из предыдущих значений четырех ее соседних точек (слева, справа, сверху и снизу). Этот процесс повторяется, пока вычисление не завершится. Ниже строится простая последовательная программа и приводится ряд оптимизаций кода, повышающих производительность программы.

Предположим, что сетка представляет собой квадрат размером n×n и окружена квадратом граничных точек. Одна матрица нужна для представления области и ее границы, другая — для хранения новых значений.

```
real grid[0:n+1,0:n+1], new[0:n+1,0:n+1];
```

Границы обеих матриц инициализируются соответствующими граничными условиями, а внутренние точки — некоторым начальным значением, например 0. (В первом из последующих алгоритмов граничные значения в матрице new не нужны, но они понадобятся позже.)

Предположим, что вычисления завершаются, когда разность между каждым новым, полученным на итерации, и предыдущим значением по модулю не больше EPSILON. Тогда главный цикл вычислений по методу итераций Якоби имеет следующий вид.

```
while (true) {
  # вычислить новые значения во всех внутренних точках
  for [i = 1 to n, j = 1 to n]
    new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                grid[i,j-1] + grid[i,j+1] / 4;
  iters++;
  # вычислить максимальную разность
  maxdiff = 0.0;
  for [i = 1 to n, j = 1 to n]
    maxdiff = max(maxdiff, abs(new[i,j]-grid[i,j]));
  # проверить условие завершения вычислений
  if (maxdiff < EPSILON)
    break;
  # скопировать new в grid, чтобы подготовить следующие обновления
  for [i = 1 to n, j = 1 to n]
    grid[i,j] = new[i,j];
}
```

Здесь maxdiff имеет действительный тип, iters — целый (это счетчик числа фаз обновления). В программе предполагается, что массивы хранятся в памяти машины по строкам; если же массивы хранятся по столбцам (как в Фортране), то в циклах for сначала должны быть итерации по j, а затем по i.

Приведенный выше код корректен, но не очень эффективен. Однако его производительность можно значительно повысить. Для этого рассмотрим каждую часть программы и сделаем ее более эффективной.

В первом цикле for присваивания выполняются n2 раз в каждой фазе обновлений. Суммирование оставим без изменений, а деление на 4 заменим умножением на 0.25. Такая замена повысит производительность, поскольку умножение выполняется быстрее, чем деление. Очевидно, что данная операция выполняется много раз, поэтому улучшение будет заметным. Такая оптимизация называется *сокращением мощности*, поскольку заменяет "мощное" (дорогое) действие более слабым (дешевым). (В действительности для целых значений деление на 4 можно заменить еще более слабым действием — смещением вправо на 2.)

Рассмотрим часть кода, в которой вычисляется максимальная разность. Эта часть выполняется на каждой итерации цикла while, но только один раз приводит к выходу из цикла. *Намного* эффективней заменить цикл while конечным циклом, в котором итерации выполняются определенное количество раз. По существу, iters и maxdiff меняются ролями. Вместо того, чтобы подсчитывать число итераций и использовать maxdiff для завершения вычислений, iters теперь используется для управления количеством итераций. Тогда максимальная разность будет вычисляться только один раз *после* главного цикла вычислений. После проведения этой и первой оптимизаций программа примет следующий вид.

```
for [iters = 1 to MAXITERS] {
  # вычислить новые значения во всех внутренних точках
  for [i = 1 to n, j = 1 to n]
    new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                grid[i,j-1] + grid[i,j+1] * 0.25;
  # скопировать new в grid, чтобы подготовить следующие обновления
  for [i = 1 to n, j = 1 to n]
    grid[i,j] = new[i,j];
}
# вычислить максимальную разность
maxdiff = 0.0;
for [i = 1 to n, j = 1 to n]
  maxdiff = max(maxdiff, abs(new[i,j]-grid[i,j]));
```

Здесь MAXITERS — константа или аргумент в командной строке. Хотя этот код не эквивалентен исходному, он по-прежнему реализует метод итераций Якоби. Если программист при запуске нового кода обнаружит, что конечное значение maxdiff слишком велико, следует перезапустить программу с большим значением MAXITERS. Чтобы подобрать нужное значение MAXITERS, достаточно сделать несколько пробных запусков программы. Затем подобранное таким образом значение MAXITERS можно использовать в рабочих запусках программы.

Вторая версия кода намного эффективнее, чем первая, однако ее можно еще улучшить. Рассмотрим цикл копирования матрицы new в grid. Его нельзя просто удалить, обновляя значения точек сетки на месте, поскольку в итерации Якоби при вычислении новых значений нужны предыдущие значения точек. Однако цикл копирования занимает значительное время. Намного эффективнее использовать указатели, ссылающиеся на матрицы, и после каждой фазы обновлений менять их местами. Таким образом можно скомбинировать две матрицы в одну трехмерную.

```
real grid[0:1][0:n+1,0:n+1];
int old = 0, new = 1;
```

Переменные old и new служат для индексации первой размерности grid.

Учитывая измененное представление матриц, главный цикл можно записать так.

```
for [iters = 1 to MAXITERS] {
  # вычислить новые значения во всех внутренних точках
  for [i = 1 to n, j = 1 to n]
    grid[new][i,j] = (grid[old][i-1,j] +
         grid[old][i+1,j] + grid[old][i,j-1] +
         grid[old][i,j+1]) * 0.25;
  # поменять местами роли матриц
  old = 1-old; new = 1-new;
}
```

Чтобы матрицы поменялись ролями, достаточно поменять местами значения old и new. Однако избавление от цикла копирования не проходит даром: теперь каждая ссылка на grid в цикле обновлений имеет дополнительный индекс, т.е. для вычисления адресов значений в grid нужны дополнительные машинные команды. И все же дополнительное время для выполнения этих команд меньше, чем время, сэкономленное при удалении цикла копирования.

Выбора между циклом копирования и более дорогой индексацией массива можно избежать, проведя еще одну оптимизацию — *развертку цикла*. Суть ее в том, что код цикла повторяется несколько раз и во столько же раз уменьшается число итераций. Например, развернув внешний цикл дважды, получим такой код.

```
for [iters = 1 to MAXITERS by 2] {
  # вычислить новые значения во всех внутренних точках
  for [i = 1 to n, j = 1 to n]
    grid[new][i,j] = (grid[old][i-1,j] +
         grid[old][i+1,j] + grid[old][i,j-1] +
         grid[old][i,j+1]) * 0.25;
  # поменять местами роли матриц
  old = 1-old; new = 1-new;
  #вычислить новые значения во всех внутренних точках
  for [i = 1 to n, j = 1 to n]
    grid[new][i,j] = (grid[old][i-1,j] +
         grid[old][i+1,j] + grid[old][i,j-1] +
         grid[old][i,j+1]) * 0.25;
  #поменять местами роли матриц
  old = 1-old; new = 1-new;
}
```

Развертывание цикла само по себе мало влияет на рост производительности, но часто позволяет применить другие оптимизации.[21] Например, в последнем коде больше не нужно менять местами роли матриц. Достаточно переписать второй цикл обновлений так, чтобы данные считывались из new, полученной в первом цикле обновлений, и записывались в старую матрицу grid. Но тогда трехмерная матрица не нужна, и можно вернуться к двум отдельным матрицам!

В листинге 11.1 представлена оптимизированная программа для метода итераций Якоби. Проведены четыре оптимизации исходного кода: 1) деление заменено умножением; 2) для завершения вычислений использовано конечное число итераций, а максимальная разность вычисляется только один раз; 3) две матрицы скомбинированы в одну с дополнительным измерением, что избавляет от копирования матриц; 4) цикл развернут дважды и его тело переписано, поскольку дополнительный индекс и операторы изменения ролей матриц не нужны. В действительности в данной задаче можно было бы перейти от второй оптимизации прямо к четвертой. Однако третья оптимизация часто полезна.

Листинг 11.1. Оптимизированная последовательная программа для метода итераций Якоби

```
real grid[0:n+1,0:n+1], new[0:n+1,0:n+1];
real maxdiff = 0.0;
инициализация сеток, включая границы;
for [iters = 1 to MAXITERS by 2] {
  # вычислить новые значения во всех внутренних точках
  for [i = 1 to n, j = 1 to n]
    new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                grid[i,j-1] + grid[i,j+1]) * 0.25;
# снова вычислить новые значения во внутренних точках
  for [i = 1 to n, j = 1 to n]
    grid[i,j] = (new[i-1,j] + new[i+1,j] +
                 new[i,j-1] + new[i,j+1]) * 0.25;
}
# вычислить максимальную разность
for [i = 1 to n, j = 1 to n]
  maxdiff = max(maxdiff, abs(grid[i,j]-new[i,j]));
печать окончательных значений и максимальной разности;
```

Представленную в листинге 11.1 программу можно оптимизировать еще двумя способами. Поскольку новые значения точек вычисляются дважды в теле внешнего цикла for, из первого цикла вычислений можно убрать умножение на 0.25, а во втором вставить умножение на 0.0625. Такая замена избавит от половины всех умножений. Программу можно улучшить, *встраивая* вызовы функций в последнем цикле, вычисляющем максимальную разность. Вызовы функций max и abs можно заменить телами этих функций.

```
temp = grid[i,j]-new[i,j];
if (temp < 0)
  temp = -temp;
if (temp > maxdiff)
  maxdiff = temp;
```

Это позволяет избежать накладных расходов на два вызова функций. Здесь использовано также то, что maxdiff и является аргументом max, и сохраняет результат. Встраивание функции мало повышает производительность, поскольку максимальная разность вычисляется в программе только один раз. Однако, если функция вызывается во внутреннем цикле, выполняемом многократно, встраивание может оказаться очень эффективным.

[21] Производительность ухудшится, если развертываний цикла настолько много, что команды в теле цикла не умещаются в кэше команд процессора.

11.1.3. Метод итераций Якоби с разделяемыми переменными

Рассмотрим, как распараллелить метод итераций Якоби. В листинге 3.16 был приведен пример программы, параллельной по данным. В ней на одну точку сетки приходился один процесс, что дает степень параллельности, максимально возможную для данной задачи. Хотя программа эффективно работала бы на синхронном мультипроцессоре (SIMD-машине), в ней слишком много процессов, чтобы она могла эффективно выполняться на обычном MIMD-мультипроцессоре. Дело в том, что каждый процесс выполняет очень мало работы, поэтому во времени выполнения программы будут преобладать накладные расходы на переключение контекста. Кроме того, каждому процессу нужен свой собственный стек, поэтому, если сетка велика, программа может не поместиться в памяти. Следовательно, производительность параллельной программы окажется гораздо хуже, чем последовательной.

Предположим, что есть PR процессоров, и размерность сетки n намного больше PR. Чтобы параллельная программа была эффективной, желательно распределить вычисления между процессорами поровну. Для данной задачи сделать это несложно, поскольку для обновления каждой точки сетки нужно одно и то же количество работы. Следовательно, можно использовать PR процессов так, чтобы каждый из них отвечал за одно и то же число точек сетки. Можно разделить сетку на PR прямоугольных блоков или прямоугольных полос. Используем полосы, поскольку для них немного проще написать программу. Вероятно также, что использование полос более эффективно, поскольку при длинных полосах локальность данных выше, чем при коротких блоках, что оптимизирует использование кэша данных.

Предположив для простоты, что n кратно PR, а массивы хранятся в памяти по строкам, назначим каждому процессу горизонтальную полосу размера n/PR × n. Каждый процесс обновляет свою полосу точек. Однако, поскольку процессы имеют общие точки, расположенные на границах полос, после каждой фазы обновлений нужна барьерная синхронизация для того, чтобы все процессы завершили фазу обновлений перед тем, как любой процесс начнет выполнять следующую.

В листинге 11.2 приведена параллельная программа для метода итераций Якоби с разделяемыми переменными. Использован стиль "одна программа — много данных" (single program, multiple data — SPMD): каждый процесс выполняет один и тот же код, но обрабатывает различные части данных. Здесь каждый рабочий процесс сначала инициализирует свои полосы, включая границы, в двух матрицах. Тело каждого рабочего процесса основано на программе в листинге 11.1. Барьерная синхронизация происходит с помощью разделяемой процедуры, реализующей один из алгоритмов в разделе 3.4. Для достижения максимальной эффективности можно применить симметричный барьер, например, барьер с распространением.

Листинг 11.2. Метод итераций Якоби с разделяемыми переменными

```
real grid[0:n+1,0:n+1], new[0:n+1,0:n+1];
int HEIGHT = n/PR;    # предполагается, что n кратно PR
real maxdiff[1:PR] = ([PR] 0.0);

procedure barrier(int id) {
  # эффективный алгоритм барьера из раздела 3.4
}

process worker[w = 1 to PR] {
  int firstRow = (w-1)*HEIGHT + 1;
  int lastRow = firstRow + HEIGHT - 1;
  real mydiff = 0.0;
  инициализация своей полосы, включая границы, в grid и new;
  barrier(w);
  for [iters =1 to MAXITERS by 2] {
    # вычислить новые значения в своей полосе
```

```
    for [i = firstRow to lastRow, j = 1 to n]
      new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                  grid[i,j-1] + grid[i,j+1])* 0.25;
    barrier(w);
    # снова вычислить значения в своей полосе
    for [i = firstRow to lastRow, j = 1 to n]
      grid[i,j] = (new[i-1,j] + new[i+1,j] +
                   new[i,j-1] + new[i,j+1])* 0.25;
    barrier(w);
  }
  # вычислить максимальную разность в своей полосе
  for [i = firstRow to lastRow, j = 1 to n]
    mydiff = max(mydiff, abs(grid[i,j]-new[i,j]));
  maxdiff[w] = mydiff;
  barrier(w);
  #максимальная разность — это максимум в maxdiff[*]
}
```

Листинг 11.2 также иллюстрирует эффективный способ вычисления максимальной разности во всех парах окончательных значений в точках сетки. Каждый рабочий процесс вычисляет максимальную разность для точек в своей полосе, используя *локальную* переменную `mydiff`, а затем сохраняет полученное значение в массиве максимальных разностей. После второго барьера каждый рабочий процесс может параллельно вычислить максимальное значение в `maxdiff[*]` (все эти вычисления можно также проводить только в одном рабочем процессе). Локальные переменные в каждом процессе позволяют избежать использования критических секций для защиты доступа к единственной глобальной переменной, а также конфликтов в кэше, которые могли бы возникнуть из-за ложного разделения массива `maxdiff`.

Программа в листинге 11.2 могла бы работать немного эффективнее, если встроить вызовы процедуры `barrier`. Однако встраивание кода вручную сделает программу тяжелой для чтения. Поэтому лучше всего использовать компилятор, поддерживающий оптимизацию встраивания.

11.1.4. Метод итераций Якоби с передачей сообщений

Рассмотрим реализацию метода итераций Якоби на машине с распределенной памятью. Можно было бы использовать реализацию распределенной разделяемой памяти (раздел 10.4) и просто взять программу из листинга 11.2. Однако такая реализация не всегда доступна и вряд ли дает наилучшую производительность. Как правило, эффективнее написать программу с передачей сообщений.

Один из способов написать параллельную программу с передачей сообщений — модифицировать программу с разделяемыми переменными. Сначала разделяемые переменные распределяются между процессами, затем в тех местах, где процессам нужно обменяться данными, добавляются операторы отправки и получения. Другой способ — сначала изучить парадигмы взаимодействия процессов, описанные в разделах 9.1–9.3 (портфель задач, алгоритм пульсации и конвейер), и выбрать подходящую. Часто, как в нашем примере, можно использовать оба метода.

Начав с программы в листинге 11.2, вновь используем `PR` рабочих процессов, обновляющих каждый раз полосу точек сетки. Распределим массивы `grid` и `new` так, чтобы полосы были локальными для соответствующих рабочих процессов. Также нужно продублировать строки на краях полос, поскольку рабочие не могут считывать данные, размещенные в других процессах. Таким образом, каждому процессу нужно хранить точки не только своей полосы, но и границ соседних полос. Каждый процесс выполняет последовательность фаз обновления; после каждого обновления он обменивается краями своей полосы с соседями. Такая схема обмена соответствует алгоритму пульсации (см. раздел 9.2). Обмены заменяют точки барьерной синхронизации в листинге 11.2.

Нужно также распределить вычисление максимальной разности после того, как каждый процесс завершит обновление в своей полосе сетки. Как и в листинге 11.2, каждый процесс просто вычисляет максимальную разность в своей полосе. Затем выбирается один процесс, который собирает все полученные значения. Все это можно запрограммировать, используя либо сообщения, передаваемые от процесса к процессу, либо коллективное взаимодействие, как в MPI (раздел 7.8).

В листинге 11.3 представлена программа для метода итераций Якоби с передачей сообщений. В каждой из двух фаз обновления используется алгоритм пульсации, поэтому соседние процессы дважды обмениваются краями во время одной итерации главного вычислительного цикла. Первый обмен программируется следующим образом.

```
if (w > 1)  send up[w-1](new[1,*]);
if (w < PR) send down[w+1](new[HEIGHT,*]);
if (w < PR) receive up[w](new[HEIGHT+1,*]);
if (w > 1)  receive down[w](new[0,*]);
```

Все процессы, кроме первого, отправляют верхний ряд своей полосы соседу сверху. Все процессы, кроме последнего, отправляют нижний ряд своей полосы соседу снизу. Затем каждый получает от своих соседей края; они становятся границами его полосы. Второй обмен идентичен первому, только вместо new используется grid.

Листинг 11.3. Метод итераций Якоби с передачей сообщений

```
chan up[1:PR](real edge[0:n+1]);
chan down[1:PR](real edge[0:n+1]);
chan diff(real);

process worker[w = 1 to PR] {
  int HEIGHT = n/PR;     # n кратно PR
  real grid[0:HEIGHT+1,0:n+1], new[0:HEIGHT+1,0:n+1];
  real mydiff = 0.0, otherdiff = 0.0;
  инициализация grid и new с границами;
  for [iters = 1 to MAXITERS by 2] {
  #вычислить новые значения в своей полосе
    for [i = 1 to HEIGHT, j = 1 to n]
      new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                  grid[i,j-1] + grid[i,j+1]) * 0.25;
    обмен краями в матрице new (см. текст);
    # снова вычислить новые значения в своей полосе
    for [i = 1 to HEIGHT, j = 1 to n]
      grid[i,j] = (new[i-1,j] + new[i+1,j] +
                   new[i,j-1] + new[i,j+1]) * 0.25;
    обмен краями в матрице grid (см. текст);
  }
  # вычислить максимальную разность в своей полосе
  for [i = 1 to HEIGHT, j = 1 to n]
    mydiff = max(mydiff, abs(grid[i,j]-new[i,j]));
  if (w > 1)
    send diff(mydiff);
  else        # рабочий процесс 1 собирает разности
    for [i = 1 to w-1] {
      receive diff(otherdiff);
      mydiff = max(mydiff, otherdiff);
    }
  # максимальная разность - это значение mydiff в процессе 1
}
```

После подходящего числа итераций каждый рабочий процесс вычисляет максимальную разность в своей полосе, затем первый из них собирает полученные значения. Как отмечено в конце листинга, глобальная максимальная разность равна окончательному значению mydiff в первом процессе.

Программу в листинге 11.3 можно оптимизировать. Во-первых, для данной программы и многих других сеточных вычислений нужно проводить обмен краями после каждой фазы обновлений. Здесь, например, можно обменивать края после каждой второй фазы обновлений. Это вызовет "скачки" значений в точках на краях, но, поскольку алгоритм сходится, он все равно будет давать правильный результат. Во-вторых, можно перепрограммировать оставшийся обмен так, чтобы между отправкой и получением сообщений выполнялись локальные вычисления. Например, можно реализовать взаимодействие, при котором каждый рабочий: 1) отправляет свои края соседям; 2) обновляет внутренние точки своей полосы; 3) получает края от соседей; 4) обновляет края своей полосы. Такой подход значительно повысит вероятность того, что края от соседей придут раньше, чем они нужны, и, следовательно, задержки операций получения не будет. В листинге 11.4 представлена оптимизированная программа. Предлагаем читателю сравнить производительность и результаты последних двух программ с передачей сообщений.

Листинг 11.4. Улучшенный метод итераций Якоби с передачей сообщений

```
chan up[1:PR](real edge[0;n+1]);
chan down[1:PR](real edge[0:n+1]);
chan diff(real);

process worker[w = 1 to PR] {
  int HEIGHT = n/PR;     # n кратно PR
  real grid[0:HEIGHT+1,0:n+1], new[0:HEIGHT+1,0:n+1];
  real mydiff = 0.0, otherdiff = 0.0;
  инициализация grid и new, включая границы;
  for [iters = 1 to MAXITERS by 2] {
    # вычислить новые значения в своей полосе
    for [i = 1 to HEIGHT, j = 1 to n]
       new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                   grid[i,j-1] + grid[i,j+1]) * 0.25;
    # отправить края в new соседям
    if (w > 1)
      send up[w-1](new[1,*]);
    if (w < PR)
      send down[w+1](new[HEIGHT,*]);
    # вычислить новые значения во внутренних точках своей полосы
    for [i = 2 to HEIGHT-1, j = 1 to n]
        grid[i,j] = (new[i-1,j] + new[i+1,j] +
                     new[i,j-1] + new[i,j+1]) * 0.25;
    # получить края в new от соседей
    if (w < PR)
      receive up[w](new[HEIGHT+1,*]);
    if (w > 1)
       receive down[w](new[0,*]);
    # вычислить новые значения на краях своей полосы
    for [j = 1 to n]
      grid[1,j] = (new[0,j] + new[2,j] +
                   new[1,j-1] + new[1,j+1]) * 0.25;
    for [j = 1 to n]
      grid[HEIGHT,j] = (new[HEIGHT-1,j] +
                        new[HEIGHT+1,j] + new[HEIGHT,j-1] +
                        new[HEIGHT,j+1]) * 0.25;
  }
  вычислить максимальную разность как в листинге 11.3
}
```

11.1.5. Последовательная сверхрелаксация по методу "красное-черное"

Метод итераций Якоби сходится очень медленно, поскольку для того, чтобы значения некоторой точки повлияли на значения удаленных от нее точек, нужно длительное время. Например, нужны n/2 фаз обновлений, чтобы влияние граничных значений сетки дошло до ее центра.

Схема Гаусса–Зейделя (GS) сходится намного быстрее и к тому же использует меньше памяти. Новые значения в точках вычисляются с помощью комбинации старых и новых значений соседних точек. Проход по сетке слева направо и сверху вниз образует развертку, почти как для телевизионных изображений на электронно-лучевой трубке. Новые точки вычисляются *на месте* следующим образом.

```
for [i = 1 to n, j = 1 to n]
  grid[i,j] = (grid[i-1,j] + grid[i,j-1] +
               grid[i+1,j] + grid[i,j+1]) * 0.25;
```

Таким образом, каждое новое значение зависит от только что вычисленных значений слева и сверху и от предыдущих значений справа и снизу. При обновлении точек на месте вторая матрица не нужна.

Последовательная сверхрелаксация (successive over-relaxation — SOR) является обобщением метода Гаусса–Зейделя. По методу SOR новые точки вычисляются на месте таким образом.

```
for [i = 1 to n, j = 1 to n]
  grid[i,j] = omega * (grid[i-1,j] + grid[i,j-1] +
                       grid[i+1,j] + grid[i,j+1]) * 0.25
            + (1-omega) * grid[i,j];
```

Переменная omega называется параметром релаксации; ее значение выбирается из диапазона $0 < \text{omega} < 2$. Если omega равна 1, то метод SOR сводится к методу Гаусса–Зейделя. Если ее значение равно 0.5, новое значение точки сетки есть половина среднего значения ее соседей плюс половина ее старого значения. Выбор подходящего значения omega зависит от решаемого дифференциального уравнения в частных производных и граничных условий.

Хотя методы GS и SOR сходятся быстрее, чем метод итераций Якоби, и требуют вдвое меньше памяти, непосредственно распараллелить их непросто, поскольку при обновлении точки используются как старые, так и новые значения. Другими словами, методы GS и SOR позволяют обновлять точки на месте именно потому, что применяется последовательный порядок обновлений. (В циклах этих двух алгоритмов присутствует так называемая зависимость по данным. Их можно распараллелить, используя волновые фронты. Оба вопроса обсуждаются в разделе 12.2.)

К счастью, GS и SOR можно распараллелить, слегка изменив сами алгоритмы (их сходимость сохранится). Сначала покрасим точки в красный и черный цвет, как клетки шахматной доски. Начав с верхней левой точки, окрасим точки через одну красным цветом, а остальные — черным. Таким образом, у красных точек будут только черные соседи, а у черных — только красные. Заменим единый цикл в фазе обновлений двумя вложенными: в первом обновляются значения красных точек, а во втором — черных.

Теперь схему "красное-черное" можно распараллелить: у красных точек все соседи только черного цвета, а у черных — красного. Следовательно, все красные точки можно обновлять параллельно, поскольку их значения зависят только от старых значений черных точек. Так же обновляются и черные точки. Однако после каждой фазы обновлений нужна барьерная синхронизация, чтобы перед началом обновления черных точек было завершено обновление всех красных точек и наоборот.

В листинге 11.5 приведена параллельная программа для метода "красное-черное" по схеме Гаусса–Зейделя, использующая разделяемые переменные. (Последовательная сверхрелаксация отличается только выражением для обновления точек.) Вновь предположим, что есть PR рабочих процессов и n кратно PR. Разделим сетку на горизонтальные полосы и назначим каждому процессу по одной полосе.

Листинг 11.5. Схема Гаусса–Зейделя по методу "красное-черное" с разделяемыми переменными

```
real grid[0:n+1,0:n+1],
int HEIGHT = n/PR; # n кратно PR
real maxdiff[1:PR] = ([PR] 0.0);

procedure barrier(int id) {
  # эффективный алгоритм барьера из раздела 3.4
}

process worker[w = 1 to PR] {
  int firstRow = (w-1)*HEIGHT + 1;
  int lastRow = firstRow + HEIGHT - 1;
  int jStart;
  real mydiff = 0.0;
  инициализация своей полосы grid, включая границы;
  barrier(w);
  for [iters = 1 to MAXITERS] {
    #вычислить новые значения для красных точек своей полосы
    for [i = firstRow to lastRow] {
      if (i%2 == 1) jStart = 1;      #нечетный ряд
      else jStart = 2;               #четный ряд
      for [j = jStart to n by 2]
        grid[i,j] = (grid[i-1,j] + grid[i,j-1] +
                     grid[i+1,j] + grid[i,j+1]) * 0.25;
    }
    barrier(w);
    #вычислить новые значения для черных точек своей полосы
    for [i = firstRow to lastRow] {
    if (i%2 == 1) jStart = 2;      #нечетный ряд
    else jStart = 1;               #четный ряд
    for [j = jStart to n by 2]
        grid[i,j] = (grid[i-1,j] + grid[i,j-1] +
                     grid[i+1,j] + grid[i,j+1]) * 0.25;
    }
    barrier(w);
  }
  #вычислить максимальную разность на своей полосе
  выполнить еще одну серию обновлений, отслеживая максимальную
    разность между старым и новым значениями grid[i,j];
  maxdiff[w] = mydiff;
  barrier(w);
  #максимальная разность есть максимум в maxdiff[*]
}
```

По структуре программа идентична представленной в листинге 11.2 программе для метода итераций Якоби. Более того, максимальная разность вычисляется здесь точно так же. Однако теперь в каждой фазе вычислений обновляется вдвое меньше точек по сравнению с соответствующей фазой в параллельной программе для метода итераций Якоби. Соответственно, и внешний цикл выполняется вдвое большее число раз. Это приводит к удвоению числа барьеров, поэтому дополнительные расходы из-за барьерной синхронизации будут составлять более значительную часть от общего времени выполнения. С другой стороны, при одном и том же значении MAXITERS данный алгоритм приведет к лучшим результатам (или к сравнимым результатам при меньших значениях MAXITERS).

Метод "красное-черное" можно запрограммировать, используя передачу сообщений. Программа будет иметь такую же структуру, как и соответствующая программа для метода

итераций Якоби (листинги 11.3 или 11.4). Как и раньше, каждый рабочий процесс отвечает за обновление точек своей полосы, и соседним процессам нужно обмениваться краями после каждой фазы обновлений. Красные и черные точки на краях можно обменивать отдельно, но, если они обмениваются вместе, нужно меньше сообщений.

Мы предположили, что отдельные точки окрашены в красный или черный цвет. В результате `i` и `j` пошагово увеличиваются на два во всех циклах обновлений. Это приводит к слабому использованию кэша: в каждой фазе обновлений доступна вся полоса целиком, но из двух соседних точек читается или записывается только одна.[22] Можно повысить производительность, окрашивая блоки точек, как квадраты на шахматной доске, состоящие из множеств точек. Еще лучше окрасить полосы точек. Например, разделить каждую полосу рабочего пополам (по горизонтали) и закрасить верхнюю половину красным цветом, а нижнюю — черным. В листинге 11.5 каждый рабочий многократно обновляет сначала красные, затем черные точки, и после каждой фазы обновлений установлен барьер. В каждой фазе обновлений доступна только половина всех точек (каждая и читается, и записывается).

11.1.6. Многосеточные методы

Если при аппроксимации уравнений в частных производных используются сеточные вычисления, зернистость сетки влияет на время вычислений и точность решения. На грубых (крупнозернистых) сетках решение находится быстрее, но упускаются некоторые его подробности. Мелкие сетки дают более точные решения, но за большее время. Моделирование физических систем обычно включает развитие системы во времени, поэтому сеточные вычисления нужны на каждом временном шаге. Это обостряет противоречия между временем вычислений и точностью решения.

Предсказание погоды является типичным приложением такого рода. Процесс моделирования погоды начинается с текущих условий (температура, давление, скорость ветра и т.п.), а затем для предсказания будущих условий проходит по временным шагам. Предположим, нужно сделать прогноз для континентальных США (без Аляски), используя сетку с разрешением в одну милю. Тогда понадобятся около 3000×1500 (4,5 миллиона) точек и вычисления такого масштаба на каждом временном шаге! Чтобы учесть значительное влияние океанов, нужна еще большая сетка. Однако с сеткой такого размера прогноз может безнадежно опоздать! Используя более грубую сетку с разрешением, скажем, 10 миль, вычисления можно будет завершить достаточно быстро, но, возможно, "потеряв" локальные возмущения.

Для достаточно быстрого и результативного решения подобных задач существует два подхода. Один из них состоит в применении *адаптивной сетки*. При этом подходе зернистость сетки непостоянна. Она грубая там, где решение относительно однородно, и точная, где решение нуждается в детализации. Кроме того, зернистость может не быть постоянной, чтобы адаптироваться к изменениям, происходящим по мере имитации. Например, на плоской равнине и под центрами высокого давления погода, как правило, однородна, но варьируется вблизи гор и быстро изменяется на границах фронтов.

Второй способ быстрого решения задач большого размера состоит в применении *множественных сеток*. Используются сетки различной зернистости и переходы между ними в ходе вычислений, чтобы ускорить сходимость на самой точной сетке. Многосеточные методы используют так называемую *коррекцию грубой сетки*, состоящую из следующих этапов.

- Начать с точной сетки и обновить точки путем нескольких итераций, применив один из методов релаксации (Якоби, Гаусса–Зейделя, последовательной сверхрелаксации).
- Ограничить полученный результат более грубой сеткой, точки которой вдвое дальше друг от друга. Соответствующие граничные точки имеют одни и те же значения на

[22] Сгенерированный машинный код будет выполняться медленнее еще и потому, что в нем больше команд ветвления. Кроме того, увеличение параметра цикла на два может выполняться медленнее, чем на один.

обеих сетках. Для присвоения значений внутренним точкам грубой сетки на основе значений в соседних точках точной используется *оператор ограничения*.

- Найти решение на грубой сетке с необходимой степенью точности.
- Интерполировать грубую сетку обратно в точную. Это делается с помощью *оператора интерполяции*, который присваивает значения внутренним точкам точной сетки на основе значений в точках грубой.
- Обновить точки точной сетки, проведя несколько итераций.

Существуют различные варианты описанного основного метода. Рассмотрим некоторые из них.

В качестве конкретного примера рассмотрим перекрывающиеся сетки (рис. 11.2). Точная сетка содержит 81 точку (точки и кружки), грубая — 25 (только кружки). Угловые точки обеих сеток совпадают, поскольку сетки покрывают одну и ту же область пространства, но расстояние между точками в грубой сетке вдвое больше.

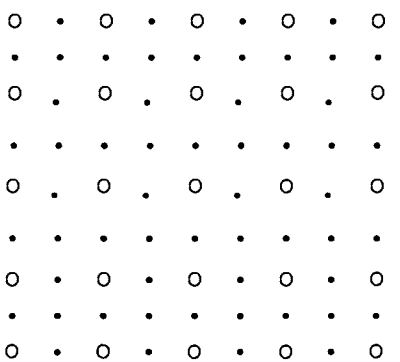


Рис. 11.2. Точная и грубая сетки с граничными точками

Существует много операторов ограничения для инициализации точек в грубой сетке. Типичный оператор определяется следующей матрицей.

$$\begin{bmatrix} 0 & 1/8 & 0 \\ 1/8 & 1/2 & 1/8 \\ 0 & 1/8 & 0 \end{bmatrix}$$

Данная матрица гласит, что значение в каждой точке грубой сетки складывается из половины начального значения в соответствующей точке точной сетки и суммы значений в четырех ближайших соседях точки точной сетки, деленных на восемь. Пусть `coarse[x,y]` — точка грубой сетки, а `fine[i,j]` — соответствующая ей точка точной сетки. Тогда `coarse[x,y]` обновляется следующим образом.

```
coarse[x,y] = fine[i,j] * 0.5 + (fine[i-1,j] +
    fine[i,j-1] + fine[i,j+1] + fine[i+1,j]) * 0.125;
```

В операторах ограничения можно также использовать различные веса (дающие в сумме 1) или включать больше соседей.

Существуют различные операторы интерполяции для перехода от грубой сетки обратно к точной. Типичный оператор, называемый *билинейной интерполяцией*, определяется следующей матрицей.

$$\begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{bmatrix}$$

Значение в точке грубой сетки копируется в соответствующую точку точной сетки, половина его копируется в четыре ближайшие точки точной сетки, а четверть — в четыре точки, ближайшие по диагонали. Таким образом, точка точной сетки, расположенная между двумя точками грубой сетки, получает по половине их значений, а точка в центре квадрата из четырех точек грубой сетки получает по четверти их значений.

С помощью описанного оператора интерполяции можно инициализировать внутренние точки точной сетки следующим образом. Сначала присвоить точки грубой сетки соответствующим точкам точной.

```
fine[i,j] = coarse[x,y];
```

Затем обновить другие точки точной сетки в *столбцах*, которые только что были обновлены. Одно такое присвоение имеет следующий вид.

```
fine[i-1,j] = (fine[i-2,j] + fine[i,j]) * 0.5;
```

Наконец обновить остальные точки точной сетки (в столбцах, состоящих только из точек на рис. 11.2), присвоив каждой из них среднее арифметическое значений в соседних точках ее строки. Одно присвоение имеет такой вид.

```
fine[i-1,j-1] = (fine[i-1,j-2] + fine[i-1,j]) * 0.5;
```

Таким образом, значение в точке, расположенной в той же строке, что и точки грубой сетки, равно среднему арифметическому значений в ближайших точках грубой сетки, а в точке, расположенной в другой строке, — среднему значений, соседствующих по горизонтали, т.е. четверти суммы значений в четырех ближайших точках грубой сетки.

Существует несколько видов многосеточных методов. В каждом из них используются разные схемы коррекции грубой сетки. В простейшем методе используется одна коррекция: вначале на точной сетке выполняем несколько итераций, переходим на грубою сетку и решаем задачу на ней, затем интерполируем полученное решение на точную сетку и выполняем на ней еще несколько итераций. Такая схема называется двухуровневым V-циклом.

На рис. 11.3 показаны три общие многосеточные схемы, в которых используются четыре различных размера сеток. Если расстояние между точками самой точной сетки равно *h*, то расстояние между точками других сеток — *2h*, *4h* и *8h* (рис. 11.3). *V-цикл* в общем виде имеет несколько уровней: вначале на самой точной сетке выполняем несколько итераций, переходим к более грубой сетке, снова выполняем несколько итераций, и так до тех пор, пока не окажемся на самой грубой сетке. На ней решим задачу. Затем интерполируем полученное решение на более точною сетку, выполним несколько итераций, и так далее, пока не проведем несколько итераций на самой точной сетке.

W-цикл повышает точность с помощью многократных переходов между сетками. На более грубых сетках проводятся дополнительные релаксации, и снова, как обычно, точное решение задачи находится на самой грубой сетке.

В *полном многосеточном методе* различные сетки используются столько же раз, сколько и в *W*-цикле, но при этом достигаются более точные результаты. В нем перед тем, как впервые использовать самую точную сетку, несколько раз выполняются вычисления на более грубых сетках, поэтому начальные значения для самой точной сетки оказываются правильнее. Полный многосеточный метод основан на рекуррентной последовательности шагов: 1) создаем начальное приближение к решению на самой грубой сетке, затем вычисляем на ней точное решение; 2) переходим к более точной сетке, выполняем на ней несколько итераций, возвращаемся к более грубой сетке и вновь находим на ней точное решение; 3) повторяем этот процесс, каждый раз поднимаясь на один уровень выше, пока не окажемся на самой точной сетке, после чего проходим полный V-цикл.

Многосеточные методы сходятся намного быстрее, чем основные итерационные. Однако программировать их намного сложнее, поскольку в них обрабатываются сетки различных размеров и нужны постоянные переходы между ними (ограничения и интерполяции)

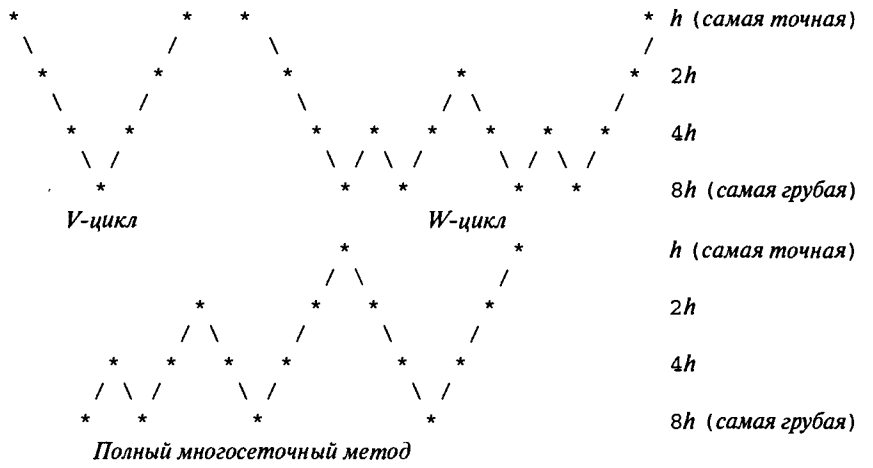


Рис. 11.3. Схемы для многосеточных методов

Имея последовательную программу для многосеточного метода, можно разделить каждую решетку на полосы и распараллелить процессы ограничения, обновления и интерполяции практически так же, как были распараллелены методы итераций Якоби и Гаусса–Зейделя. В программе с разделяемыми переменными после каждой фазы нужны барьеры, а в программе с передачей сообщений — обмен краями полос. Однако получить хорошее ускорение с помощью параллельной многосеточной программы непросто. Переходы между сетками и дополнительные точки синхронизации увеличивают накладные расходы. Кроме того, многосеточный метод дает хороший результат быстрее, чем простая итерационная схема, за счет того, что на всех сетках, кроме самой грубой, выполняется лишь несколько итераций. В результате накладные расходы занимают бо́льшую часть общего времени выполнения. И все же разумное ускорение достижимо, особенно на сетках очень больших размеров, а именно к ним и применяются многосеточные методы.

11.2. Точечные вычисления

Сеточные вычисления обычно используются для моделирования непрерывных систем, которые описываются дифференциальными уравнениями в частных производных. Для моделирования дискретных систем, состоящих из отдельных частиц (точек), воздействующих друг на друга, применяются точечные методы. Примерами являются заряженные частицы, взаимодействующие с помощью электрических и магнитных сил, молекулы (их взаимодействие обусловлено химическим строением) и астрономические тела, воздействующие друг на друга благодаря гравитации. Здесь рассматривается типичное приложение, называемое гравитационной задачей n тел. После постановки задачи представлены последовательная и параллельная программы, основанные на алгоритме сложности $O(n^2)$, затем описаны два приближенных метода сложности $O(n\log_2 n)$: метод Барнса–Хата (Barnes–Hut) и быстрый метод мультиполей.

11.2.1. Гравитационная задача n тел

Предположим, что дано большое число астрономических тел галактики (звезд, пылевых облаков и черных дыр), и нужно промоделировать ее эволюцию.

Каждое тело имеет массу, начальное положение и скорость. Гравитация вызывает перемещение и ускорение тел. Движение системы *n* тел имитируется пошагово с помощью дискретных отрезков времени. На каждом временном шаге вычисляются силы, действующие на каждое тело, и обновляются скорости и положения тел. Таким образом, имитация гравитационной задачи *n* тел имеет следующую структуру.

```
инициализировать тела;
for [time = start to finish by DT] {
  вычислить силы;
  переместить тела;
}
```

Значение DT (delta time) является временным шагом.

Согласно физике Ньютона *величина* силы гравитации между двумя телами *i* и *j* вычисляется по формуле

$$F = \frac{G m_i m_j}{r^2},$$

где m_i и m_j — массы тел, r — расстояние между ними. Гравитация является чрезвычайно слабым взаимодействием, поэтому значение гравитационной постоянной G — очень малое число $6{,}67 \cdot 10^{-11}$.

Предположим для простоты, что все тела расположены на одной плоскости, т.е. задача является двухмерной. Пусть $(p_i \cdot x, p_i \cdot y)$ и $(p_j \cdot x, p_j \cdot y)$ — положения двух тел. Евклидово расстояние r между ними определяется по формуле

$$\sqrt{(p_i \cdot x - p_j \cdot x)^2 + (p_i \cdot y - p_j \cdot y)^2}\ .$$

Если два тела подходят слишком близко друг к другу (близки к столкновению), r очень мало. Это приводит к значительной неустойчивости при вычислении сил, однако здесь данная проблема не рассматривается.

Направление силы, действующей на тело *i* со стороны тела *j*, задается единичным вектором, указывающим от *i* в сторону *j*, а силы воздействия тела *i* на тело *j* — противоположным вектором. Таким образом, величины сил, действующих между любыми двумя телами, равны, а направления противоположны. *Общая сила*, действующая на тело, есть сумма сил воздействия всех остальных тел. (Поскольку суммы векторов коммутативны, векторы сил можно складывать в любом порядке.)

Гравитационные силы, действующие на тело, вызывают его ускорение и перемещение. Отношение между силой, массой и ускорением описывается знаменитым уравнением

$$a = F/m,$$

т.е. ускорение тела *i* равно общей силе, действующей на тело, деленной на его массу. Если за малый интервал времени *dt* ускорение a_i тела *i* остается практически постоянным, то изменение скорости приблизительно равно

$$dv_i = a_i dt.$$

Изменение положения тела есть интеграл его скорости и ускорения на интервале времени *dt*, который приблизительно равен

$$dp_i = v_i dt + \frac{a_i}{2} dt^2 = (v_i + \frac{dv_i}{2}) dt\ .$$

Эта формула устроена по так называемой *скачкообразной* (leapfrog — чехарда) схеме, в которой одна половина изменения в положении тела обусловлена предыдущей скоростью, а другая — новой.

В листинге 11.6 представлена последовательная программа для решения задачи *n* тел. В программе использован простой алгоритм сложности $O(n^2)$, о чем свидетельствует цикл `for` процедуры `calculateForces`. Для каждого тела вычисляются силы, действующие на него со стороны других тел. При этом учитывается свойство симметрии сил, действующих между телами: каждая пара тел рассматривается только один раз (второй индекс `j` в главном цикле вычислений изменяется от `i+1` до `n`).

Листинг 11.6. Последовательная программа для решения задачи *n* тел

```
type point = rec(double x,y);
point p[1:n], v[1:n], f[1:n]; # положение,скорость, сила и
double m[1:n];                # масса тел
double G = 6.67e-11;
инициализировать положения, скорости, силы и массы;

#вычислить общую силу для всех пар тел
procedure calculateForces() {
  double distance, magnitude; point direction;
  for [i = 1 to n-1, j = i+1 to n] {
    distance = sqrt( (p[i].x - p[j].x)**2 +
                     (p[i].y - p[j].y)**2 );
    magnitude = (G*m[i]*m[j]) / distance**2;
    direction = point(p[j].x-p[i].x, p[j].y-p[i].y);
    f[i].x = f[i].x + magnitude*direction.x/distance;
    f[j].x = f[j].x - magnitude*direction.x/distance;
    f[i].y = f[i].y + magnitude*direction.y/distance;
    f[j].y = f[j].y - magnitude*direction.y/distance;
  }
}

# вычислить новые скорости и положения тел
procedure moveBodies() {
  point deltav;   # dv = f/m * DT
  point deltap;   # dp = (v + dv/2) * DT
  for [i = 1 to n] {
    deltav = point(f[i].x/m[i] * DT, f[i].y/m[i] * DT);
    deltap = point( (v[i].x + deltav.x/2) * DT,
                    (v[i].y + deltav.y/2) * DT );
    v[i].x = v[i].x + deltav.x;
    v[i].y = v[i].y + deltav.y;
    p[i].x = p[i].x + deltap.x;
    p[i].y = p[i].y + deltap.y;
    f[i].x = f[i].y = 0.0; #обнуление вектора силы
  }
}

# имитация с временным шагом DT
for [time = start to finish by DT] {
  calculateForces();
  moveBodies();
}
```

Программу в листинге 11.6 можно сделать более эффективной. Например, сократить мощность, заменив операции деления и возведения в степень умножением. Кроме того, некоторые выражения можно вычислять не многократно, а только один раз. Встраивание двух процедур немного поможет, но усложнит понимание программы.

11.2.2. Программа с разделяемыми переменными

Рассмотрим, как распараллелить программу для задачи *n* тел. Как всегда, вначале нужно решить, как распределить работу. Предположим, что есть PR процессоров, и, следовательно, будут использованы PR рабочих процессов. В реальных задачах n намного больше PR. Предположим, что n кратно PR.

Большая часть расчетов в последовательной программе проводится в цикле `for` процедуры `calculateForces`. В методе итераций Якоби работа просто распределялась по полосам, и на каждый рабочий процесс приходилось по n/PR строк сетки. В результате вычислительная нагрузка была сбалансированной, поскольку по методу итераций Якоби в каждой точке сетки выполняется один и тот же объем работы.

Для задачи *n* тел соответствующим разделением было бы назначение каждому рабочему процессу непрерывного блока тел: процесс 1 обрабатывает первые n/PR тел, 2 — следующие n/PR тел и т.д. Однако это привело бы к слишком несбалансированной нагрузке. Процессу 1 придется вычислить силы взаимодействия тела 1 со всеми остальными, затем — тела 2 со всеми, кроме тела 1, и т.д. С другой стороны, последнему процессу нужно вычислить только те силы, которые не были вычислены предыдущими процессами, т.е. лишь между последними n/PR телами.

Пусть для конкретного примера n равно 8, а PR — 2, т.е. используются два рабочих процесса. Назовем их черным (black — B) и белым (white — W). На рис. 11.4 показаны три способа назначения тел процессам. При блочном распределении первые четыре тела назначаются процессу B, последние четыре — W. Таким образом, число пар сил, вычисляемых B, равно 22 (7+6+5+4), а вычисляемых W, — 6 (3+2+1).

	1 2 3 4 5 6 7 8	
Схема		*Рабочая нагрузка*
блоки	B B B B W W W W	B = 22, W = 6
полосы	B W B W B W B W	B = 16, W = 12
обратные полосы	B W W B B W W B	B = 14, W = 14

Рис. 11.4. Схемы назначения тел рабочим процессам

Вычислительная нагрузка будет более сбалансированной, если назначать тела иначе: 1 — процессу B, 2 — W, 3 — B и т.д. Эта схема называется *распределением по полосам* по аналогии с полосами зебры. (Она называется еще *циклическим распределением*, поскольку схема повторяется.) Распределение по полосам приводит к следующей нагрузке процессов: 16 пар сил для B и 12 — для W.

Нагрузку можно еще больше сбалансировать, если распределить тела по схеме, похожей на ту, которая обычно используется при выборе команд в спортивных играх: назначим тело 1 процессу B, тела 2 и 3 — W, тела 4 и 5 — B и т.д. Эта схема называется *распределением по обратным полосам*, поскольку полосы каждый раз меняют порядок: черный, белый; белый, черный; черный, белый и т.д. Например, на рис. 11.4 нагрузка полностью сбалансирована — по 14 пар сил на каждый процесс.

Любая из приведенных выше стратегий распределения легко обобщается на любое число PR рабочих процессов. Блочная стратегия тривиальна. Для полос используется PR различных цветов. Первым цветом окрашивается тело 1, вторым — тело 2 и т.д. Цикл повторяется, пока все тела не будут окрашены. При использовании обратных полос порядок цветов меняется, когда начинается новый цикл, т.е. после окраски PR тел.

Для этой и любой другой задачи с похожей схемой индексов в главном вычислительном цикле распределение данных по схеме обратных полос дает наиболее сбалансированную вычислительную нагрузку. Однако схема полос программируется намного легче, чем схема обратных полос, и для больших значений n дает практически сбалансированную нагрузку. Поэтому в дальнейшем используем схему полос.

Следующие вопросы связаны с синхронизацией. Есть ли критические секции? Нужна ли барьерная синхронизация? Ответ на оба вопроса утвердительный.

Сначала рассмотрим процедуру `calculateForces`. В цикле `for` рассматривается каждая пара тел `i` и `j`. Пока один процесс вычисляет силы между телами `i` и `j`, другой вычисляет силы между телами `i'` и `j'`. Некоторые из этих номеров могут быть равными: например, `j` в одном процессе может совпадать с `i'` в другом. Следовательно, процессы могут мешать друг другу при обновлении вектора сил `f`. Таким образом, четыре оператора присваивания, обновляющих `f`, образуют код критической секции.

Один из способов защитить критическую секцию — использовать одну переменную блокировки. Однако это неэффективно, поскольку критическая секция выполняется много раз каждым процессом. В другом предельном случае можно использовать массив переменных блокировки, по одной на тело. Это почти устраняет конфликты при блокировке, но за счет использования гораздо большего объема памяти. Промежуточный способ — использовать одну переменную блокировки на каждый блок из `B` тел, но и тут возникают конфликты блокировки.

Для хорошей производительности лучше всего вообще устранить накладные расходы на блокировку, избавившись от критических секций (если возможно). Это можно сделать двумя способами. При первом каждый рабочий процесс берет на себя полную ответственность за назначенные ему тела, т.е. процесс, отвечающий за тело `i`, вычисляет все силы, действующие на тело `i`, но *не* обновляет вектор сил для любого другого тела `j`. Взамен, процесс, отвечающий за тело `j`, вычисляет все силы, действующие на него, в том числе и со стороны тела `i`. Такой способ можно запрограммировать, используя индексную переменную `j` в `for` цикле `calculateForces` со значениями в диапазоне от `1` до `n`, а не от `i+1` до `n`, и исключая присваивания `f[j]`. Однако при этом не используется симметричность сил между двумя телами, так что все силы вычисляются дважды.

Второй способ устранения критических секций — вектор сил заменить матрицей сил, каждая строка которой соответствует одному процессу. Вычисляя силу между телами `i` и `j`, процесс `w` обновляет два элемента *своей* строки в матрице сил. А когда нужно будет вычислить новые положение и скорость тела, он сначала сложит все силы, действующие на данное тело и вычисленные ранее всеми остальными процессами.

Оба способа устранения критических секций используют *дублирование*. В первом методе дублируются вычисления, во втором — данные. Это еще один пример пространственно-временного противоречия. Поскольку целью параллельной программы обычно является уменьшение времени выполнения, то лучший выбор — использовать как можно больше пространства (разумеется, в пределах доступной памяти).

Осталось рассмотреть еще один важный момент — нужны ли барьеры, и если да, то где именно. Значения новых скоростей и положений, вычисляемых в `moveBodies`, зависят от сил, вычисленных в `calculateForces`. Поэтому, пока не будут вычислены все силы, нельзя перемещать тела. Аналогично силы зависят от положений и скоростей, поэтому нельзя пересчитывать силы, пока не перемещены тела. Таким образом, после каждого вызова процедур `calculateForces` и `moveBodies` нужна барьерная синхронизация.

Итак, здесь описаны три способа назначения тел процессам. Использование схемы полос приводит к вполне сбалансированной и легко программируемой вычислительной нагрузке. Рассмотрена проблема критической секции, показано, как применять блокировку и избегать ее. Наиболее эффективен следующий подход: исключить критические секции, чтобы каждый процесс обновлял свой собственный вектор сил. Наконец, независимо от распределения рабочей нагрузки и управления критическими секциями нужны барьеры после вычисления сил и перемещения тел.

Все представленные решения включены в код в листинге 11.7. В основном структура программы совпадает со структурой последовательной программы (см. листинг 11.6). Для реализации барьерной синхронизации добавлена третья процедура `barrier`. Единый главный цикл имитации заменен имитацией на `PR` рабочих процессорах, и каждый из них выполняет цикл имитации. Процедура `barrier` вызывается после как вычисления сил, так и перемещения тел. Во всех вызовах процедур содержится аргумент `w`, который указывает на вызывающий процесс.

Листинг 11.7. Программа для задачи *n* тел с разделяемыми переменными

```
type point=rec(double x,y); double G=6.67e-11;
point p[1:n], v[1:n], f[1:PR,1:n]; # положения,
double m[1:n];                     # скорости, силы и массы тел
инициализировать положения, скорости, силы и массы;

procedure barrier(int w) {
  # эффективный алгоритм барьера из раздела 3.4
}

# вычислить силы, действующие на тела, назначенные процессу w
procedure calculateForces(int w) {
  double distance, magnitude; point direction;
 for [i = w to n by PR, j = i+1 to n] {
    distance = sqrt( (p[i].x - p[j].x)**2 +
                     (p[i].y - p[j].y)**2 );
    magnitude = (G*m[i]*m[j]) / distance**2;
    direction = point( p[j].x-p[i].x, p[j].y-p[i].y );
    f[w,i].x = f[w,i].x + magnitude*direction.x/distance;
    f[w,j].x = f[w,j].x - magnitude*direction.x/distance;
    f[w,i].y = f[w,i].y + magnitude*direction.y/distance;
    f[w,j].y = f[w,j].y - magnitude*direction.y/distance;
  }
}

# переместить тела, назначенные процессу w
procedure moveBodies(int w) {
  point deltav;    # dv = f/m * DT
  point deltap;    # dp = (v + dv/2) * DT
  point force = (0.0, 0.0);
  for [i = w to n by PR] {
    #сложить силы, действующие на тело i, и обнулить f[*,i]
    for [k = 1 to PR] {
      force.x +=f[k,i].x; f[k,i].x = 0.0;
      force.y +=f[k,i].y; f[k,i].y = 0.0;
    }
    deltav = point(force.x/m[i] * DT,
force.y/m[i] * DT);
    deltap = point( (v[i].x + deltav.x/2)* DT,
                    (v[i].y + deltav.y/2)*DT );
    v[i].x = v[i].x + deltav.x;
    v[i].y = v[i].y + deltav.y;
    p[i].x = p[i].x + deltap.x;
    p[i].y = p[i].y + deltap.y;
    force.x = force.y = 0.0;
  }
}

process Worker[w = 1 to PR] {
  # имитировать с временным шагом DT
  for [time = start to finish by DT] {
    calculateForces(w);
    barrier(w);
    moveBodies(w);
    barrier(w);
  }
}
```

Тела процедур, вычисляющих силы и перемещающих тела, изменены так, как описано выше. Наконец, главные циклы в этих процедурах изменены так, что приращение по i равно PR, а не 1 (этого достаточно, чтобы присваивать тела рабочим по схеме полос). Код в листинге 11.7 можно сделать более эффективным, оптимизировав его, как и последовательную программу.

11.2.3. Программы с передачей сообщений

Рассмотрим, как решить задачу *n* тел, используя передачу сообщений. Нам нужно построить метод распределения вычислений между рабочими процессами, чтобы рабочая нагрузка была сбалансированной. Необходимо также минимизировать накладные расходы на взаимодействие или хотя бы снизить их по сравнению с объемом полезной работы. Эти проблемы непросты, поскольку в задаче *n* тел вычисляются силы для всех пар тел, и, следовательно, каждому процессу приходится взаимодействовать со всеми остальными.

Вначале полезно разобраться, можно ли использовать парадигмы взаимодействия между процессами, описанные в начале главы 9, — "управляющий-рабочие" (распределенный портфель задач), алгоритм пульсации и конвейер. Поскольку задачи достаточно хорошо определены, а нагрузку сбалансировать сложно, подходит парадигма "управляющий-рабочие". Можно также применить алгоритм пульсации, используя распределение тел и обмен данными между процессами. Однако явно не достаточно того, что каждый процесс обменивается информацией только с одним или двумя соседями. Наконец, можно использовать конвейер, в котором тела переходят от процесса к процессу. Ниже приведены решения задачи *n* тел с помощью каждой из этих парадигм, затем обсуждаются их преимущества и недостатки.

Программа типа "управляющий-рабочие"

Применим парадигму "управляющий-рабочие" (см. раздел 9.1). Управляющий обслуживает портфель задач; рабочие многократно получают задачу, выполняют ее и возвращают результат управляющему. В задаче *n* тел есть две фазы. В задачах первой фазы вычисляются силы между всеми парами тел, в задачах второй — перемещаются тела. Вычисление сил является основной частью работы, которая может оказаться несбалансированной. Поэтому есть смысл при вычислении сил использовать динамические задачи, а и при перемещении тел — статические (по одной на рабочего).

Предположим, что у нас есть PR процессоров и используются PR рабочих процессов. (Управляющий процесс работает на том же процессоре, что и один из рабочих.) Предположим для простоты, что n кратно PR. При использовании парадигмы "управляющий-рабочие" для сбалансированности нагрузки нужно, чтобы задач было хотя бы вдвое больше, чем рабочих процессов. Однако очень большое число задач или рабочих процессов нежелательно, поскольку рабочие процессы потратят слишком много времени на взаимодействие с менеджером. Один из способов распределить вычисления сил, действующих на n тел, состоит в следующем.

> Множество тел разделяется на PR блоков размером n/PR. Первый блок содержит первые n/PR тел, второй — следующие n/PR тел и т.д.
>
> Образуются пары (i,j) для всех возможных комбинаций номеров блоков. Таких пар будет PR*(PR+1)/2 (сумма целых чисел от 1 до PR).
>
> Пусть каждая пара представляет задачу, например, для пары (i,j) — вычислить силы, действующие между телами в блоках i и j.

В качестве конкретного примера предположим, что PR равно 4. Тогда десять задач представлены парами

(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4).

этого рабочему нужны данные о положениях, скоростях и массах тел в одном или двух блоках. Управляющий мог бы передавать эти данные вместе с задачей, но можно значительно сократить длину сообщений, если у каждого рабочего процесса будет своя собственная копия данных о всех телах. Аналогично можно избавиться и от необходимости отправлять результаты управляющему, если каждый рабочий процесс отслеживает силы, которые он вычислил для каждого тела.

После вычисления всех сил, т.е. когда портфель задач пуст, рабочим процессам нужно обменяться силами и затем переместить тела. Простейшее статическое назначение работы в фазе перемещения тел состоит в том, чтобы каждый рабочий перемещал тела в одном блоке: рабочий процесс 1 перемещает тела в блоке 1, рабочий 2 — в блоке 2 и т. д. Следовательно, каждому рабочему w нужны векторы сил для тел блока w.

Листинг 11.8. Программа типа "управляющий-рабочие" для задачи *n* тел

```
chan getTask(int worker), task[1:PR](int block1, block2);
chan bodies[1:PR](int worker; point pos[*], vel[*]);
chan forces[1:PR](point force[*]);

process Manager {
  декларации и инициализация локальных переменных;
  for [time = start to finish by DT] {
    инициализация портфеля задач;
    for [i = 1 to numTasks+PR ] {
      receive getTask(worker);
      выбрать следующую задачу; если портфель пуст, сигнализировать (0, 0);
      send task[worker] (block1, block2);
    }
  }
}

process Worker[w = 1 to PR] {
  point p[1:n], v[1:n], f[1:n]; # положения, скорости
  double m[1:n]; # силы и массы тел
  декларации остальных локальных переменных, например tf[1:n], tp[1:n], tv[1:n];
  инициализация всех локальных переменных;
  for [time = start to finish by DT] {
    while (true) {
      send getTask(w); receive task[w](block1, block2);
      if (block1 == 0) break; # портфель пуст
      вычислить силы между телами block1 и block2;
    }
    for [i = 1 to PR st i != w]  # обмен силами
      send forces[i](f[*]);
    for [i = 1 to PR st i != w]
      receive forces[w](tf[*]);
      добавить значения tf к значениям в f;
    }
    обновить p и v в своем блоке тел;
    for [i = 1 to PR st i !=w] # обмен телами
      send bodies[i](w, p[*], v[*]);
    for [i = 1 to PR st i !=w]
      receive bodies[w](worker, tp[*], tv[*]);
      переместить тела процесса worker из tp и tv в p и v;
    }
    реинициализировать f нулями;
  }
}
```

Собирать и распределять силы, вычисленные каждым рабочим процессом, мог бы управляющий, но эффективность повысится, если каждый рабочий будет отправлять силы для блока тел непосредственно тому рабочему, который будет перемещать эти тела. Если же доступны глобальные примитивы взаимодействия (как в библиотеке MPI), возможен другой вариант: рабочие используют их для рассылки и удаления (добавления) значений в векторах сил. После того как все рабочие переместят тела в своих блоках, им нужно обменяться новыми положениями и скоростями тел. Для этого можно использовать сообщения "от точки к точке" или глобальное взаимодействие.

Листинг 11.8 содержит эскиз кода для программы типа "управляющий-рабочие". Внешний цикл в каждом процессе выполняется один раз на каждом временном шаге имитации. Внутренний цикл в управляющем процессе выполняется `numTasks+PR` раз, где `numTasks` — число задач в портфеле. На последних `PR` итерациях портфель пуст, и управляющий отправляет пару (0, 0) как сигнал о том, что портфель пуст. Получая этот сигнал, каждый из `PR` процессов выходит из цикла вычисления сил.

Программа пульсации

Аналогом алгоритма с разделяемыми переменными является алгоритм пульсации, использующий передачу сообщений (раздел 9.2); вычислительные фазы в нем чередуются с барьерами. Программа с разделяемыми переменными (см. листинг 11.7) имеет соответствующую структуру, поэтому, чтобы использовать передачу сообщений, программу можно изменить: 1) каждому рабочему процессу назначается подмножество тел; 2) каждый рабочий сначала вычисляет силы в своем подмножестве, а затем обменивается силами с другими рабочими; 3) каждый рабочий перемещает свои тела; 4) рабочие обмениваются новыми положениями и скоростями тел. При имитации эти действия повторяются на каждом шаге по времени.

Назначать тела рабочим процессам можно по любой схеме распределения (см. рис. 11.4): по блокам, полосам или обратным полосам. Распределение по полосам или обратным полосам дает гораздо более сбалансированную вычислительную нагрузку, чем по блокам фиксированного размера. Однако каждому рабочему процессу придется иметь свою собственную копию векторов положений, скоростей, сил и масс. Кроме того, после каждой фазы необходим обмен целыми векторами со всеми остальными рабочими. Все это приводит к большому числу длинных сообщений.

Если рабочим назначать блоки тел, то будет использоваться приблизительно вдвое меньше сообщений, причем они будут короче. (Рабочим процессам также нужно меньше локальной памяти.) Чтобы понять, почему так происходит, рассмотрим пример из предыдущего раздела, в котором четыре процесса и десять задач. При назначении по блокам процесс 1 обрабатывает первые четыре задачи: (1, 1), (1, 2), (1, 3) и (1, 4). Он вычисляет все силы, связанные с телами блока 1. Для этого ему необходимы положения и скорости тел из блоков 2, 3 и 4, и в эти же блоки ему нужно возвратить силы. Однако процесс 1 никогда не передает другим процессам положения и скорости своих тел, и ему не нужна информация о силах от любого другого процесса. С другой стороны, процесс 4 отвечает только за задачу (4, 4). Ему не нужны положения или скорости остальных тел, и он не должен отправлять силы остальным рабочим.

Если все блоки одного размера, то рабочая нагрузка оказывается несбалансированной; затраты времени при этом могут даже превысить выигрыш от отправки меньшего числа сообщений. Но нет и причин, по которым все блоки должны быть одного размера, как в программе с разделяемыми переменными. Более того, тела можно переместить за линейное время, поэтому использование блоков разных размеров приведет к относительно небольшому дисбалансу нагрузки в фазе перемещения тел.

В листинге 11.9 представлена схема кода для программы пульсации, в которой рабочим процессам назначаются блоки разных размеров. Как показано, части кода, связанные с передачей сообщений, не симметричны. Разные рабочие отправляют неодинаковое число сообщений в различные моменты времени. Однако этот недостаток превращен в преимущество с помощью совмещения взаимодействий и вычислений во времени. Каждый рабочий процесс отправляет сообщения и затем до получения и обработки сообщений выполняет локальные вычисления.

Листинг 11.9. Программа пульсации для задачи *n* тел

```
chan bodies[1:PR](int worker; point pos[*], vel[*]);
chan forces[1:PR](point force[*]);

process Worker[w = 1 to PR] {
  int blockSize = размер своего блока тел;
  int tempSize = максимальное число других тел в сообщениях;
  point p[1:blockSize], v[1:blockSize], f[1:blockSize];
  point tp[1:tempSize], tv[1:tempSize], tf[1:tempSize];
  double m[1:n];
  декларации остальных локальных переменных;
  инициализация всех локальных переменных;
  for [time = start to finish by DT] {
    #отправить свои тела рабочим с меньшими номерами
    for [i = 1 to w-1]
      send bodies[i](w, p[*], v[*]);
    вычислить силы f для своего блока тел;
    #получить тела от рабочих с большими
    # номерами и отправить им силы
    for [i = w+1 to PR] { #получить тела от других
      receive bodies[w](other, tp[*], tv[*]);
      вычислить силы между своим и другим блоками;
    send forces[other](tf[*]);
    }
    # получить силы от рабочих с меньшими номерами
    for [i = 1 to w-1] {
      receive forces[w](tf[*]);
      добавить силы из tf к силам в f;
    }
    обновить p и v для своих тел;
    реинициалировать f нулями;
  }
}
```

Программа с конвейером

Рассмотрим решение задачи *n* тел с помощью конвейерного алгоритма (раздел 9.3). Напомним, что в конвейере информация двигается в одном направлении от процесса к процессу. Конвейер бывает открытым, круговым и замкнутым (см. рис. 9.1). Здесь нужна информация о телах, циркулирующая между рабочими процессами, поэтому следует использовать или круговой, или замкнутый конвейер. Управляющий процесс не нужен (за исключением, возможно, лишь инициализации рабочих процессов и сбора конечных результатов), поэтому достаточно кругового конвейера.

Рассмотрим идею использования кругового конвейера. Предположим, что каждому рабочему процессу назначен блок тел и каждый рабочий вычисляет силы, действующие только на тела в своем блоке. (Таким образом, сейчас выполняются лишние вычисления; позже мы используем симметрию сил между телами.) Каждому рабочему для вычисления сил, создаваемых "чужими" телами, нужна информация о них. Таким образом, достаточно, чтобы блоки тел циркулировали между рабочими процессами. Это можно сделать следующим образом.

```
Отправить p и v своего блока тел следующему рабочему процессу;
вычислить силы, действующие между телами своего блока;
for [i = 1 to PR-1] {
  получить p и v блока тел от предыдущего рабочего процесса;
  отправить этот блок тел следующему рабочему процессу;
```

```
        вычислить силы, с которыми тела нового блока действуют на тела своего блока;
    }
    получить обратно свой блок тел;
    переместить тела;
    реинициализировать силы, действующие на свои тела, нулями;
```

Каждый рабочий процесс выполняет данный код на каждом временном шаге имитации. (Последняя отправка и получение не обязательны в приведенном коде, однако они понадобятся позже.)

При описанном подходе выполняется вдвое больше вычислений сил, чем необходимо. Следовательно, нам желательно рассматривать каждую пару тел только один раз и пропускать силы, уже вычисленные для какой-либо группы тел. Чтобы сбалансировать вычислительную нагрузку, нужно или назначать разные количества тел каждому процессу (как в программе пульсации), или назначать тела по полосам или обратным полосам, как в программе с разделяемыми переменными. Применим одну из схем назначения тел по полосам, которая даст наиболее сбалансированную нагрузку.

В листинге 11.10 приведен код кругового конвейера, использующий схему присваивания тел по полосам. Каждое сообщение содержит положения, скорости и силы, уже вычисленные для подмножества тел. В коде не показаны все подробности учета, необходимые, чтобы точно отслеживать, какие тела принадлежат каждому подмножеству; однако это можно определить по идентификатору владельца подмножества. Каждое подмножество тел циркулирует по конвейеру, пока не вернется к своему владельцу, который затем переместит эти тела.

Листинг 11.10. Программа с конвейером для задачи *n* тел

```
chan bodies[1:PR](int owner; point p[*], v[*], f[*]);

process Worker[w = 1 to PR] {
  int owner, setSize = n/PR, next = w%PR + 1;
  point p[1:setSize], v[1:setSize], f[1:setSize];
  point tp[1:setSize], tv[1:setSize], tf[1:setSize];
  double m[1:n];
  декларации других локальных переменных;
  инициализация своего блока тел и других переменных;
  for[time = start to finish by DT] {
    send bodies[next](w, p[*], v[*], f[*]);
    вычислить силы, действующие между телами своего блока;
    for [i = 1 to PR-1] {
      receive bodies[w](owner, tp[*], tv[*], tf[*]);
      вычислить силы между своими и новыми телами;
      send bodies[w](owner, tp[*], tv[*], tf[*]);
    }
    # получить свои тела (owner будет равен w)
    receive bodies[w](owner, tp[*], tv[*], tf[*]);
    добавить силы из tf к силам в f;
    обновить p и v для своего подмножества тел;
    реинициализировать силы, действующие на свои тела, нулями;
  }
}
```

Сравнение программ

Приведенные три программы с передачей сообщений отличаются друг от друга по нескольким параметрам: легкость программирования, сбалансированность нагрузки, число сообщений и объем локальных данных. Легкость программирования субъективна; она зависит от того, насколько программист знаком с каждым из стилей. Тем не менее, если сравнить длину и сложность программ, простейшей окажется программа с конвейером. Она короче

других, отчасти потому, что в ней используется наиболее регулярная схема взаимодействия. Программа типа "управляющий-рабочие" также весьма проста, хотя ей нужен дополнительный процесс. Программа пульсации сложнее других (хотя и не намного), поскольку в ней используются блоки тел разных размеров и асимметричная схема взаимодействия.

Рабочая нагрузка у всех трех программ будет достаточно сбалансированной. Программа типа "управляющий-рабочие" динамически распределяет работу, поэтому нагрузка здесь почти сбалансирована, даже если процессоры имеют разные скорости или некоторые из них одновременно выполняют другие программы. При выполнении на специализированной однородной архитектуре у конвейерной программы с назначением по полосам рабочая нагрузка будет сбалансирована достаточно, а с назначением по обратным полосам — почти идеально. Программа пульсации использует блоки разных размеров, поэтому нагрузка окажется не идеально, но все-таки почти сбалансированной; кроме того, асимметричная схема взаимодействия и перекрытие взаимодействия и вычислений во времени отчасти скроет несбалансированность вычисления сил.

Во всех программах на каждом шаге по времени отправляются (и получаются) $O(PR2)$ сообщений. Однако действительные количества сообщений отличаются, как и их размеры (табл. 11.1). Алгоритм пульсации дает наименьшее число сообщений. Конвейер — на PR сообщений больше, но они самые короткие. Больше всего сообщений в программе типа "управляющий-рабочие", однако здесь обмен значениями между рабочими процессами можно реализовать с помощью операций группового взаимодействия.

Таблица 11.1. Сравнение программ с передачей сообщений для задачи *n* тел

Программа/Фаза	Количество сообщений	Число тел на сообщение
Управляющий-рабочие		
Получение задачи	`2 * (numTasks+PR)`	2
Обмен силами	`PR * (PR-1)`	n
Обмен телами	`PR * (PR-1)`	`2 * n`
Пульсация		
Обмен телами	`PR * (PR-1) / 2`	`2 * n`
Обмен силами	`PR * (PR-1) / 2`	n
Конвейер		
Циркуляция тел	`PR * PR`	`3 * n * PR`

Наконец, программы отличаются объемом локальной памяти каждого рабочего процесса. В программе типа "управляющий-рабочие" каждый рабочий процесс имеет копию данных о всех телах; ему также нужна временная память для сообщений, число которых может достигать 2n. В программе пульсации каждому процессу нужна память для своего подмножества тел и временная память для наибольшего из блоков, которые он может получить от других. В конвейерной программе каждому рабочему процессу нужна память для своего подмножества тел и рабочая память для еще одного подмножества, размер которого — n/PR. Следовательно, конвейерной программе нужно меньше всего памяти на один рабочий процесс.

Подведем итог: для задачи *n* тел наиболее привлекательной выглядит программа с конвейером. Она сравнительно легко пишется, дает почти сбалансированную нагрузку, требует наименьшего объема локальной памяти и почти минимального числа сообщений. Кроме того, конвейерная программа будет эффективнее работать на некоторых типах коммуникационных сетей, поскольку каждый рабочий процесс взаимодействует только с двумя соседями. Однако различия между программами относительно невелики, поэтому приемлема любая из них. Оставим читателю интересную задачу реализации всех трех программ и экспериментального сравнения их производительности.

11.2.4. Приближенные методы

В задаче n тел доминируют вычисления сил. В реальных приложениях для очень больших значений n на вычисление сил приходится более 95 процентов всего времени. Поскольку каждое тело взаимодействует со всеми остальными, на каждом шаге по времени выполняется $O(n^2)$ вычислений сил. Однако величина гравитационной силы между двумя телами обратно пропорциональна квадрату расстояния между ними. Поэтому, если два тела находятся далеко друг от друга, то силой их притяжения можно пренебречь.

Рассмотрим конкретный пример: на движение Земли влияют другие тела Солнечной системы, в большей степени Солнце и Луна и в меньшей — остальные планеты. Гравитационная сила воздействия на Солнечную систему со стороны других тел пренебрежимо мала, но оказывается достаточно большой, чтобы вызвать движение Солнечной системы по галактике Млечного пути.

Ньютон доказал, что группу тел можно рассматривать как одно тело по отношению к другим, удаленным, телам. Например, группу тел B можно заменить одним телом b, которое имеет массу, равную сумме масс всех тел в B, и расположено в центре масс B. Тогда силу между отдаленным телом i и всеми телами B можно заменить силой между i и b.

Ньютоновский подход приводит к приближенным методам имитации n тел. Первым распространенным методом был *алгоритм Барнса–Хата* (Barnes–Hut), но теперь чаще используется более эффективный *быстрый метод мультиполей*. Оба метода являются примерами так называемых *древесных кодов*, поскольку лежащая в их основе структура данных представляет собой дерево тел. Время работы алгоритма Барнса–Хата равно $O(n \log n)$; для однородных распределений тел время работы быстрого метода мультиполей равно $O(n)$. Ниже кратко описана работа этих методов. Подробности можно найти в работах, указанных в исторической справке в конце данной главы.

Основой иерархических методов является древовидное представление физического пространства. Корень дерева представляет *клетку*, которая содержит все тела. Дерево строится рекурсивно с помощью разбиения клеток. В двухмерном варианте сначала корневую клетку делят на четыре одинаковые клетки, а затем дробят полученные клетки. Клетка делится, если количество тел в ней больше некоторого фиксированного числа. Полученное таким образом дерево называется *деревом квадрантов* (quadtree), поскольку у каждого нелистового узла по четыре сына. Форма дерева соответствует распределению тел, поскольку дерево имеет больше уровней в тех областях, где больше тел. (В трехмерном пространстве родительские клетки делятся на восемь меньших, и структура данных называется *деревом октантов*.)

На рис. 11.5 проиллюстрирован процесс деления двухмерного пространства на клетки и соответствующее дерево квадрантов. Каждая группа из четырех клеток одинакового размера нумеруется по часовой стрелке, начиная с верхней левой и заканчивая нижней левой, и представлена деревом в том же порядке. Каждый узел-лист представляет клетку либо пустую, либо с одним телом.

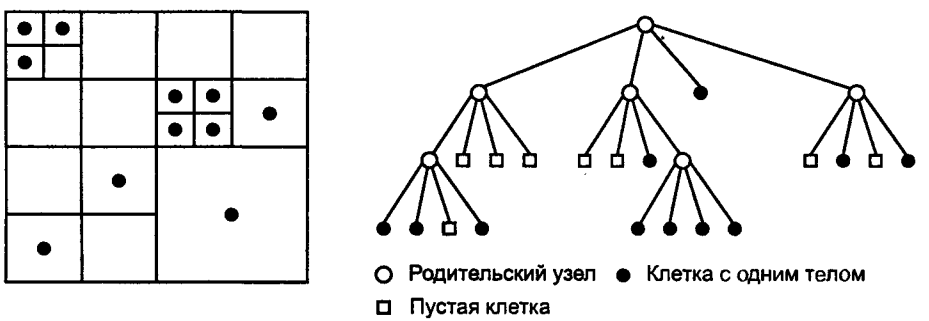


Рис. 11.5. Клетки и дерево квадрантов в двухмерном пространстве

Алгоритм Барнса–Хата

Алгоритм сложности n^2 на каждом шаге по времени имеет две фазы вычислений — вычисление сил и перемещение тел. В алгоритме Барнса–Хата есть еще две фазы предварительной обработки. Таким образом, на каждом временном шаге вычисления проводятся в четыре этапа.

1. Строится дерево квадрантов, представляющее текущее распределение тел.
2. Вычисляется полная масса и центр масс каждой клетки. Полученные значения записываются в соответствующем узле дерева квадрантов. Это делается с помощью прохода от листовых узлов к корню.
3. С использованием дерева вычисляются силы, связанные с каждым телом. Начиная с корня дерева, для каждого посещаемого узла рекурсивно выполняют следующую проверку. Если центр масс клетки "достаточно удален" от тела, то клеткой аппроксимируется все соответствующее ей поддерево; иначе обходятся все четыре подклетки. Чтобы определить, "достаточно ли удалена" клетка, используется параметр, заданный пользователем.
4. Тела перемещаются, как и раньше.

На каждом временном шаге дерево перестраивается, поскольку тела перемещаются, и, следовательно, их распределение изменяется. Однако перемещения, в основном, малы, поэтому дерево можно перестраивать, если только какое-нибудь тело перемещается за пределы своей клетки. В любом случае центры масс всех клеток нужно пересчитывать на каждом временном шаге.

Число узлов в дереве квадрантов равно $O(n \log n)$, поэтому фазы 1 и 2 имеют такую же временную сложность. Вычисление сил также имеет сложность $O(n \log n)$, поскольку в дереве обходятся только клетки, расположенные близко к телу. Перемещение тел является, как обычно, самой быстрой фазой с временной сложностью $O(n)$.

В алгоритме Барнса–Хата не учитывается свойство симметрии сил между телами, т.е. в фазе вычисления сил каждое тело рассматривается отдельно. (Учет получится слишком громоздким, если пытаться отслеживать все уже вычисленные пары тел.) Тем не менее алгоритм эффективен, поскольку его временная сложность меньше $O(n^2)$.

Алгоритм Барнса–Хата сложнее эффективно распараллелить, чем основной алгоритм, из-за следующих дополнительных проблем: 1) пространственное распределение тел в общем случае неоднородно, и, соответственно, дерево не сбалансировано; 2) дерево приходится обновлять на каждом временном шаге, поскольку тела перемещаются; 3) дерево является связанной структурой данных, которую гораздо сложнее распределить (и использовать), чем простые массивы. Эти проблемы затрудняют балансировку рабочей нагрузки и ослабление требований памяти. Например, процессорам нужно было бы назначить поровну тел, но это не будет соответствовать каким бы то ни было регулярным разбиениям пространства на клетки. Более того, процессорам нужно работать вместе, чтобы построить дерево, и при вычислении сил им может понадобиться все дерево. В результате многолетних исследований по имитации задачи n тел были разработаны эффективные методы ее параллельной реализации (см. историческую справку в конце главы). В настоящее время можно имитировать эволюцию систем, содержащих сотни миллионов тел, и на тысячах процессоров получать существенное ускорение.

Быстрый метод мультиполей

Быстрый метод мультиполей (Fast Multipole Method — FMM) является усовершенствованием метода Барнса–Хата и требует намного меньше вычислений сил. Метод Барнса–Хата рассматривает только взаимодействия типа тело-тело и тело-клетка. В FMM также вычисляются взаимодействия типа клетка-клетка. Если внутренняя клетка дерева квадрантов находится достаточно далеко от другой внутренней клетки, то в FMM вычисляются силы между двумя клетками, и полученный результат переносится на всех потомков обеих клеток.

Второе основное отличие FMM от метода Барнса–Хата состоит в способе, по которому методы управляют точностью аппроксимации сил. Метод Барнса–Хата представляет каждую клетку точкой, размещенной в центре масс клетки, и для аппроксимации силы учитывает

расстояние между телом и данной точкой. В FMM распределение массы в клетке представляется с помощью ряда разложений и считается, что две клетки достаточно удалены друг от друга, если расстояние между ними намного превосходит длину большей из их сторон. Таким образом, аппроксимации в FMM используются намного чаще, чем в методе Барнса–Хата, но это компенсируется более точным описанием распределения массы внутри клетки.

Фазы на временном шаге в FMM такие же, как и в методе Барнса–Хата, но фазы 2 и 3 реализуются иначе. Благодаря вычислению взаимодействий типа клетка-клетка и аппроксимации гораздо большего числа сил, временная сложность FMM равна $O(n)$ для однородных распределений тел в пространстве. FMM трудно распараллелить, но применение такой же техники, как в методе Барнса–Хата, все-таки позволяет сделать это достаточно эффективно.

11.3. Матричные вычисления

Сеточные и точечные вычисления являются фундаментальными в научных вычислениях. Третий основной вид вычислений — матричные. Умножение плотных и разреженных матриц уже рассматривалось. В данном разделе представлено использование матриц для решения систем линейных уравнений. Задачи такого типа составляют основу многих научных и инженерных приложений, а также задач экономического моделирования и многих других. (В действительности уравнение Лапласа, рассмотренное в разделе 11.1, можно заменить большой системой уравнений. Однако эта система получится очень разреженной, поэтому уравнение Лапласа обычно решают с помощью итерационных сеточных вычислений.)

Вначале рассмотрен *метод исключений Гаусса*. Затем описан более общий метод, который называется *LU-разложением*, и для него построена последовательная программа. Наконец разработаны параллельные программы для LU-разложения с разделяемыми переменными и с передачей сообщений. В упражнениях представлены другие матричные вычисления, в том числе обращение матриц.

11.3.1. Метод исключений Гаусса

Рассмотрим систему из трех уравнений с тремя неизвестными.

```
 a + b + c =  6
2a - b + c =  3
-a + b - c = -2.
```

Стандартный способ решения данной системы с неизвестными a, b и c — переписать одно из уравнений относительно какой-либо переменной, скажем, a, и полученное для нее выражение подставить в два других уравнения. Получим два новых уравнения с двумя неизвестными. Затем выполним с ними те же действия — перепишем одно уравнение относительно одной из переменных, скажем, b, и подставим полученное для b выражение во второе уравнение. Решим полученное уравнение относительно c, затем найдем b и, наконец, a.

Метод (процедура) исключений Гаусса является систематическим методом решения систем линейных уравнений любых размеров. Для указанных выше трех уравнений он работает следующим образом. Первый шаг: умножим первое уравнение на 2 и вычтем его из второго. Из второго уравнения исключается a. Второй шаг: умножим первое уравнение на -1 и вычтем его из третьего (т.е. сложим первое и третье уравнения); a исключается из третьего уравнения. Итак, получены уравнения

```
a + b + c =  6
   -3b - c = -9
    2b     =  4.
```

В данном примере из третьего уравнения одновременно исключается и c, но так бывает не всегда.

Повторим описанные выше действия для двух последних уравнений. Умножим второе уравнение на 2/3 и сложим его с третьим. В третьем уравнении исчезает b (и появляется c). Получается система уравнений

```
a + b +c  =   6
  -3b - c =  -9
    -2/3c =  -2.
```

Описанные выше действия систематически исключают одну переменную из оставшихся уравнений и называются *прямым ходом* (фазой исключений). Во второй фазе выполняется *обратный ход*, в котором неизвестные находятся в обратном порядке, начиная с последнего уравнения и заканчивая первым. В нашем примере из последнего уравнения получим `c = 3`. Затем во второе уравнение вместо c подставим 3 и найдем значение b: `b = 2`. Наконец подставим полученные для b и c значения в первое уравнение и найдем a: `a = 1`. Итак, решение данной системы уравнений — `a = 1, b = 2` и `c = 3`.

Решение системы линейных уравнений эквивалентно решению матричного уравнения `Ax = b`, где A — квадратная матрица коэффициентов, b — вектор-столбец правых частей уравнений, а x — вектор-столбец неизвестных. Строка с номером i матрицы A содержит коэффициенты для неизвестных i-го уравнения, а i-й элемент в столбце b — значение правой части i-го уравнения.

Метод исключений Гаусса реализуется серией преобразований матрицы A и вектора b. Матрица A приводится к *верхней треугольной матрице*, у которой все элементы, расположенные ниже главной диагонали, равны нулю. В нашем примере начальное значение матрицы A таково:

```
 1  1  1
 2 -1  1
-1  1 -1.
```

Соответствующие значения в b — `(6, 3, -2)`. Прямой ход начинается с левого столбца и преобразует A и b следующим образом. Первый шаг: вычисляем множитель `A[2,1]/A[1,1]`, умножаем на него первую строку матрицы A и первый элемент b; полученные строку и элемент вычитаем из второй строки A и второго элемента b. Второй шаг: вычисляем множитель `A[3,1]/A[1,1]`, умножаем на него первую строку матрицы A и первый элемент b, и вычитаем их из третьей строки A и третьего элемента b. После этих двух шагов в первом столбце A будут нули во второй и третьей строках. Последний шаг прямого хода в нашем несложном примере — вычисляем множитель `A[3,2]/A[2,2]`, умножаем на него вторую строку A и второй элемент b и вычитаем их из третьей строки A и третьего элемента b. В итоге матрица A примет вид

```
1   1   1
0  -3  -1
0   0 -2/3.
```

Соответствующие значения для b — `(6, -9, -2)`.

Метод исключений Гаусса можно использовать при решении многих систем *n* уравнений с *n* неизвестными.[23] Однако на каждом шаге прямого хода вычисляются множители вида `A[k,i]/A[i,i]`. Элемент `A[i,i]` называется *ведущим элементом* столбца i. Если он равен нулю, то получится "деление на ноль". Кроме того, если ведущий элемент слишком мал, то множитель будет слишком большим. Это может сделать алгоритм численно неустойчивым.

Обе проблемы можно решить, используя *метод главных элементов*. В каждом столбце i выбирается главный элемент `A[k,i]`, имеющий наибольшее абсолютное значение. Перед следующим шагом исключений выполняется перестановка строк k и i. Ее лучше реализовать с помощью перестановки указателей на строки, а не путем реальных обменов значений их элементов.

[23] При этом уравнения должны быть независимыми, т.е. никакое уравнение системы не может быть получено из других. Точнее, A должна быть несингулярной (неособенной) матрицей.

11.3.2. LU-разложение

Метод исключений Гаусса преобразует уравнение `Ax = b` в эквивалентное ему уравнение `Ux = y`, где `U` — верхняя треугольная матрица. При этом вычисляется последовательность множителей. Вместо того, чтобы отбрасывать их, предположим, что они сохраняются в третьей матрице `L`. Пусть матрица `L` — *нижняя треугольная матрица*, в которой все элементы, расположенные выше главной диагонали, равны нулю. Каждый элемент `L[j,i]` на диагонали и под ней имеет значение вида `A[j,i]/pivot`, где `pivot` — значение ведущего элемента для столбца `i`. После заполнения матрицы `L` произведение матриц `L` и `U` будет в точности равно исходной матрице `A` (если не учитывать возможных ошибок округления). В частности, для нашей системы уравнений получим:

```
 1    0    0      1   1    1        1   1   1
 2    1    0  x   0  -3   -1   =    2  -1   1
-1  -2/3   1      0   0  -2/3      -1   1  -1
     L         x       U        =        A
```

Описанное преобразование матрицы `A` в треугольные матрицы `L` и `U` называется *LU-разложением* `A`.[24] Основная причина использования LU-разложения состоит в том, что после того, как вычислены матрицы `L` и `U`, уравнение `Ax = b` легко решить для любых правых частей `b`.[25] Любую систему уравнений можно записать в виде `LUx = b`. Данное уравнение представляет две треугольные системы: `Ly = b` и `Ux = y`. Для решения первой системы относительно `y` можно применить прямой ход процедуры Гаусса, а затем для решения второй системы относительно `x` — обратный ход.

Еще одно свойство LU-разложения состоит в том, что для обеих матриц не нужна дополнительная память. Преобразуя `A` в `L` и `U`, мы можем записывать элементы `L` и `U` вместо предыдущих элементов `A`. Это очевидно для `U`, поскольку преобразованные значения `A` — это значения `U`. Множители в `L` вычисляются точно так же, как при исключении элементов `A`, поэтому их можно записывать там, где находились бы нули в `A`. Наконец, все диагональные элементы `L` равны единице, так что хранить их не нужно.

В листинге 11.11 представлена последовательная программа для LU-разложения матрицы `A`. Чтобы сделать код более понятным, результаты сохраняются в отдельной матрице `LU`. При этом сохраняется матрица `A`, которая может понадобиться в дальнейшем. Во избежание деления на нуль или на малые числа использован метод главных элементов. Индексы ведущих строк хранятся в векторе `ps`. Вначале `ps[i]` равно `i`; элементы `ps` переставляются каждый раз, когда в качестве ведущего выбран недиагональный элемент. Таким образом, `ps` всегда содержит перестановку номеров от 1 до `n`. Во время процедуры исключений строки `A` и `LU` доступны с помощью `ps[i]`, а не `i`.

Листинг 11.11. Последовательная программа LU-разложения

```
double A[1:n,1:n], LU[1:n,1:n]; # предполагается, что A инициализирована
int ps[1:n]; # индексы ведущих строк
double pivot; int pivotRow; # ведущее значение и строка
double mult; int t; #временные переменные
#инициализация ps и LU
for [i = 1 to n] {
```

[24] Название объясняется тем, что *L* — нижняя (*Lower*) треугольная матрица, а *U* — верхняя (*Upper*). — Прим. ред.

[25] Другой способ решения уравнений со многими правыми частями состоит в том, чтобы сначала вычислить матрицу, обратную матрице `A`, а затем умножать `A-1` на каждый вектор `b`. В упражнениях в конце раздела описано, как вычислить `A-1`. Однако вычисление LU-разложения матрицы занимает меньше времени, чем поиск обратной.

```
  ps[i] = i;
  for [j = 1 to n]
    LU[i,j] = A[i,j];
}
# исключения Гаусса с помощью ведущих элементов
for [k = 1 to n-1] { # итерации вниз по главной диагонали
  pivot = abs(LU[ps[k],k]); pivotRow = k;
  for [i = k+1 to n] { # выбор ведущего элемента в столбце k
    if (abs(LU[ps[i],k]) > pivot) {
     pivot = abs(LU[ps[i],k]); pivotRow = i;
    }
  }
  if (pivotRow != k) { # перестановка строк с помощью
                       # перестановки указателей
    t = ps[k]; ps[k] = ps[pivotRow]; ps[pivotRow] = t;
  }
  pivot = LU[ps[k],k]; # настоящее значение ведущего элемента
  for [i = k+1 to n] { # для всех строк в подматрице
    mult = LU[ps[i],k]/pivot; # вычислить множитель
    LU[ps[i],k] = mult;       # и сохранить его
    for [j = k+1 to n] # исключение по столбцам
      LU[ps[i],j] = LU[ps[i],j] - mult*LU[ps[k],j];
  }
}
```

В листинге 11.12 представлена последовательная программа для решения уравнения `Ax = b` по данному LU-разложению матрицы `A`. В первом цикле уравнение `Ly = b` решается относительно `y`; во втором — уравнение `Ux = y` относительно `x`. Программа сохраняет промежуточные результаты `y` в векторе `x`, поскольку это упрощает фазу обратного хода.

Листинг 11.12. Решение уравнения `A x = b` по данному LU-разложению `A`

```
double LU[1:n,1:n]; int ps[1:n]; #см. листинг 11.11
double sum, x[1:n], b[1:n];
# прямой ход для решения L y = b с записью y в x
for [i = 1 to n] {
  sum = 0.0;
  for [j = 1 to i-1]
    sum = sum + LU[ps[i],j] * x[j];
  x[i] = b[ps[i]] - sum;
}
# обратный ход для решения U x = y относительно x
for [i = n to 1 by -1] {
  sum = 0.0;
  for [j = i+1 to n]
  sum = sum + LU[ps[i],j] * x[j];
 x[i] = (x[i] - sum) / LU[ps[i],i];
}
```

11.3.3. Программа с разделяемыми переменными

Рассмотрим, как распараллелить программы в листингах 11.11 и 11.12, используя `PR` процессоров и, соответственно, `PR` рабочих процессов. Вначале рассмотрим LU-разложение (см. листинг 11.11). В нем есть две фазы: инициализация `ps` и `LU`, а затем прямой ход исключений Гаусса. В фазе инициализации тела циклов независимы, поэтому их можно разделить

между рабочими, используя любую схему распределения, которая каждому рабочему назначает поровну элементов данных.

Внешний цикл (по k) в фазе исключений должен выполняться последовательно каждым рабочим процессом, поскольку LU-разложение происходит итеративно вниз по главной диагонали и разлагает подматрицу LU[k:n,k:n]. Тело внешнего цикла имеет две фазы — выбор ведущего элемента и ведущей строки, затем сокращение строк под ведущей. Ведущий элемент можно выбрать следующими тремя способами.

- Каждый процесс просматривает все элементы в LU[k:n,k] и выбирает наибольший. Если каждый процесс сохраняет свою собственную копию индексов ведущих элементов ps, то после завершения этой фазы барьер не нужен.
- Один процесс просматривает все элементы в LU[k:n,k], выбирает наибольший и меняет местами ведущую строку и строку k. Здесь нужна точка барьерной синхронизации.
- Каждый рабочий процесс проверяет свое подмножество элементов из LU[k:n,k], выбирает наибольший элемент из подмножества и затем согласовывает с другими выбор ведущего элемента. Здесь также нужна точка барьерной синхронизации и в зависимости от того, как она запрограммирована, собственные копии ps.

Для малых значений n более быстрым будет первый подход, поскольку в нем нет барьеров. Для больших значений n более быстрым, вероятно, окажется третий подход. Точка пересечения (графиков сложности) зависит от того, как накладные расходы, связанные с барьером, соотносятся с временем выбора наибольшего элемента.

После выбора ведущего элемента все строки под ведущей строкой можно исключить параллельно. Для каждой строки сначала вычисляется и сохраняется множитель mult, затем выполняются итерации по столбцам, находящимся справа от столбца с ведущим элементом. По мере выполнения LU-разложения подматрица, в которой проводятся исключения, уменьшается, как и объем работы в фазах исключений. Таким образом, LU-матрицу нужно назначить рабочим процессам по полосам или обратным полосам, чтобы у каждого процесса постоянно была какая-то работа, кроме последних нескольких итераций в главном цикле. Вновь используем схему распределения по полосам, поскольку она проще программируется, чем схема с обратными полосами, и приводит к достаточно сбалансированной нагрузке.

В листинге 11.13 представлен эскиз параллельной программы LU-разложения с разделяемыми переменными. По сравнению с последовательной программой она имеет следующие основные отличия: 1) в фазах инициализации и исключений используются полосы строк; 2) после инициализации, каждой фазы выбора ведущего элемента (если необходимо) и каждого шага исключений установлены барьеры. Кроме того, каждому рабочему, возможно, нужна своя собственная локальная копия ведущих индексов, поскольку это упрощает перестановку строк и не требует синхронизации.

Листинг 11.13. Структура программы LU-разложения с разделяемыми переменными

```
double A[1:n,1:n], LU[1:n,1:n]; # предполагается, что A инициализирована
int ps[1:n]; # индексы ведущих строк

procedure barrier(int id) { ... ) # см. главу 3

process Worker(w = 1 to PR) {
  double pivot, mult;
  декларации других локальных переменных, например копии ps;
  for [i = w to n by PR]
    инициализация ps и своих полос LU;
  barrier (w);
```

```
  # исключения Гаусса с ведущими элементами
  for [k = 1 to n-1] { # итерации вниз по главной диагонали
    поиск наибольшего ведущего элемента — см. текст;
    если необходимо, перестановка ведущей строки со строкой k,
    затем вызов barrier(w);
    pivot = LU[ps[k],k]; # вычисление настоящего значения ведущего элемента
    for [i = k+1 to n st (i%PR == 0)] { # на своей полосе
      mult = LU[ps[i],k]/pivot; #вычислить множитель
      LU[ps[i],k] = mult; # и сохранить его
      for [j = k+1 to n] # исключения по столбцам
        LU[ps[i],j] = LU[ps[i],j] - mult*LU[ps[k],j];
    }
    barrier(w);
  }
}
```

Рассмотрим, как распараллелить прямой и обратный проходы (см. листинг 11.12). К сожалению, в каждой фазе есть вложенные циклы, и каждый внутренний цикл зависит от значений, вычисленных на предыдущих итерациях соответствующего внешнего цикла. По определению прямой ход вычисляет элементы y по одному, а обратный — элементы x, также по одному.

В эти циклы можно внести независимость. Например, в фазе прямого хода можно развернуть внутренние циклы и переписать код, чтобы значения `x[i]` вычислялись в терминах LU и b. Вручную это делать утомительно, лучше использовать компилятор (см. раздел 12.2).

Другой способ получить параллельность — использовать так называемую *синхронизацию фронта волны* (wave front synchronization). В фазе прямого хода итерации назначаются рабочим по полосам. Поскольку вычисления `x[i]` зависят от предыдущих значений `x[1:i-1]`, с каждым элементом x можно связать флаг (или семафор). Закончив вычисление `x[i]`, процесс устанавливает флаг для этого элемента. Когда процессу нужно прочитать значение `x[i]`, он сначала ждет, пока для этого элемента не будет установлен флаг. Например, код, выполняемый рабочим процессом w в прямом ходе, мог бы быть таким.

```
for [i = w to n by PR] { # на своей полосе x[*]
  sum = 0.0;       # локальное значение
  for [j = 1 to i-1] {
    ожидать, пока не будет установлен флаг x[j];
    sum = sum + LU[ps[i],j] * x[j];
  }
  x[i] = b[ps[i]] - sum;
  установить флаг x[i];
}
```

Фронт волны представляет собой установку флагов по мере вычисления новых элементов. (Термин *фронт волны* обычно используется для матриц; *волна*, как правило, представляет собой диагональную линию, движущуюся по матрице.)

Волновые фронты эффективны, если накладные расходы при синхронизации невелики по сравнению с объемом вычислений. Здесь же на каждый элемент приходится очень мало вычислений, поэтому синхронизацию можно запрограммировать с помощью простых флагов и активного ожидания. Это должно дать небольшое увеличение производительности данного приложения.

11.3.4. Программа с передачей сообщений

Рассмотрим, как реализовать LU-разложение с помощью передачи сообщений. Вновь рассмотрим три подхода — управляющий-рабочие, алгоритмы пульсации и конвейера. Можно использовать все три парадигмы, однако если в программе с разделяемыми переменными

для некоторого приложения применяются барьеры, то естественнее всего построить распределенную программу на основе алгоритма пульсации. Ниже приведен эскиз программы пульсации для LU-разложения. Две другие парадигмы рассмотрены в упражнениях в конце главы.

Как обычно, при создании распределенной программы сначала нужно решить, как распределить данные, чтобы вычислительная нагрузка оказалась сбалансированной. Поскольку LU-разложение работает с уменьшающимися подматрицами, объем работы также уменьшается по мере выполнения исключений. Поэтому можно назначить строки по полосам. Если предположить, что есть PR рабочих процессов, то рабочему процессу назначается каждая PR-я строка LU, по n/PR строк на каждый процесс.

Первый шаг в LU-разложении — инициализация локальных строк LU и индексов ведущих элементов ps. Все процессы могут выполнить этот шаг параллельно. После инициализации барьер не нужен, поскольку здесь нет разделяемых переменных.

Главный шаг в LU-разложении — многократное повторение выбора ведущего элемента и ведущей строки с последующим исключением всех строк, расположенных ниже ведущей. Каждый рабочий процесс может выбрать наибольший элемент в столбце k своих n/PR строк в матрице LU. Однако для выбора глобального максимума рабочим нужно взаимодействовать. Можно использовать один процесс в качестве управляющего, который собирает максимальные значения от всех процессов, выбирает наибольшее из них и рассылает его копии. Или же, если доступны такие глобальные примитивы взаимодействия, как в библиотеке MPI, то для вычисления ведущего значения можно использовать примитив редукции.

После выбора ведущего значения процесс, которому принадлежит ведущая строка, должен передать ее другим, поскольку она им нужна в фазе исключения. Получив ведущие значение и строку, каждый рабочий процесс может выполнить исключение строк своей области под ведущей строкой.

В листинге 11.14 содержится эскиз программы с передачей сообщений для LU-разложения. Все шаги в программе такие, как описано выше. Явные барьеры здесь не нужны, поскольку обмен сообщениями, необходимый при выборе ведущего значения и ведущей строки, по сути, является барьером. В фазе исключений используется переменная myRow, чтобы отображать глобальный индекс i-й строки (который находится в диапазоне от 1 до n) в индекс соответствующей строки в локальном массиве строк.

Листинг 11.14. Эскиз программы с передачей сообщений для LU-разложения

```
declarations каналов;

process Worker(w = 1 to PR) {
  double LU[1:n/PR,1:n/PR]; # свои строки LU
  int ps[1:n/PR];           #индексы ведущих строк
  double pivot, mult, pivotRow[n];
  int myRow;
  declarations других локальных переменных;
  initialization ps и своих строк в LU;
  # исключения Гаусса с главными элементами
  for [k = 1 to n-1] { # итерации вниз по главной диагонали
    поиск наибольшего ведущего элемента в столбце k своих строк;
    обмен pivot с другими процессами;
    выбор глобального максимального элемента и обновление ps;
    if (владелец ведущей строки)
      рассылка копий pivotRow другим процессам;
    else
      получение pivotRow;
    # исключение своих строк в LU с помощью pivot и pivotRow
```

```
    for [i = k+1 to n st (i%PR == 0)] { # для своей полосы
      myRow = i/PR; # преобразовать индекс строки
      mult = LU[ps[myRow],k]/pivot; # вычислить множитель
      LU[ps[myRow],k] = mult; # и сохранить его
      for [j = k+1 to n] # исключение по столбцам
        LU[ps[myRow],j] = LU[ps[myRow],j] -
                          mult * pivotRow[j];
    }
  }
}
```

Когда программа в листинге 11.14 завершается, результаты LU-разложения размещаются в локальных массивах рабочих процессов. Чтобы решить систему уравнений, нужно выполнить как прямой, так и обратный ход, т.е. действия, требующие доступа ко всем элементам LU-разложения. Первый подход состоит в использовании процесса, который собирает все строки LU и затем выполняет код из листинга 11.12. При втором используется круговой конвейер, чтобы реализовать синхронизацию фронта волны с помощью передачи сообщений.

В круговом конвейере первый рабочий процесс вычисляет `x[1]` и передает его второму. Второй процесс вычисляет `x[2]` и передает `x[2]` и `x[1]` третьему. Последний процесс вычисляет `x[PR]` и передает его и все предыдущие значения первому. Это продолжается до тех пор, пока не будет вычислен `x[n]`. Можно использовать такой же конвейер для обратного хода, вычисляя окончательные значения `x[n]`, `x[n-1]` и так вплоть до `x[1]` и передавая их по конвейеру.

Конвейер для прямого и обратного хода относительно легко программируется, параллелен по существу и не требует сбора всех элементов LU. Однако ему нужно много сообщений, поэтому вполне вероятно, что он может оказаться менее эффективным, чем алгоритм с одним процессом.

Историческая справка

История научных вычислений тесно связана с общей историей вычислений. Первые вычислительные машины разрабатывались для решения научных проблем, а Фортран — первый язык высокого уровня — был создан специально для программирования численных методов. Научные вычисления также стали синонимом высокопроизводительных вычислений, заставляя увеличивать предельные возможности самых быстрых машин.

Математические основы научных вычислений представлены в учебниках по численному анализу, а методы решения дифференциальных и матричных уравнений — по вычислительной математике. Книга [Forsythe and Moler, 1967] является классической; в ней содержатся очень ясные описания линейных алгебраических систем и алгоритмов их решения. Она послужила основным источником для материала раздела 11.2. Книга [Van Loan, 1997] представляет собой введение в (непараллельные) научные вычисления, включая линейные и нелинейные системы уравнений. В учебнике [Mathews and Fink, 1999] описываются численные методы для целого ряда приложений, включая интегрирование, системы уравнений и дифференциальные уравнения. В последних двух книгах для иллюстрации алгоритмов используется пакет MATLAB.

Некоторые книги по параллельным вычислениям дают более подробные описания методов данной главы, представляют родственные алгоритмы и дополнительные темы. В книге [Fox et al., 1998] рассматриваются методы конечных разностей и конечных элементов для решения дифференциальных уравнений в частных производных, матричных и точечных вычислений, а также методы Монте-Карло (рандомизированные). Особое внимание уделяется алгоритмам с передачей сообщений, которые предназначены для работы на гиперкубах, но могут быть адаптированы к другим архитектурам с распределенной памятью. В книге [Bertsekas and Tsitsiklis, 1989] рассматриваются прямые и итерационные методы для ряда линейных и нелинейных задач, динамическое программирование, проблемы потоков в сетях и асин-

хронные итерационные методы. Кумар [Kumar et al., 1994] и Куинн [Quinn, 1994] описывают сеточные и матричные вычисления, сортировку, поиск, алгоритмы в графах, а также быстрое преобразование Фурье. В книге [Wilkinson and Allen, 1999] показано, как итерации Якоби представляются в виде системы линейных уравнений; в эту книгу включены главы по обработке изображений, поиску и оптимизации.

Многосеточные методы были разработаны несколькими исследователями в 1970-х годах. Их изучение и усовершенствование продолжается, что обусловлено их важностью в вычислениях на сетках больших размеров. Отличное введение в данную тему содержится в книге [Briggs, 1987], а [Hackbrush, 1985] — одна из немногих книг, в которых многосеточные методы описаны подробно.

Гравитационная задача *n* тел изучается уже не одно десятилетие. Барнс и Хат [Barnes and Hut, 1986] написали классическую статью по иерархическому древовидному коду для решения данной задачи. Салмон (Salmon) в своей докторской диссертации проделал первичную работу по параллельным иерархическим методам для задачи *n* тел. В работе [Warren and Salmon, 1992] описана имитация с 17,5 миллиона тел, а в [Warren, Salmon, and Becker, 1997] — с числом тел, превышающим 322 миллиона, которая выполнялась за 437 временных шагов более чем на тысяче процессоров! Обе эти работы получили приз Гордона Белла (Gordon Bell) за выдающееся непрерывное выполнение на машинах с массовым параллелизмом.

В книге [Greengard, 1987] и статье [Greengard and Rokhlin, 1987] описаны быстрый метод мультиполей и его приложение к оценке потенциальных полей в точечных имитациях. В работе [Singh, Hennessy, and Gupta, 1995] представлен отличный обзор по алгоритму Барнса–Хата и быстрому методу мультиполей, описано, как их реализовать на больших мультипроцессорах и оценить влияние архитектуры на их эффективность. В [Blackston and Suel, 1997] представлены переносимые и эффективные параллельные реализации методов Барнса–Хата, мультиполей и еще один алгоритм, называемый методом Андерсона.

Литература

Barnes, J., and P. Hut. 1986. A hierarchical *O(N* log *N)* force-calculation algorithm. *Nature* 324, 446–449.

Bertsekas, D. P., and J. N. Tsitsiklis. 1989. *Parallel and Distributed Cjmputation: Numerical Methods*. Englewood Cliffs, NJ: Prentice-Hall.

Blackston, D., and T. Suel. 1997. Highly portable and efficient implementations of parallel adaptive *n*-body methods. *Proc. Supercomputing '97*, November, online at `http://www.supercomp.org/sc97/proceedings`.

Briggs, W. L. 1987. *A multigrid Tutorial*. Philadelphia: SIAM Publications.

Forsythe, G., and C. B. Moler. 1967. *Computer Solution of Linear Algebra Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Fox, G. C., M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. 1988. *Solving Problems on Concurrent Processors*, vol. 1, *General Techniques and Regular Problems*. Englewood Cliffs, NJ: Prentice-Hall.

Greengard, L. 1987. *The Rapid Evaluation of Potential Fields in Particle Systems*. New York: ACM Press.

Greengard, L., and V. Rokhlin. 1987. A fast algorithm for particle simulation. *Journal of Computational Physics* 73: 325.

Hackbrush, W. 1985. *Multigrid Methods with Applications*. New York: Springer Verlag.

Kumar, V., A. Grama, A. Gupta, and G. Karypis. 1994. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Menlo Park, CA: Benjamin/Cummings.

Mathews, J. H., and K. D. Fink. 1999. *Numerical Methods Using MATLAB*, 3rd ed. Upper Saddle River,NJ: Prentice-Hall.

Quinn, M. J. 1994. *Parallel Computing: Theory and Practice*. New York: McGraw-Hill.

Singh, J. P., J. L. Hennessy, and A. Gupta. 1995. Implications of hierarchical *n*-body methods for multiprocessor architectures. *ACM Trans. on Computer Systems* 13, 2 (May): 141–202.

Warren, M. S., and J. K. Salmon. 1992. Astrophysical *n*-body simulations using hierarchical tree data structures. *Proc. Supercomputing '92*, November, 570–576.

Warren, M. S., J. K. Salmon, and D. J. Becker. 1997. Pentium Pro inside: I. A treecode at 430 Gigaflops on ASCI Red; II. Price/performance of $50/Mflop on Loki and Hyglac. *Proc. Supercomputing '97*, November, online at `http://www.supercomp.org/sc97/proceedings`.

Wilkinson, B., and M. Allen. 1999. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, NJ: Prentice-Hall.

Van Loan, C. F. 1997. *Introduction to Scientific Computing*. Upper Saddle River, NJ: Prentice-Hall.

Упражнения

11.1. В начале подраздела по последовательным итерациям Якоби представлена простая последовательная программа. Затем описаны четыре оптимизации, приводящие к программе в листинге 11.1, и еще две оптимизации:

а) постройте серию экспериментов для измерения индивидуального и совокупного эффекта от данных шести оптимизаций. Начните с измерения времени выполнения главного цикла вычислений в программе для метода итераций Якоби на сетках различных размеров, например `64x64`, `128x128` и `256x256`. Подберите такие значения `EPSILON` и/или `iters`, чтобы вычисления занимали несколько минут. (Время должно быть достаточно большим, чтобы заметить эффект от улучшений.) Затем добавляйте в код по одной оптимизации и измеряйте повышение производительности. Составьте краткий отчет с результатами измерений и выводами;

б) проведите эксперименты, позволяющие оценить все шесть оптимизаций по отдельности и в различных комбинациях друг с другом, в отличие от пункта *а*, где оптимизации добавлялись одна за другой. Какие оптимизации оказались наиболее продуктивными? Укажите, имеет ли значение порядок их применения;

в) рассмотрите другие способы оптимизации программы. Ваша цель — получить максимально быструю программу. Опишите и измерьте эффект от каждой дополнительной оптимизации, которую вы придумали.

11.2. Реализуйте программу для метода итераций Якоби с разделяемыми переменными (см. листинг 11.2) и измерьте ее производительность. Используйте разные размеры сеток и количества процессоров. Оптимизируйте программу своим способом и измерьте производительность улучшенной программы. Составьте отчет, в котором будут описаны все проделанные изменения, представлены результаты измерений и выводы.

11.3. Реализуйте неоптимизированную и оптимизированные программы для метода итераций Якоби с передачей сообщений (см. листинги 11.3 и 11.4) и измерьте их производительность. Используйте сетки разных размеров и разное количество процессоров. Затем оптимизируйте программы своим способом и измерьте производительность улучшенных программ. Составьте отчет, в котором будут описаны все проделанные изменения, представлены результаты измерений и выводы.

11.4. Реализуйте программу по методу "красное-черное" Гаусса–Зейделя с передачей сообщений (см. листинг 11.5) и измерьте производительность. Сделайте свою программу максимально эффективной. Какое ускорение получается, если запускать программу на сетках разных размеров и с разным числом процессоров?

11.5. Рассмотрите многосеточные методы, описанные в разделе 11.1 и проиллюстрированные на рис. 11.2 и 11.3:

а) составьте последовательную программу, которая реализует один *V*-цикл. Используйте три уровня и, соответственно, сетки четырех размеров. Используйте размеры сеток, которые облегчают расчеты. Например, если самая малая сетка имеет ширину 16, то у нее 15 внутренних точек. Ширина других сеток тогда будет равна 32, 64 и 128;

б) распараллельте программу, написанную для пункта *а*, чтобы получить программу с разделяемыми переменными;

в) измените составленную для пункта *б* программу, используя передачу сообщений вместо разделяемых переменных. Возможно использование парадигмы пульсации, как в листинге 11.4;

г) измерьте быстродействие программ, разработанных для пунктов *б* и *в*, на сетках разных размеров и с разным числом процессоров. Оцените ускорение. Составьте отчет, в котором описываются и анализируются полученные результаты;

д) измените свои программы, разработанные для пунктов *а*–*в*, вместо *V*-цикла используя *W*-цикл. Вновь измерьте производительность всех программ и сравните с результатами из пункта *г*;

е) измените свои программы так, чтобы они реализовывали полный многосеточный метод. Вновь измерьте их производительность и сравните с результатами из пункта *г*.

11.6. В разделе 11.1 описаны три метода решения уравнения Лапласа: итераций Якоби, "красное-черное" Гаусса–Зейделя и многосеточный. Реализуйте последовательные и параллельные версии этих методов. Составляя параллельные программы, используйте разделяемые переменные и/или передачу сообщений, в зависимости от доступного аппаратного обеспечения. Измерьте время выполнения построенных программ и ускорение параллельных программ. Составьте отчет, в котором представлены и проанализированы полученные результаты. Также опишите, насколько трудно было написать каждую из параллельных программ. Хотя это и субъективно, постарайтесь точно описать трудности, с которыми вы столкнулись, и то, как вы их преодолели.

11.7. В листингах 11.6. и 11.7–11.10 представлены пять программ для решения двухмерной гравитационной задачи *n* тел:

а) реализуйте каждую программу. Распределите тела однородно по окружности или внутри нее. Массы тел задайте достаточно малыми, чтобы уменьшить вероятность столкновений. (Столкновение проявляется в виде арифметического переполнения при делении на `distance**2`.) Можно также "поиграть" значением `G`;

б) проведите серию экспериментов по измерению производительности своих программ для различных (больших) количеств тел и количеств процессоров. Составьте отчет, представив и проанализировав полученные результаты;

в) модифицируйте программы для трех измерений пространства и повторите пункт *б*.

11.8. Рассмотрите алгоритм Барнса–Хата, описанный в конце раздела 11.2. Напишите для него последовательную программу и параллельные программы с разделяемыми переменными и передачей сообщений. Для программы с передачей сообщений выберите одну из парадигм — управляющий-рабочие, пульсация или конвейер. Затем проведите серию экспериментов по измерению производительности разработанных программ для различных (больших) количеств тел и количеств процессоров. Составьте отчет, представив и проанализировав полученные результаты. Какое ускорение наблюдалось?

11.9. Промоделируйте движение плоских дисков на бильярдном столе без трения или движение шаров внутри ящика с нулевой гравитацией. Каждый объект представьте как частицу. Задайте начальные положения и скорости всех частиц, затем промоделируйте их

движение, при котором они ударяются о стенки или друг о друга. Предположите, что движение частиц идеальное, упругое и без трения. Реализуйте вашу модель сначала в виде последовательной, а затем параллельной программы. При выводе результатов желательно использовать какое-либо графическое устройство.

11.10. Рассмотрим следующую систему уравнений.

```
 e + f +  g +  h = 10
 e     -  g      = -2
2e - f + 4g - 3h =  0
     f +  g -  h =  1
```

Запишите ее в виде матричной задачи `A x = b`:

а) выполните исключения Гаусса для `A` и `b`. Покажите, как изменяются `A` и `b` по мере исключения `e` из трех уравнений, затем `f` — из двух оставшихся и т.д. Выполните обратный ход, чтобы вычислить значения `x`;

б) выполните LU-разложение матрицы `A`. Покажите, как изменяется матрица каждый раз после выбора ведущей строки. Решите систему уравнений, сначала используя прямой ход для решения уравнения `Ly = b`, а затем обратный ход для решения `Ux = y`.

11.11. Рассмотрите последовательную программу LU-разложения (см. листинг 11.11):

а) реализуйте программу. Добавьте в нее код, в котором проверялась бы корректность вычислений, т.е. что `L×U` равно `A`;

б) измените программу LU-разложения, чтобы разложение вычислялось на месте, т.е. в матрице `A`, как описано в тексте.

11.12. Рассмотрите эскизы распределенной программы и программы с разделяемыми переменными для LU-разложения в листингах 11.13 и 11.14:

а) разработайте реализации программ, используя язык высокого уровня, например SR, или библиотеку передачи сообщений, например MPI;

б) проведите серию экспериментов по измерению производительности построенных программ. Используйте различные большие размеры матриц и запускайте программы с разным числом процессоров. Вычислите ускорение каждой программы;

в) постройте программу LU-разложения, реализующую алгоритм конвейера, запустите ее и измерьте производительность;

г) постройте программу LU-разложения на основе алгоритма "управляющий-рабочие", запустите ее и измерьте производительность;

д) сравните производительность трех программ с передачей сообщений, реализующих алгоритмы пульсации, конвейера и "управляющий-рабочие". Составьте таблицу, аналогичную табл. 11.1, и определите ускорение каждой программы для разных размеров матриц и количеств процессоров.

11.13. LU-разложение имеет очень много накладных расходов на взаимодействие и синхронизацию. Реализуйте параллельную программу LU-разложения, используя разделяемые переменные или передачу сообщений. Затем проведите серию экспериментов, которая позволит определить, как много вычислений нужно, чтобы программа была масштабируемой, т.е. имела эффективность, близкую к 1.0, по мере увеличения числа процессоров. Составьте отчет с анализом экспериментов и результатов.

11.14. Матрица `A-1` называется *обратной* к несингулярной матрице `A`, если `A×A-1 = I`, где `I` — единичная матрица, т.е. матрица, у которой по диагонали расположены единицы, а все остальные элементы равны нулю. Столбцы `A-1` являются решениями линейных систем

$$A\,x_1 = e_1,\quad A\,x_2 = e_2,\quad \ldots,\quad A\,x_n = e_n.$$

Каждый e_i — это единичный вектор, у которого i-й элемент равен единице, а все остальные равны нулю:

а) напишите последовательную программу для вычисления матрицы, обратной к исходной матрице A. Сначала вычислите LU-разложение A (см. листинг 11.11). Затем решите указанные n уравнений (см. листинг 11.12);

б) распараллельте программу из пункта *а*, используя или разделяемые переменные, или передачу сообщений. (*Указание.* Заданные n уравнений независимы.) Оцените производительность программы и вычислите ее ускорение для матриц различных размеров и разных количеств процессоров.

11.15. Исследуйте какую-нибудь вычислительную задачу или метод (например, быстрое преобразование Фурье, быстрый метод мультиполей, задачу нелинейной оптимизации, метод сопряженных градиентов для решения дифференциальных уравнений в частных производных или методы Монте-Карло). Выберите алгоритм, напишите для него последовательную и параллельные программы, затем измерьте производительность программ. Составьте отчет, в котором подробно описана задача, алгоритм, программы и полученные результаты по производительности.

Глава 12

Языки, компиляторы, библиотеки и инструментальные средства

До сих пор основное внимание в данной книге уделялось созданию императивных программ с явно заданными процессами, взаимодействием и синхронизацией. Эти программы наиболее распространены и являются тем, что выполняет аппаратура. Для ясности и компактности программы записывались с помощью нотации высокого уровня. Способы описания параллельности в ней похожи на механизмы языка SR, а нотация последовательных процессов аналогична языку C. В данной главе рассматриваются дополнительные подходы и инструментальные средства для организации высокопроизводительных вычислений.

При написании параллельных программ чаще всего берется какой-нибудь последовательный язык и соответствующая библиотека подпрограмм. Тела процессов записываются на последовательном языке, например, C или Фортране. Затем с помощью вызовов библиотечных функций программируется создание процессов, их взаимодействие и синхронизация. Нам уже знакомы библиотека Pthread, предназначенная для машин с разделяемой памятью, и библиотека MPI для обмена сообщениями. В разделе 12.1 показано, как с помощью этих библиотек запрограммировать метод итераций Якоби. Затем рассматривается технология OpenMP — новый стандарт программирования с разделяемыми переменными. Использование OpenMP проиллюстрировано также на примере итераций Якоби.

Совершенно другой подход к разработке параллельных программ состоит в использовании распараллеливающего компилятора. Такой компилятор сам находит параллельность в последовательной программе и создает корректно синхронизированную параллельную программу, которая содержит последовательный код и библиотечные вызовы. В разделе 12.2 приводится обзор распараллеливающих компиляторов. Там описан анализ зависимостей, на основе которого определяется, какие части программы могут выполняться параллельно. Затем рассмотрены различные преобразования программ, наиболее часто применяемые для конвертирования последовательных циклов в параллельные. Преимущество распараллеливающих компиляторов состоит в том, что они освобождают программиста от изучения правил написания параллельных программ и могут быть использованы для распараллеливания уже имеющихся приложений. Однако с их помощью часто невозможно распараллелить сложные алгоритмы и, как правило, трудно добиться оптимальной производительности.

Третий способ разработки параллельных программ — использовать языки высокого уровня, в которых параллельность (вся или ее часть), взаимодействие и синхронизация неявны. В разделе 12.3 описано несколько классов языков высокого уровня и проанализированы основные языки из каждого класса. Для иллюстрации использования каждого из языков и их сравнения в качестве примеров используются метод итераций Якоби и другие приложения из предыдущих глав. Также описаны три абстрактные модели, которые можно использовать для характеристики времени работы параллельных алгоритмов. Раздел заканчивается учебным примером по быстродействующему Фортрану (High Performance Fortran — HPF), самому последнему в семействе языков на основе Фортрана, предназначенных для научных вычислений. Компиляторы языков, подобных HPF, опираются на методы распараллеливания и создают программы, содержащие последовательный код и библиотечные вызовы.

В разделе 12.4 представлены программные инструменты, помогающие в разработке, оценке и использовании параллельных программ. Сначала рассмотрены инструментальные средства для измерения производительности, визуализации и так называемого управления

вычислениями. Затем описаны *метавычисления* — новый подход, позволяющий объединять вычислительную мощность разнотипных машин, соединенных высокоскоростными сетями. Например, моделирующая часть научных вычислений может выполняться на удаленном суперкомпьютере, а управляющая и графическая части — на локальной графической рабочей станции. В качестве конкретного примера в конце раздела 12.4 описан новый инфраструктурный набор программных инструментов Globus для поддержки метавычислений.

12.1. Библиотеки параллельного программирования

Библиотеки параллельного программирования представляют собой набор подпрограмм, обеспечивающих создание процессов, управление ими, взаимодействие и синхронизацию. Эти подпрограммы, и особенно их реализация, зависят от того, какой вид параллельности поддерживает библиотека — с разделяемыми переменными или с обменом сообщениями.

При создании программ с разделяемыми переменными на языке C обычно используют стандартную библиотеку Pthreads. При использовании обмена сообщениями стандартными считаются библиотеки MPI и PVM; обе они имеют широко используемые общедоступные реализации, которые поддерживают как C, так и Фортран. OpenMP является новым стандартом программирования с разделяемыми переменными, который реализован основными производителями быстродействующих машин. В отличие от Pthreads, OpenMP является набором директив компилятора и подпрограмм, имеет связывание, соответствующее языку Фортран, и обеспечивает поддержку вычислений, параллельных по данным. Далее в разделе показано, как запрограммировать метод итераций Якоби с помощью библиотек Pthreads и MPI, а также директив OpenMP.

12.1.1. Учебный пример: Pthreads

Библиотека Pthreads была представлена в разделе 4.6, где рассматривались подпрограммы для использования потоков и семафоров. В разделе 5.5 были описаны и проиллюстрированы подпрограммы для блокировки и условных переменных. Эти механизмы можно использовать и в программе, реализующей метод итераций Якоби (листинг 12.1) и полученной непосредственно из программы с разделяемыми переменными (см. листинг 11.2). Как обычно в программах, использующих Pthreads, главная подпрограмма инициализирует атрибуты потока, читает аргументы из командной строки, инициализирует глобальные переменные и создает рабочие процессы. После того как завершаются вычисления в рабочих процессах, главная программа выдает результаты.

Листинг 12.1. Метод итераций Якоби с использованием Pthreads

```
/* Метод итераций Якоби с использованием Pthreads */
#include <pthread.h>
#include <stdio.h>
#define SHARED 1
#define MAXGRID 258 /* максимальный размер сетки с границами */
#define MAXWORKERS 16 /* максимальное число рабочих потоков */

void *Worker(void *);
void Barrier(int);

int gridSize, numWorkers, numIters, stripSize;
double maxDiff[MAXWORKERS];
double grid[MAXGRID][MAXGRID], new[MAXGRID][MAXGRID];
декларации других глобальных переменных, например барьерных флажков;

int main(int argc, char *argv[]) {
  pthread_t workerid[MAXWORKERS]; /* индексы потоков и */
  pthread_attr_t attr;   /* их атрибуты */
```

```
  int i; double maxdiff = 0.0;

/* установка глобальных атрибутов потока */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* чтение аргументов из командной строки */
  /* предполагается, что gridSize кратно numWorkers */
  gridSize = atoi(argv[1]); numWorkers = atoi(argv[2]);
numIters = atoi(argv[3]);
stripSize = gridSize/numWorkers;

  инициализация сеток и барьерных флажков;

  /* создание рабочих потоков и ожидание их завершения */
  for (i = 0; i < numWorkers; i++)
    pthread_create(&workerid[i], &attr, Worker,
                   (void *) i);
  for (i = 0; i < numWorkers; i++)
    pthread_join(workerid[i], NULL);
  вычислить maxdiff и вывести результаты;
}

void *Worker(void *arg) {
  int myid = (int) arg;
  double mydiff;
  int i, j, iters, firstRow, lastRow;
/* определить первую и последнюю строки своей полосы */
  firstRow = myid*stripSize + 1;
  lastRow = firstRow + stripSize - 1;
  for (iters = 1; iters <= numIters; iters++) {
    /* обновить свои точки */
    for (i = firstRow; i<= lastRow; i++)
      for (j = 1; j <= gridSize; j++)
        new[i][j] = (grid[i-1][j] + grid[i+1][j] +
                     grid[i][j-1] + grid[i][j+1])*0.25;
    Barrier(myid);
    /* вновь обновить свои точки */
    for (i = firstRow; i<= lastRow; i++)
      for (j = 1; j <= gridSize; j++)
        grid[i][j] = (new[i-1][j] + new[i+1][j] +
                      new[i][j-1] + new[i][j+1])*0.25;
    Barrier(myid);
  }
  вычислить максимальную погрешность на своей полосе;
  maxDiff[myid] = mydiff;
}

void Barrier(int workerid) {
  /* подробно не рассматривается */
}
```

Каждый рабочий поток отвечает за сплошную полосу в двух сетках. Чтобы упростить задачу, объявляется фиксированный максимальный размер каждой сетки. Также предполагается, что размер сеток кратен числу рабочих потоков. Тело барьера в программе не записано; одна из Pthreads-реализаций барьера с помощью блокировок и условных переменных представлена в листинге 5.12. Однако, если каждый рабочий поток выполняется на своем собственном процессоре, намного эффективнее использовать барьер с распространением и активным ожиданием (см. раздел 3.4).

12.1.2. Учебный пример: MPI

Библиотека MPI была представлена в разделе 7.8. Напомним, что MPI содержит множество подпрограмм для глобального и локального взаимодействия. Некоторые из них использованы в программе для метода итераций Якоби (листинг 12.2). Она аналогична программе с обменом сообщениями в листинге 11.3.

Листинг 12.2. Метод итераций Якоби с использованием MPI

```
/* Метод итераций Якоби с использованием MPI */
#include <mpi.h>
#include <stdio.h>
#define MAXGRID 258 /* максимальный размер сетки с границами */
#define COORDINATOR 0 /* номер управляющего процесса */
#define TAG 0  /* не используется */

static void Coordinator(int,int,int);
static void Worker(int,int,int,int,int);

int main(int argc, char *argv[]) {
  int myid, numIters;
  int numWorkers, gridSize; /* предполагается, что */
  int stripSize;     /* gridSize кратно numWorkers */

  MPI_Init(&argc, &argv); /* инициализация MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &numWorkers);
  numWorkers--;  /* один управляющий, остальные - рабочие */
  прочитать gridSize и numIters; вычислить stripSize;
  if (myid == COORDINATOR)
    Coordinator(numWorkers, stripSize, gridSize);
  else
    Worker(myid,numWorkers,stripSize,gridSize,numIters);
  MPI_Finalize(); /* окончание работы MPI */
}

static void Coordinator(int numWorkers,
                  int stripSize, int gridSize) {
  double grid[MAXGRID][MAXGRID];
  double mydiff = 0.0, maxdiff = 0.0;
  int i, worker, startrow, endrow;
  MPI_Status status;

  /* получить окончательные значения в сетке от Workers */
  for (worker = 1; worker <= numWorkers; worker++) {
    startrow = (worker-1)*stripSize + 1;
    endrow = startrow +stripSize -1;
    for (i = startrow; i<= endrow; i++)
      MPI_Recv(&grid[i][1], gridSize, MPI_DOUBLE, worker,
          TAG, MPI_COMM_WORLD, &status);
  }
  /* редукция погрешностей от Workers */
  MPI_Reduce(&mydiff, &maxdiff, 1, MPI_DOUBLE,
        MPI_MAX, COORDINATOR, MPI_COMM_WORLD);
  вывести результаты;
}
```

```
static void Worker(int myid, int numWorkers,
         int stripSize, int gridSize, int numIters) {
  double grid [2][MAXGRID][MAXGRID];
  double mydiff, maxdiff;
  int i, j, iters;
  int current = 0, next = 1; /* текущая и следующая сетки */
  int left, right; /* соседние рабочие */
  MPI_Status status;

  инициализировать матрицы; определить соседей left и right;
  for (iters = 1; iters <= numIters; iters++) {
    /* обмен границами с соседями */
    if (right != 0) MPI_Send(&grid[next][stripSize][1],
        gridSize, MPI_DOUBLE, right, TAG, MPI_COMM_WORLD);
    if (left != 0) MPI_Send(&grid[next][1][1],gridSize,
        MPI_DOUBLE, left, TAG, MPI_COMM_WORLD);
    if (left != 0) MPI_Recv(&grid[next][0][1],gridSize,
        MPI_DOUBLE, left, TAG, MPI_COMM_WORLD, &status);
    if (right != 0) MPI_Recv(&grid[next][stripSize+1][1],
        gridSize, MPI_DOUBLE, right, TAG,
        MPI_COMM_WORLD, &status);
    /* обновить свои точки */
    for (i = 1; i <= stripSize; i++)
      for (j = 1; j <= gridSize; j++)
        grid[next][i][j] = (grid[current][i-1][j] +
          grid[current][i+1][j] + grid[current][i][j-1] +
          grid[current][i][j+1]) * 0.25;
    current = next; next = 1-next; /* поменять местами сетки */
  }

  /* отправить строки окончательной сетки управляющему процессу */
  for (i = 1; i <= stripSize; i++) {
    MPI_Send(&grid[current][i][1], gridSize, MPI_DOUBLE,
             COORDINATOR, TAG, MPI_COMM_WORLD);
  }
  вычислить mydiff;
  /* редукция mydiff в управляющем процессе */
  MPI_Reduce(&mydiff, &maxdiff, 1, MPI_DOUBLE,
              MPI_MAX, COORDINATOR, MPI_COMM_WORLD);
}
```

Программа в листинге 12.2 содержит три функции: `main`, `Coordinator` и `Worker`. Предполагается, что выполняются все `numWorkers+1` экземпляров программы. (Они запускаются с помощью команд, специфичных для конкретной версии MPI.) Каждый экземпляр начинается с выполнения подпрограммы `main`, которая инициализирует MPI и считывает аргументы командной строки. Затем в зависимости от номера (идентификатора) экземпляра из `main` вызывается либо управляющий процесс `Coordinator`, либо рабочий `Worker`.

Каждый процесс `Worker` отвечает за полосу точек. Сначала он инициализирует обе свои сетки и определяет своих соседей, `left` и `right`. Затем рабочие многократно обмениваются с соседями краями своих полос и обновляют свои точки. После `numIters` циклов обмена-обновления каждый рабочий отправляет строки своей полосы управляющему процессу, вычисляет максимальную разность между парами точек на своей полосе и, наконец, вызывает `MPI_Reduce`, чтобы отправить `mydiff` управляющему процессу.

Процесс `Coordinator` просто собирает результаты, отправляемые рабочими процессами. Сначала он получает строки окончательной сетки от всех рабочих. Затем вызывает под-

программу MPI_Reduce, чтобы получить и сократить максимальные разности, вычисленные каждым рабочим процессом. Заметим, что аргументы в вызовах MPI_Reduce одинаковы и в рабочих, и в управляющем процессах. Предпоследний аргумент COORDINATOR задает, что редукция должна происходить в управляющем процессе.

12.1.3. Учебный пример: OpenMP

OpenMP — это набор директив компилятора и библиотечных подпрограмм, используемых для выражения параллельности с разделением памяти. Прикладные программные интерфейсы (APIs) для OpenMP были разработаны группой, представлявшей основных производителей быстродействующего аппаратного и программного обеспечения. Интерфейс Фортрана был определен в конце 1997 года, интерфейс C/C++ — в конце 1998, но стандартизация обоих продолжается. Интерфейсы поддерживают одни и те же функции, но выражаются поразному из-за лингвистических различий между Фортраном, C и C++.

Интерфейс OpenMP в основном образован набором директив компилятора. Программист добавляет их в последовательную программу, чтобы указать компилятору, какие части программы должны выполняться параллельно, и задать точки синхронизации. Директивы можно добавлять постепенно, поэтому OpenMP обеспечивает распараллеливание существующего программного обеспечения. Эти свойства OpenMP отличают ее от библиотек Pthread и MPI, которые содержат подпрограммы, вызываемые из последовательной программы и компонуемые с нею, и требуют от программиста вручную распределять работу между процессами.

Ниже описано и проиллюстрировано использование OpenMP для Фортран-программ. Вначале представлена последовательная программа для метода итераций Якоби. Затем в нее добавлены директивы OpenMP, выражающие параллельность. В конце раздела кратко описаны дополнительные директивы и интерфейс C/C++.

В листинге 12.3 представлен эскиз последовательной программы для метода итераций Якоби. Ее синтаксис своеобразен, поскольку программа написана с использованием соглашений Фортрана по представлению данных с фиксированной точкой. Строки с комментариями начинаются с буквы c в первой колонке, а декларации и операторы — с колонки 7. Дополнительные комментарии начинаются символом !. Все комментарии продолжаются до конца строки.

Листинг 12.3. Последовательный метод итераций Якоби на Фортране

```
      program main          ! главная программа
      integer n,maxiters    ! общие данные
      common /idat/ n,maxiters
      считывание значений n и maxiters (не показано)
      call jacobi()
      stop
      end

      subroutine jacobi()       ! реализует метод итераций Якоби
      integer n,maxiters        ! повторная декларация переменных
      common /idat/ n,maxiters
      integer i,j,iters
      double precision grid(n,n), new(n,n)
      double precision maxdiff, tempdiff
      инициализировать grid и new (см. текст)

c главный цикл: обновить сетки maxiters раз
      do iters = 1,maxiters,2  ! цикл от 1 до maxiters с шагом 2
        do j = 2,n-1             ! обновить точки new
          do i = 2,n-1
```

```
          new(i,j) = (grid(i-1,j) + grid(i+1,j) +
           grid(i,j-1) +grid(i,j+1)) * 0.25
         enddo
       enddo
       do j = 2,n-1          ! обновить точки grid
         do i = 2,n-1
           grid(i,j) = (new(i-1,j) + new(i+1,j) +
           new(i,j-1) +new(i,j+1)) * 0.25
         enddo
       enddo
     enddo
c вычисление максимальной погрешности
     maxdiff = 0.0
     do j = 2,n-1
       do i = 2,n-1
         tempdiff = abs(grid(i,j)-new(i,j))
         maxdiff = max(maxdiff,tempdiff)
       enddo
     enddo
     return
     end
```

Последовательная программа состоит из двух подпрограмм: `main` и `jacobi`. В подпрограмме `main` считываются значения `n` (размер сетки с границами) и `maxiters` (максимальное число итераций), а затем вызывается подпрограмма `jacobi`. Значения данных хранятся в общей области памяти и, следовательно, неявно передаются из `main` в `jacobi`. Это позволяет `jacobi` распределять память для массивов `grid` и `new` динамически.

В подпрограмме `jacobi` реализован последовательный алгоритм, представленный выше в листинге 11.1. Основное различие между программами в листингах 12.3 и 11.1 обусловлено синтаксическим отличием псевдо-С от Фортрана. В Фортране нижняя граница каждой размерности массива равна 1, поэтому индексы внутренних точек матриц по строкам и столбцам принимают значения от 2 до `n-1`. Кроме того, Фортран сохраняет матрицы в памяти машины по столбцам, поэтому во вложенных циклах `do` сначала выполняются итерации по столбцам, а затем по строкам.

В OpenMP используется модель выполнения "разветвление-слияние" (fork-join). Вначале существует один поток выполнения. Встретив одну из директив `parallel`, компилятор вставляет код, чтобы разделить один поток на несколько подпотоков. Вместе главный поток и подпотоки образуют так называемое множество *рабочих потоков*. Действительное количество рабочих потоков устанавливается компилятором (по умолчанию) или определяется пользователем — либо статически с помощью переменных среды (environment), либо динамически с помощью вызова подпрограммы из библиотеки OpenMP.

Чтобы распараллелить программу с помощью OpenMP, программист сначала определяет части программы, которые могут выполняться параллельно, например циклы, и окружает их директивами `parallel` и `end parallel`. Каждый рабочий поток выполняет этот код, обрабатывая разные подмножества в пространстве итераций (для циклов, параллельных по данным) или вызывая разные подпрограммы (для программ, параллельных по задачам). Затем в программу добавляются дополнительные директивы для синхронизации потоков во время выполнения. Таким образом, компилятор отвечает за разделение потоков и распределение работы между ними (в циклах), а программист должен обеспечить достаточную синхронизацию.

В качестве конкретного примера рассмотрим следующий последовательный код, в котором внутренние точки `grid` и `new` инициализируются нулями.

```
do j = 2,n-1
  do i = 2,n-1
    grid(i,j) = 0.0
```

```
            new(i,j) = 0.0
          enddo
        enddo
```

Чтобы распараллелить этот код, добавим в него три директивы компилятора OpenM.

```
!$omp parallel do
!$omp& shared(n,grid,new), private(i,j)
      do j = 2,n-1
        do i = 2,n-1
          grid(i,j) = 0.0
          new(i,j) = 0.0
        enddo
      enddo
!$omp end parallel do
```

Каждая директива компилятора начинается с `!$omp`. Первая определяет начало параллельного цикла `do`. Вторая дополняет первую, что обозначено добавлением символа `&` к `!$omp`. Во второй директиве сообщается, что во всех рабочих потоках `n`, `grid` и `new` являются разделяемыми переменными, а `i` и `j` — локальными. Последняя директива указывает на конец параллельного цикла `do` и устанавливает точку неявной барьерной синхронизации.

В данном примере компилятор разделит итерации *внешнего* цикла `do` (по `j`) и назначит их рабочим процессам некоторым способом, зависящим от реализации. Чтобы управлять назначением, программист может добавить предложение `schedule`. В OpenMP поддерживаются различные виды назначения, в том числе по блокам, по полосам (циклически) и динамически (портфель задач). Каждый рабочий поток будет выполнять внутренний цикл `do` (по `i`) для назначенных ему столбцов.

В листинге 12.4 представлен один из способов распараллеливания тела подпрограммы `jacobi` с использованием директив OpenMP. Основной поток разделяется на рабочие потоки для инициализации сеток, как было показано выше. Однако `maxdiff` инициализируется в основном потоке. Инициализация `maxdiff` перенесена, поскольку ее желательно выполнить в одном потоке до начала вычислений максимальной погрешности. (Вместо этого можно было бы использовать директиву `single`, обсуждаемую ниже.)

После инициализации разделяемых переменных следует директива `parallel`, разделяющая основной поток на несколько рабочих. В следующих двух предложениях указано, какие переменные являются общими, а какие — локальными. Каждый рабочий выполняет главный цикл. В цикл добавлены директивы `do` для указания, что итерации внешних циклов, обновляющие `grid` и `new`, должны быть разделены между рабочими. Окончания этих циклов обозначены директивами `end do`, которые также обеспечивают неявные барьеры.

После главного цикла (который завершается одновременно всеми рабочими) используется еще одна директива `do`, чтобы максимальная погрешность вычислялась параллельно. В этом разделе `maxdiff` используется в качестве переменной редукции, поэтому к директиве `do` добавлено предложение `reduction`. Семантика переменной редукции такова, что каждое обновление является неделимым (в данном примере с помощью функции `max`). В действительности OpenMP реализует переменную редукции, используя скрытые переменные в каждом рабочем потоке; значения этих переменных "сливаются" неделимым образом в одно на неявном барьере в конце распараллеленного цикла.

Программа в листинге 12.4 иллюстрирует наиболее важные директивы OpenMP. Библиотека содержит несколько дополнительных директив для распараллеливания, синхронизации и управления рабочей средой (data environment). Например, для обеспечения более полного управления синхронизацией операторов можно использовать следующие директивы.

`critical`	Выполнить блок операторов как критическую секцию.
`atomic`	Выполнить один оператор неделимым образом.

`single`	В одном рабочем потоке выполнить блок операторов.
`barrier`	Выполнить барьер, установленный для всех рабочих потоков.

В OpenMP есть несколько библиотечных подпрограмм для запросов к рабочей среде и управления ею. Например, есть подпрограммы установки числа рабочих потоков и его динамического изменения, а также определения идентификатора потока.

Листинг 12.4. Параллельный метод итераций Якоби с использованием OpenMP

```
      subroutine jacobi()
      декларации общих, разделяемых и локальных переменных
      параллельная инициализация grid и new (см. текст)
      maxdiff = 0.0     ! инициализация в основном потоке
с старт рабочих потоков; каждый выполняет главный цикл
!$omp parallel
!$omp& shared(n,maxiters,grid,new,maxdiff)
!$omp& private(i,j,iters,tempdiff)
      do iters = 1,maxiters,2
!$omp do       ! разделение итераций внешнего цикла
        do j = 2,n-1
          do i = 2,n-1
            new(i,j) = (grid(i-1,j) + grid(i+1,j) +
                       grid(i,j-1) +grid(i,j+1)) * 0.25
          enddo
        enddo
!$omp end do       ! неявный барьер
!$omp do           ! разделение итераций внешнего цикла
        do j = 2,n-1
          do i = 2,n-1
            grid(i,j) = (new(i-1,j) + new(i+1,j) +
                        new(i,j-1) +new(i,j+1)) * 0.25
          enddo
        enddo
!$omp enddo        ! неявный барьер
      enddo              ! конец главного цикла

с вычисление максимальной погрешности в переменной редукции
!$omp do       ! разделение итераций внешнего цикла
!$omp$ reduction(max: maxdiff)   ! используется переменная редукции
      do j = 2,n-1
        do i = 2,n-1
          tempdiff = abs(grid(i,j)-new(i,j))
          maxdiff = max(maxdiff,tempdiff) ! неделимое обновление
        enddo
      enddo
!$omp end do    ! неявный барьер
!$omp end parallel   ! конец параллельнго раздела
      return
      end
```

Интерфейс OpenMP для C/C++ обеспечивает те же функции, что и интерфейс для Фортрана. Разница между ними обусловлена лингвистическими отличиями C/C++ от Фортрана. Например, директива параллельности `parallel` имеет следующий вид.

```
pragma omp parallel clauses
```

Ключевое слово `pragma` обозначает директиву компилятора. Поскольку в C вместо циклов `do` для определенного количества итераций используются циклы `for`, эквивалентом директивы `do` в C является

```
pragma omp for clauses.
```

В интерфейсе C/C++ нет директивы `end`. Вместо нее блоки кода заключаются в фигурные скобки, обозначающие область действия директив.

12.2. Распараллеливающие компиляторы

Главной темой этой книги является создание явно параллельных программ, т.е. программ, в которых программист должен определить процессы, разделить между ними данные и запрограммировать все необходимые взаимодействия и синхронизацию. Явно параллельная программа — вот что в конечном счете может выполняться на многопроцессорных машинах. Однако существуют и другие подходы к разработке параллельных программ. В данном разделе рассматриваются распараллеливающие компиляторы, которые преобразуют последовательные программы в параллельные. В следующем разделе описываются высокоуровневые подходы на основе языков с поддержкой параллельных данных и функциональных языков.

При распараллеливании программы необходимо определить задачи, которые не зависят друг от друга и, следовательно, могут выполняться одновременно. Некоторые программы являются *существенно параллельными*, т.е. содержащими огромное число независимых задач, например, умножение матриц или вычисление множеств Мандельброта. Однако большинство программ состоят из частей, которые зависят друг от друга и требуют периодической синхронизации при выполнении. Например, взаимодействующим процессам необходимо синхронизироваться в программе типа "производитель-потребитель". Или процессам в сеточных вычислениях нужна барьерная синхронизация после каждой фазы обновлений.

Цель распараллеливающего компилятора — получить последовательную программу и создать корректную параллельную. Для этого он выполняет так называемый *анализ зависимости*. Компилятор определяет, какие части последовательной программы независимы (и являются кандидатами на параллельное выполнение), а какие зависят друг от друга и требуют последовательного выполнения или какой-то другой формы синхронизации. Поскольку большинство вычислений в последовательной программе происходит внутри циклов, особенно вложенных, основным для компилятора является распараллеливание циклов.

В этом разделе определяются различные виды зависимости по данным и демонстрируется, как компилятор выполняет анализ зависимости. Затем рассматриваются некоторые наиболее распространенные преобразования последовательных циклов в параллельные. Подробнее они описаны в литературе, указанной в исторической справке в конце главы.

Распараллеливающие компиляторы постоянно совершенствуются и в настоящее время вполне пригодны для создания эффективных параллельных программ с разделяемыми переменными. Особенно это касается научных программ, содержащих много циклов и длительных вычислений. Однако создать хорошую программу с обменом сообщениями гораздо сложнее, поскольку всегда существует много вариантов структуры программы и взаимодействия, как было показано в главе 11. Кроме того, некоторые сложные последовательные алгоритмы трудно распараллелить, например, многосеточные методы или метод Барнса–Хата.

12.2.1. Анализ зависимости

Предположим, что последовательная программа содержит два оператора S_1 и S_2, причем S_1 находится перед S_2. Говорят, что между двумя операторами существует *зависимость по*

данным, если они считывают или записывают данные в общей области памяти так, что порядок их выполнения нельзя изменять. Существует три основных типа зависимости по данным.[26]

1. *Потоковая зависимость*. Оператор S_2 потоково зависит от S_1, если S_2 считывает из ячейки, в которую записывает S_1. (Такая зависимость еще называется *истинной*.)
2. *Антизависимость*. Оператор S_2 является антизависимым относительно S_1, если S_2 записывает в ячейку, из которой S_1 считывает.
3. *Зависимость по выходу*. Оператор S_2 зависит по выходу от S_1, если оператор S_2 записывает данные в ту же ячейку памяти, что и S_1.

Будем просто говорить, что S_2 *зависит от* S_1, если это зависимость по данным; тип зависимости не важен.

В качестве примера рассмотрим следующую последовательность операторов.

```
S1: a = b + d;
S2: c = a * 3;
S3: a = b + c;
S4: e = a / 2;
```

Оператор S_2 потоково зависит от S_1, поскольку считывает a. Оператор S_3 антизависим относительно S_2, поскольку он записывает a; S_3 также зависит по выходу от S_1, поскольку они оба записывают a. Наконец, оператор S_4 потоково зависит от S_3, поскольку считывает a. (S_4 также потоково зависит от S_1, но S_1 должен выполняться перед S_3.) Вследствие этих зависимостей операторы должны выполняться в порядке, указанном в списке; изменение порядка операторов приведет к изменению результатов.

Зависимости по данным легко определить в последовательном коде, содержащем ссылки только на скалярные переменные. Намного сложнее определить зависимости в циклах и при ссылках в массивы (которые обычно встречаются вместе), поскольку ссылки в массивы имеют индексы, а в индексах обычно используются параметры циклов, т.е. индексы массива имеют различные значения на разных итерациях цикла. В действительности общая проблема вычисления всех зависимостей по данным в программе неразрешима из-за применения синонимов имени массива, которое может возникнуть при использовании указателей или вызовов функций внутри индексных выражений. Даже если указатели не разрешены, как в Фортране, и индексные выражения являются линейными функциями, проблема является NP-трудной, т.е. эффективного алгоритма для нее, по всей вероятности, нет.

Чтобы лучше понять проблемы, создаваемые циклами и массивами, рассмотрим еще раз код прямого хода в решении системы уравнений (см. листинг 11.12).

```
for [i = 1 to n] {
 S1: sum = 0.0;
     for [j = 1 to i-1]
 S2:    sum = sum +LU[ps[i],j] * x[j];
 S3: x[i] = b[ps[i]] - sum;
}
```

В каждой итерации внешнего цикла S_2 зависит от S_1, а S_3 зависит и от S_1, и от S_2. Внутренний цикл состоит из одного оператора, поэтому никакой зависимости в этом цикле нет. Однако внутренний цикл *приводит* к зависимости S_2 от самого себя, поскольку S_2 и считывает, и записывает sum. Аналогично и внешний цикл создает зависимость: S_2 зависит от S_3, поскольку значение, записанное в x[i] на одной итерации внешнего цикла, считывается на всех последующих.

[26] Четвертый тип зависимости — по входу; она обычно возникает, когда два оператора считывают данные из одной и той же ячейки памяти. Зависимость по входу не ограничивает порядок выполнения операторов.

Проверка зависимости представляет собой задачу определения, есть ли зависимость по данным в произвольной паре индексированных ссылок. В общем виде задача ставится для вложенного цикла следующего вида.

```
for [i1 = l1 to u1] { ...
    for[in = ln to un] {
S1: a[f1, ..., fn] = ...;
S2: ... = a[g1, ..., gn];
    } ... }
```

Есть n вложенных циклов и, соответственно, n индексных переменных. Нижние и верхние границы n индексных переменных определяются функциями l_j и u_j. Наиболее глубоко вложенный цикл состоит из двух операторов, содержащих ссылки на элементы n-мерного массива a; первый оператор записывает в a, второй — считывает из a. Вызовы функций f_j и g_j в этих операторах содержат в качестве своих аргументов индексные переменные; функции возвращают значения индексов.

Итак, возникает следующий вопрос о зависимости в данном цикле: существуют ли значения индексных переменных, при которых $f_1 = g_1$, $f_2 = g_2$ и т.д.? Ответ определяет, зависит S_2 от S_1 на одной и той же итерации цикла или между операторами есть зависимости, создаваемые циклом.

Проверку зависимости можно представить в виде специальной системы линейных уравнений и неравенств следующего вида.

$$A_1 x = b_1 \quad \text{и} \quad A_2 x \leq b_2$$

Коэффициенты в матрицах A_1 и A_2 определяются функциями f и g в программе, а значения в векторах b_1 и b_2 — границами для индексных переменных. Решением первого уравнения является присваивание значений индексным переменным, при котором ссылки массива перекрываются. Второе уравнение (в действительности система неравенств) обеспечивает, что значения индексных переменных находятся внутри границ, определяемых функциями l и u. Решение данной задачи похоже на решение двух систем линейных уравнений, рассмотренное в разделе 11.3, однако имеет два принципиальных отличия: 1) решение должно быть вектором целых, а не действительных чисел, 2) второе выражение имеет соотношение "меньше или равно". Именно эти отличия приводят к тому, что проблема становится NP-трудной, даже если индексные выражения являются линейными функциями и не содержат указателей. (Проверка зависимости при этих условиях эквивалентна частному случаю задачи целочисленного линейного программирования.)

Хотя проверка зависимости является сложной (а в худшем случае — неразрешимой) задачей, существуют эффективные проверки, применимые в некоторых частных случаях. В действительности почти для всех циклов, встречающихся на практике, можно определить, перекрываются ли две ссылки в массив. Например, эффективные проверки существуют для ситуации, при которой все границы циклов являются константами, или границы внутренних циклов зависят только от границ внешних циклов. Цикл прямого хода в методе исключений Гаусса, приведенный выше, удовлетворяет данным ограничениям: границы для i — константы, одна граница для j — константа, а другая зависит от значения i. Но если компилятор не может определить, отличаются ли две ссылки на элементы массива, то он пессимистически предполагает, что зависимость есть.

12.2.2. Преобразования программ

Проверка зависимости является первым этапом в работе распараллеливающего компилятора. Второй шаг — распараллеливание циклов с использованием результатов первого этапа. Ниже на нескольких примерах показано, как это происходит.

В первом примере предположим, что функция f не имеет побочного эффекта, и рассмотрим следующий вложенный цикл.

```
for [i = 1 to n]
  for [j = 1 to n]
    a[i,j] = a[i,j] + f(i,j);
```

Здесь нет зависимостей по данным, поскольку ссылки на элементы массива не перекрываются. Следовательно, присваивания a можно выполнять в произвольном порядке. Когда распараллеливающий компилятор встречает оператор такого типа, он, скорее всего, распараллелит внешний цикл следующим образом.[27]

```
co [i = 1 to n]
  for [j = 1 to n]
    a[i,j] = a[i,j] + f(i,j);
```

В результате появится n процессов, каждый из которых присвоит строке матрицы a.

Если матрица хранится в памяти по столбцам, а не по строкам, то эффективнее параллельно обновлять столбцы. Однако в последовательной программе просто распараллелить внутренний цикл вместо внешнего неэффективно. (Пришлось бы многократно создавать и уничтожать процессы, состоящие только из одного оператора присваивания.) Вместо этого можно сначала поменять местами два цикла `for` и получить эквивалентную программу.

```
for [j = 1 to n]
  for [i = 1 to n]
    a[i,j] = a[i,j] +f(i,j);
```

Теперь можно распараллелить внешний цикл, используя оператор co.

Перестановка циклов — это один из видов преобразования программ, используемых в распараллеливании. Ниже рассматриваются еще несколько полезных преобразований: локализация, расширение скаляра, распределение цикла, слияние циклов, развертка и сжатие, развертка цикла, разделение на полосы, разделение на блоки, а также перекос цикла (рис. 12.1). Они помогают выявлять параллельность, устранять зависимости и оптимизировать использование памяти. Рассмотрим их.

Перестановка циклов	Внешний и внутренний циклы меняются местами
Локализация	Каждому процессу дается копия переменной
Расширение скаляра	Скаляр заменяется элементом массива
Распределение цикла	Один цикл расщепляется на два отдельных цикла
Слияние циклов	Два цикла объединяются в один
Развертка и сжатие	Комбинируются перестановка циклов, разделение на полосы и развертка
Развертка цикла	Тело цикла повторяется и выполняется меньше итераций
Разделение на полосы	Итерации одного цикла разделяются на два вложенных цикла
Разделение на блоки	Область итераций разбивается на прямоугольные блоки
Перекос цикла	Границы цикла изменяются, чтобы выделить параллельность фронта волны

Рис. 12.1. Преобразования программ, используемые параллельными компиляторами

Рассмотрим стандартную последовательную программу вычисления матричного произведения c двух квадратных матриц a и b размером n×n.

```
for [i = 1 to n]
  for [j = 1 to n] {
```

[27] Для задания параллельных циклов в этой книге используется оператор co. В других языках используются подобные операторы, например `parallel do` или `for all`.

```
    sum = 0.0;
    for[k = 1 to n]
      sum = sum + a[i,k] * b[k,j];
    c[i,j] = sum;
  }
```

Три оператора в теле второго цикла (по j) зависят друг от друга, поэтому должны выполняться последовательно. Два внешних цикла независимы в действиях с матрицами, поскольку a и b только считываются, а каждый элемент c встречается только один раз. Однако все три цикла создают зависимости, поскольку sum — одна скалярная переменная. Можно распараллелить оба внешних цикла или любой из них, если *локализовать* sum, т.е. дать каждому процессу собственную копию этой переменной. Таким образом, локализация является преобразованием, устраняющим зависимости.

Другой способ распараллелить программу умножения матриц — применить *расширение скаляра*. Одиночная переменная заменяется элементом массива. В данном случае можно изменить переменную sum на c[i,j]. Это преобразование также позволяет избавиться от последнего оператора присваивания и получить следующую программу.

```
for [i = 1 to n]
  for [j = 1 to n] {
    c[i,j] = 0.0;
    for[k = 1 to n]
      c[i,j] = c[i,j] + a[i,k] * b[k,j];
  }
```

Два внешних цикла больше не создают зависимости, поэтому теперь можно распараллелить цикл по i, выполнить перестановку циклов и распараллелить цикл по j или распараллелить оба цикла.[28]

Наиболее глубоко вложенный цикл в данной программе зависит от инициализации массива c[i,j]. Однако элементы массива c можно инициализировать в любом порядке — лишь бы все они инициализировались до начала их использования в вычислениях. Чтобы отделить инициализацию от вычисления произведений во внутреннем цикле, можно применить еще одно преобразование цикла — *распределение*. При распределении цикла независимые операторы, записанные в теле одного цикла (или вложенных циклов), помещаются в отдельные циклы с одинаковыми заголовками, как в следующем примере.

```
for [i = 1 to n]  # инициализация c[*,*]
  for [j = 1 to n] {
    c[i,j] = 0.0;

for [i = 1 to n]  #вычисление сумм произведений
  for [j = 1 to n] {
    for[k = 1 to n]
      c[i,j] = c[i,j] + a[i,k] * b[k,j];
```

Теперь для распараллеливания каждого внешнего цикла (по i) можно использовать co. При альтернативном подходе внешние циклы можно объединить и использовать директиву co только один раз, установив между внутренними циклами точку барьерной синхронизации следующим образом.

```
co [i = 1 to n]  {
  for [j = 1 to n]
    c[i,j] = 0.0;
  BARRIER;
  for [j = 1 to n] {
```

[28] В данном примере локализация была бы эффективнее, чем замена скаляра. Во-первых, sum могла бы сохраняться в регистре. Во-вторых, ссылки на c[i,j] в полученном коде могут привести к ложному разделению переменных и, как следствие, к слишком непроизводительному использованию кэша.

```
        for[k = 1 to n]
          c[i,j] = c[i,j] + a[i,k] * b[k,j];
    }
```

Выбор между приведенными подходами определяется тем, что дешевле — создавать процессы дважды или создать их один раз и использовать барьер. Как правило, барьерная синхронизация эффективнее, чем создание процессов.

В приведенном примере распределение цикла увеличивает число процессов и делает их мельче, т.е. каждый из них выполняет меньше работы. Распределение цикла обычно используется для устранения зависимости и, следовательно, для выявления параллельности. Предположим, например, что цикл имеет следующий вид.

```
for [i = 1 to n] {
  a[i] = ...;
  ...  = ... + a[i-1];
}
```

Здесь второй оператор зависит от результата, созданного первым оператором на предыдущей итерации. Если в остальном операторы независимы, цикл можно распределить так, чтобы первое присваивание выполнялось в первом цикле, а второе — во втором.

```
for [i = 1 to n]
  a[i] = ...;
for [i = 1 to n]
  ...  = ... + a[i-1];
}
```

Тогда оба цикла можно распараллелить (при условии, что других зависимостей нет).

Слияние циклов обратно распределению циклов. В этом преобразовании циклы, имеющие одинаковые заголовки и независимые тела, объединяются (сливаются) в один цикл. Слияние циклов приводит к уменьшению числа процессов и их укрупнению (каждый выполняет больше работы).

Развертка и сжатие — это еще одно преобразование, изменяющее зависимости. Оно также приводит к тому, что доступы к одним и тем же ячейкам памяти располагаются ближе друг к другу. Это может повысить производительность за счет повторного использования регистров или кэш-памяти и повышения эффективности параллельности на уровне команд в многозадачных процессорах. Вернемся к циклам умножения матриц, вычисляющим скалярные произведения.

```
for [i = 1 to n]
  for [j = 1 to n]
    for[k = 1 to n]
      c[i,j] = c[i,j] + a[i,k] * b[k,j];
```

При развертывании и сжатии сначала *разворачивается* внешний цикл, затем полученные циклы *сжимаются* (сливаются), чтобы операторы умножения оказались в наиболее глубоко вложенном цикле. Применив развертку и сжатие к приведенному выше коду умножения матриц и развернув внешний цикл один раз, получим такой код.

```
for [i = 1 to n by 2]  # половина всех итераций
  for [j = 1 to n]
    for[k = 1 to n] {
      c[i,j] = c[i,j] + a[i,k] * b[k,j];
      c[i+1,j] = c[i+1,j] * b[k,j];
    }
```

Каждый раз во внешнем цикле вычисляются два скалярных произведения. Они запоминаются в `c[i,j]` и `c[i+1,j]`, которые будут смежными ячейками памяти, если матрицы хранятся по столбцам.

Разделение на полосы — простое преобразование, которое делит один цикл на два; во внешнем цикле перебираются полосы, а внутренний цикл повторяется для всех элементов полосы. Например, в программе умножения матриц развертывание внешнего цикла один раз эквивалентно разделению на полосы ширины два.

```
for [i = 1 to n by 2]  # половина всех итераций
 for [I = i to i+1]    # полоса из двух итераций
  for [j = 1 to n]
    for[k = 1 to n]
      c[I,j] = c[I,j] + a[I,k] * b[k,j];
```

Заметим, что оператор присваивания использует `I` вместо `i`. Разделение на полосы регулирует мелкомодульность циклов и используется вместе с другими преобразованиями, о которых речь пойдет ниже.

Преобразование развертки и сжатия эквивалентно разделению на полосы внешнего цикла, перемещению разделенного цикла на место внутреннего цикла и его полной развертке. В нашем примере нужно сначала сделать цикл по `I` внутренним.

```
for [i = 1 to n by 2]  # половина всех итераций
  for [j = 1 to n]
    for[k = 1 to n]
      for [I = i to i+1]  # полоса из двух итераций
        c[I,j] = c[I,j] + a[I,k] * b[k,j];
```

Развернув теперь внутренний цикл, т.е. заменив `I` двумя ее значениями `i` и `i+1`, получим такую же программу, как после развертки и сжатия.

Разделение на блоки — это еще одно преобразование, которое может улучшить размещение данных в памяти и, соответственно, сделать более эффективным использование кэша на машинах с разделяемой памятью или размещение данных в распределенной памяти. Рассмотрим следующий код, в котором матрица `a` транспонируется в матрицу `b`.

```
for [i = 1 to n]
  for [j = 1 to n]
    b[i,j] = a[j,i];
```

При условии, что массивы хранятся в памяти по строкам, доступ к элементам `b` осуществляется с *шагом* 1, т.е. каждый элемент `b` размещается рядом с предыдущим использованным элементом. Однако доступ к элементам `a` имеет шаг `n`, т.е. равен длине строки. Таким образом, если строки кэша содержат более одного значения, то при ссылках на `a` в кэш загружается много значений, которые не используются.

Разделение на блоки делит область итераций на квадраты или прямоугольники, стороны которых равны ширине строки кэша. Если строка кэша содержит `W` значений `a` или `b`, код транспонирования матрицы можно изменить следующим образом.

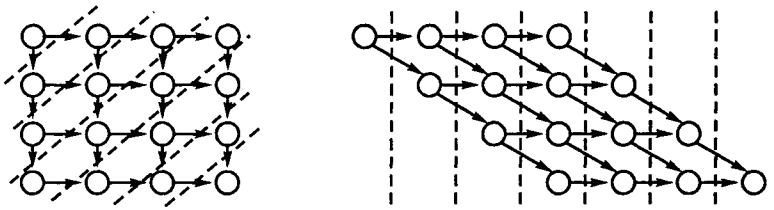
```
for [i = 1 to n by W]     #для каждой W-й строки
  for [j = 1 to n by W]   # и столбца использовать
    for [I = i to min(i+W-1, n)] # блок размером W
      for [J = j to min(j+W-1, n)]
        b[I,J] = a[J,I];
```

Теперь в двух внутренних циклах доступен квадрат из `W×W` элементов. Заметим, что разделение цикла на блоки является, по существу, комбинацией разделения на полосы и перестановки циклов.

Наконец, рассмотрим *перекос цикла*. Это преобразование используется для выделения параллельности, особенно вдоль волновых фронтов. При вычислениях типа волновых фронтов обновления в массивах распространяются подобно волне. Волновые фронты уже встречались в прямом и обратном ходе LU-разложения. Еще один пример вычислений типа волновых фронтов — метод Гаусса–Зейделя для решения дифференциальных уравнений в частных производных (раздел 11.1).

```
for [i = 1 to n]
  for [j = 1 to n]
    a[i,j] = (a[i-1,j] + a[i,j-1] +
              a[i+1,j] + a[i,j+1]) * 0.25;
```

Каждое новое значение a зависит от двух значений, вычисленных в предыдущих итерациях, и от двух значений, которые не обновляются до следующих итераций. На рис. 12.2, *а* показаны эти зависимости; пунктирными линиями обозначены волновые фронты.



а) исходное пространство б) после сдвига цикла

Рис. 12.2. Зависимости по данным в методе итераций Гаусса–Зейделя

Перекос цикла не изменяет выполняемых вычислений, а просто смещает границы цикла так, чтобы волновые фронты устанавливались не по диагоналям, а по столбцам (рис. 12.2, *б*). Перекос осуществляется за счет того, что к граничным значениям индексной переменной внутреннего цикла прибавляется число, кратное значению индексной переменной внешнего цикла, а затем в теле внутреннего цикла из его индексной переменной при каждом ее использовании это же число вычитается. Сомножитель, образующий кратное, называется *сомножителем перекоса*. Программа для метода Гаусса–Зейделя с сомножителем перекоса, равным 1, принимает следующий вид.

```
for [i = 1 to n]
  for [j = i+1 to i+n]  # прибавить i к имеющимся границам
    a[i,j-i] = (a[i-1,j-i] + a[i,j-i-1] +
                a[i+1,j-i] + a[i,j-i+1]) * 0.25;
```

Перекос цикла позволяет выявить параллельность, поскольку никаких зависимостей в каждом столбце области итераций больше нет. Однако, чтобы использовать параллельность цикла по i, нужно переставить циклы. (Как сказано выше, внешний цикл создает зависимости.)

Перестановку сдвинутых циклов выполнить непросто, поскольку границы внутреннего цикла зависят от значений индексной переменной внешнего цикла. Однако границы исходных циклов независимы, поэтому границы для переставленных сдвинутых циклов можно вычислить следующим образом. Пусть LI и UI — нижняя и верхняя границы исходного внешнего цикла, LJ и UJ — нижняя и верхняя границы исходного внутреннего цикла, а f — сомножитель перекоса. После перестановки границы сдвинутых циклов изменятся, как показано в следующем коде.

```
for [j = (f*LI + LJ) to (f*UI + UJ)]
  for [i = max(LI, ceil((j-UJ)/f)) to
                   min(UI, ceil(j-LJ)/f))]
```

Применение этих формул к программе метода Гаусса–Зейделя дает такой код.

```
for [j = 2 to n+n]
  for [i = max(1, j-n) to min(n, j-1)]
    a[i,j-i] = (a[i-1,j-i] + a[i,j-i-1] +
                a[i+1,j-i] + a[i,j-i+1]) * 0.25;
```

Теперь внутренний цикл можно распараллелить одним из двух способов — либо использовать оператор co, создавая процессы на каждой итерации внешнего цикла, либо создать процессы один раз, чтобы для синхронизации волн они использовали флажки. Оба подхода не столь эффективны, как распараллеливание внешнего цикла, но параллельность волнового фронта полезна, если n велико или машина непосредственно поддерживает мелкомодульный параллелизм.

12.3. Языки и модели

Большинство эффективных параллельных программ написаны с помощью последовательного языка (как правило, C или Фортран) и библиотеки. Для этого есть несколько причин. Во-первых, программисты хорошо знают какой-нибудь последовательный язык и имеют опыт написания научных программ. Во-вторых, библиотеки совместимы с обычно используемыми параллельными вычислительными платформами. В-третьих, высокая производительность — основная цель быстродействующих вычислений, а библиотеки создаются под конкретное аппаратное обеспечение, предоставляя программисту управление на низком уровне.

Однако использование языка высокого уровня, содержащего механизмы как последовательного, так и параллельного программирования, тоже имеет ряд преимуществ. Во-первых, язык обеспечивает более высокий уровень абстракции, что может помочь программисту ориентироваться в решении задачи. Во-вторых, последовательные и параллельные механизмы можно объединять, чтобы они работали вместе, а аналогичные алгоритмы имели сходное выражение. Эти два свойства облегчают создание и понимание программ (как правило, относительно коротких). В-третьих, язык высокого уровня обеспечивает контроль типов, освобождая программиста от необходимости проверять соответствие типов данных (об этом можно даже и не вспоминать). Это оберегает программиста от многих ошибок, позволяет создавать более короткие и устойчивые программы. Основные проблемы разработчиков языка — разработать хороший набор абстракций, мощный и ясный набор механизмов, а также эффективную реализацию.

Для параллельного программирования разработаны различные языки. На рис. 12.3 перечислены языки, которые часто используются или воплощают важные идеи. Языки на рисунке сгруппированы в классы в соответствии с лежащей в их основе моделью программирования: разделяемые переменные, передача сообщений, координация, параллельность по данным и функциональность. Языки первых трех классов используются для создания императивных программ, в которых процессы, взаимодействие и синхронизация программируются явно. Языки с параллельностью по данным имеют неявную синхронизацию, а функциональные — неявные параллельность и синхронизацию. Последняя группа языков на рис. 12.3 содержит три абстрактные модели, которые можно использовать для сравнения алгоритмов и оценки производительности.

Некоторые языки, перечисленные на рис. 12.3, были описаны в предыдущих главах книги. В данном разделе представлен краткий обзор новых, не рассмотренных до сих пор, аспектов языков и моделей и дается более подробное описание быстродействующего Фортрана (HPF). В исторической справке в конце главы содержатся ссылки на источники дальнейшей информации по конкретным языкам, а также описываются книги и обзорные статьи, дающие общее представление о языках и моделях.

12.3.1. Императивные языки

Языки первых трех классов на рис. 12.3 используются для создания императивных программ, в которых явно обновляется и синхронизируется доступ к состоянию программы, т.е. к значениям переменных. Все эти языки имеют средства для определения процессов, задач или потоков, которые отличаются способом программирования взаимодействия и синхронизации. По первому способу для взаимодействия применяются разделяемые переменные, а для синхронизации — разделяемые блокировки, флаги, семафоры или мониторы. По второму — и для взаимодействия, и для синхронизации используются сообщения. Некоторые языки поддерживают один из этих способов, некоторые — оба, предоставляя программисту выбор в зависимости от решаемой задачи или архитектуры машины, на которой будет работать программа. Третий способ заключается в обеспечении области разделяемых данных и операций над ними, подобных сообщениям (см. обсуждение языков с координацией ниже).

Учебные примеры многих языков, поддерживающих явное параллельное программирование, уже рассматривались. Три из них (Ada, Java и SR) на рис. 12.3 указаны дважды, по-

скольку поддерживают и разделяемые переменные, и обмен сообщениями. Ada (раздел 8.6) обеспечивает задачи и защищенные типы для программирования мониторов, а также рандеву для распределенных программ. Java поддерживает потоки и синхронизированные методы для программирования мониторов (раздел 5.4), предопределенный языковой модуль для обмена сообщениями с помощью сокетов и дейтаграмм (раздел 7.9) и еще один модуль для удаленного вызова методов (см. раздел 8.5). SR (раздел 8.7) поддерживает процессы, оператор co для рекурсивного параллелизма, семафоры для синхронизации разделяемых переменных и множество механизмов передачи сообщений — асинхронный, синхронный, удаленный вызов процедур (RPC) и рандеву. Четвертая пара языков, CSP/Occam (раздел 7.6), поддерживает синхронную передачу сообщений, но не разделяемые переменные.

Ниже дается обзор языков Cilk и Фортран M. Первый интересен для программирования с разделяемыми переменными, второй — с обменом сообщениями.

Разделяемые переменные	
Ada	Защищенные типы
Cilk	Разветвление-слияние процессов
Java	Синхронизированные методы
SR	Разветвление-слияние процессов и семафоры
Обмен сообщениями	
Ada	Рандеву
CPS/Occam	Синхронная передача сообщений
Fortran M	Асинхронная передача сообщений
Java	Модули сетевого и удаленного вызова
SR	Передача сообщений, RPC, рандеву
Координация	
Linda	Пространство кортежей и примитивы, подобные сообщениям
Orca	Объекты данных и удаленные операции
Параллельность по данным	
C*	C, макет данных и параллельное выполнение
HPF	Отображения данных, операторы с массивами, редукции
NESL	Вложенная параллельность по данным
ZPL	Области данных и направления, операции с массивами
Функциональность	
NESL	Рекурсивный параллелизм
Sisal	Итеративный (для всех) и рекурсивный параллелизм
Абстрактные модели	
BSP	Массовая синхронная передача сообщений
LogP	Процессоры с распределенной памятью
PRAM	Параллельный прямой доступ к разделяемым переменным

Рис. 12.3. Языки и модели параллельного программирования

Cilk

Язык Cilk расширяет C пятью простыми механизмами для параллельного программирования. Cilk хорошо подходит для рекурсивного параллелизма, хотя также поддерживает параллельность по задачам и по данным. Cilk разрабатывался для эффективного выполнения на симметричных мультипроцессорах с разделяемой памятью.

Новшеством в Cilk является то, что если из Cilk-программы убрать специфические для Cilk конструкции, то оставшийся код будет синтаксически и семантически корректной C-программой. Пять механизмов Cilk обозначаются ключевыми словами `cilk`, `spawn`, `synch`, `inlet` и `abort`. Ключевое слово `cilk` добавляется в начало декларации C-функции и указывает, что функция является Cilk-процедурой. В такой процедуре при выполнении оператора `spawn` происходит разделение процессов. Порожденные потоки выполняются параллельно с родительским, вызывающим потоком. Для ожидания, пока все порожденные потоки завершат вычисления и возвратят результаты, в родительском потоке используется оператор `synch`. Таким образом, последовательность операторов `spawn`, за которой следует оператор `synch`, в сущности, является оператором `co/oc`.

Следующий пример иллюстрирует реализацию (неэффективную) рекурсивной параллельной программы для вычисления n-го числа Фибоначчи.

```
cilk int fib(int n) {
  if (n < 2)
    return n;
  else {
    int x, y;
     x = spawn fib(n-1);
     y = spawn fib(n-2);
     synch;
     return (x + y);
  }
}
```

Отметим, что удаление трех механизмов Cilk (`cilk`, `spawn` и `synch`) из приведенной программы дает функцию C.

`Inlet` (вход) является внутренней C-функцией процедуры Cilk. Она используется в таких вычислениях, как редукция при возвращении из порожденных вызовов. Например, можно добавить `inlet` в функцию `fib` следующим образом.

```
cilk int fib(int n) {
  int x = 0;

  inlet void plusReduce (int result)
     { x += result; return; }

  if (n < 2)
    return n;
  else {
    plusReduce(spawn fib(n-1));
    plusReduce(spawn fib(n-2));
    synch;
    return (x);
  }
}
```

Когда каждый порожденный экземпляр `fib` возвращает результат, вызывается вход `plusReduce`, который выполняется *неделимым образом*; он складывает результаты, возвращаемые из `fib`, редуцируя переменную `x`.

Последний механизм Cilk, `abort`, используется внутри входов; он уничтожает потоки, порожденные одним и тем же родителем, которые еще выполняются. Одним из применений механизма `abort` может служить программа рекурсивного поиска, в которой многие пути исследуются параллельно; программист может уничтожить поток, исследующий путь, если уже найдено лучшее решение.

Новизной реализации Cilk является так называемый *диспетчер захвата работы*. Cilk реализует две версии каждой Cilk-процедуры: быстрая, работающая, как обычная С-функция, и медленная, которая полностью поддерживает параллельность со всеми сопутствующими накладными расходами. При порождении процедуры выполняется быстрая версия. Если некоторому рабочему процессу (который выполняет Cilk-потоки) нужна работа, он захватывает процедуру от какого-либо другого рабочего процесса. В этот момент процедура преобразуется в медленную версию.

Фортран М

Этот язык является небольшим набором расширений Фортрана, поддерживающим модульную разработку программ с обменом сообщениями. Механизмы обмена сообщениями аналогичны механизмам, описанным в главе 7. Хотя этот язык больше не поддерживается, его возможности включены в MPI-связывание для языка HPF.

Процесс в Фортране М похож на процедуру с параметрами, но имеет ключевое слово `process` вместо `procedure`. Оператор `processcall` используется для создания нового процесса. Такие вызовы встречаются в частях программы, отмеченных операторами `processes/endprocesses`, которые подобны оператору `co/oc`.

Процессы в Фортране М взаимодействуют друг с другом с помощью портов и каналов; они не могут разделять переменные. Канал представляет собой очередь сообщений. Он создается с помощью оператора `channel`, который определяет пару портов — вывода и ввода. Оператор `send` добавляет сообщение в порт вывода. Оператор `endchannel` добавляет в порт вывода сообщение, отмечающее конец канала. Оба оператора являются неблокирующими (асинхронными). Оператор `receive` ожидает сообщения в порту ввода, затем удаляет сообщение (таким образом, `receive` является блокирующим).

Каналы можно создавать и уничтожать динамически, передавать в качестве аргументов процессам и отправлять в сообщениях как значения. Например, чтобы позволить двум процессам взаимодействовать друг с другом, программисту сначала нужно создать два канала, затем — два процесса и передать им каналы как аргументы. Первый процесс использует порт вывода одного канала и порт ввода другого; второй — другие порты этих двух каналов.

Фортран М главным образом предназначен для программирования параллельности по задачам, в котором процессы выполняют, вообще говоря, разные задачи. Однако с помощью распределенных массивов язык поддерживает и программирование, параллельное по данным. Операторы распределения данных похожи на операторы, поддерживаемые HPF, который обсуждается в конце раздела 12.3.

12.3.2. Языки с координацией

Язык с координацией является расширением какого-либо основного языка с разделяемой областью данных и примитивами, подобными сообщениям, для обработки области данных. Идея таких языков происходит от набора примитивов Linda (раздел 7.7). Напомним, что Linda расширяет основной язык, например С, с помощью абстракции ассоциативной памяти, называемой *пространством кортежей* (ПК). ПК концептуально разделяется всеми процессами. Новый кортеж помещается в ПК при выполнении примитива `out`, извлекается оттуда с помощью `in` и проверяется при выполнении `rd`. Наконец, с помощью примитива `eval` процессы создаются; завершая работу, процесс помещает возвращаемое им значение в ПК. Таким образом, в Linda комбинируются аспекты разделяемых переменных и асинхронного обмена сообщениями — пространство кортежей разделяется всеми процессами, кортежи помещаются и извлекаются неделимым образом, как будто они — сообщения.

Orca

Это более современный пример языка с координацией. Подобно Linda, он основан на структурах данных, которые концептуально разделяются, хотя физически могут быть распределенными. Объединяющим понятием в Orca является не разделяемый кортеж, а разделяемый объект данных. Процессы на разных процессорах могут совместно использовать пассивные объекты — экземпляры абстрактных типов. Процессы получают доступ к разделяемому объекту, вызывая операцию, которая определена объектом. Операции реализуются с помощью механизма, сочетающего аспекты RPC и рандеву.

С помощью примитива `fork` процесс в Orca создает еще один процесс. Параметры процесса могут быть значениями (`value`) или разделяемыми (`shared`). Для параметра-значения родительский процесс создает копию аргумента и передает ее процессу-сыну. Для разделяемого параметра аргумент должен быть переменной, которая является объектом типа, определяемого пользователем. После создания процесса-сына он и родитель разделяют этот объект.

Каждый определенный пользователем тип описывает операции над объектами этого типа. Каждый экземпляр разделяемого объекта является монитором или серверным процессом, в зависимости от реализации. В обоих случаях операции над объектом выполняются неделимым образом. Реализация операции в Orca может состоять из одного или нескольких защищенных операторов. В Orca поддерживается синхронизация условий с помощью булевых выражений, а не условных переменных.

В первоначальной версии Orca поддерживалась только параллельность по задачам, в современной — также и по данным. Это реализовано с помощью разбиения на части объектов данных, основанных на массивах, распределения этих частей по разным процессорам и их обработки с использованием параллельных операций, определенных пользователем.

12.3.3. Языки с параллельностью по данным

Языки с параллельностью по данным непосредственно поддерживают стиль программирования, в котором все процессы выполняют одни и те же операции на разных частях разделяемых данных. Таким образом, языки с параллельностью по данным являются императивными. Однако, несмотря на явное взаимодействие в этих языках, синхронизация выражается неявно — после каждого оператора есть неявный барьер.

Понятие *параллельность по данным* появилось в середине 1980-х с появлением первой Connection Machine, CM-1, с архитектурой типа SIMD.[29] CM-1 состояла из обычного управляющего процессора и специального мультипроцессора, состоявшего из тысяч небольших (серверных) процессоров. Управляющий процессор выполнял последовательные команды и рассылал параллельные команды всем серверным процессорам; там эти команды выполнялись синхронно.

Чтобы упростить программирование CM-1, сотрудники корпорации Thinking Machines разработали язык C* (C Star) — вариант языка C, параллельный по данным. И хотя SIMD-машины, подобные CM-1, больше не производятся, стиль программирования, параллельного по данным, остался, поскольку многие приложения легче всего программируются именно в этом стиле. Ниже мы дадим обзор основных черт языков C* и ZPL — нового интересного языка. В следующем разделе описан NESL — еще один новый язык, параллельный по данным и функциональный. Затем представлен учебный пример по HPF, наиболее широко используемому языку с параллельностью по данным. Поскольку в этих языках синхронизация (в основном) неявна, их компиляторы генерируют код, необходимый для синхронизации. Таким образом, не программист, а компилятор использует базовую библиотеку.

C*

Структура языка C* тесно связана с архитектурой CM-1. Он дополняет C свойствами, позволяющими выражать топологию данных и параллельные вычисления. Например, в C* есть

[29] Сам стиль архитектуры SIMD появился гораздо раньше — в 1960-х.

конструкция shape для задания формы параллельных структур данных, например матриц. Параллельное выполнение задается с помощью оператора with, который состоит из последовательности операторов, параллельно обрабатывающих данные. Оператор where поддерживает условное выполнение параллельных операторов. C* также определяет количество операторов редукции, которые комбинируют значения неделимым образом.

Рассмотрим следующий несложный фрагмент программы.

```
shape [256] [256] grid;
real: grid a, b;
инициализация b;
real sum;
with (grid) {
  a = b;  /* параллельно скопировать b в a */
  where (a > 0.0)
    sum = (+= a); /* сложить положительные элементы */
}
```

В первой строке определяется квадратная форма (shape) с именем grid, во второй — две матрицы действительных чисел, имеющие эту форму. В теле оператора with матрица b копируется в a, затем следуют оператор where и оператор редукции для накопления суммы всех положительных элементов матрицы a. Оба операторы присваивания внутри оператора where выполняются параллельно по всем элементам a и b.

C*-программа начинает работу на управляющем процессоре, где выполняются все последовательные операторы. Параллельные операторы компилируются в команды, рассылаемые по одной серверным процессорам. Connection Machine обеспечивала аппаратную поддержку операторов редукции и параллельного перемещения данных между обрабатывающими элементами.

ZPL

Этот новый язык поддерживает и параллельность по данным, и обработку массивов. Таким образом, выражения вроде A+B обозначают сложение целых массивов, которое может выполняться параллельно и заканчивается неявным барьером. ZPL является законченным языком, а не расширением какого-то базового. Однако он компилируется в ANSI C, согласовывается с кодами на Фортране или C и обеспечивает доступ к научным библиотекам.

Новые аспекты ZPL — области и направления. Вместе с выражениями обработки массивов они значительно упрощают программирование обработки матриц. Для иллюстрации объявления и использования областей и направлений используем метод итераций Якоби. Отметим, насколько проще выглядит программа по сравнению с явно параллельной программой в листинге 11.2 и особенно — с программами на C/Pthreads и C/MPI (см. листинги 12.1 и 12.2).

Область (region) — это просто набор индексов. Направление используется для модификации областей или выражений с индексами массивов. Объявляются они следующим образом.

```
region R = [1..n, 1..n];
direction north = [-1, 0]; south = [1, 0];
          east = [0, 1]; west = [0, -1];
```

Область R может использоваться в декларациях так:

```
var A, Temp: [R] float;
    error: float;
```

В этом фрагменте переменные A и Temp объявлены как массивы n×n, состоящие из чисел с плавающей точкой, а error — как одно такое число. Области также можно использовать в выражениях с массивами. Например, в следующем коде внутренние точки массива A инициализируются нулями, а границы вокруг A — единицами.

```
[R]            A := 0.0;
[north of R]   A := 1.0;
[south of R]   A := 1.0;
[east of R]    A := 1.0;
[west of R]    A := 1.0;
```

Префиксы областей указывают на то, что операторы применяются к целым массивам. Таким образом, в первой строке совершается n^2 присваиваний. В каждой из остальных четырех строк выполняется n присваиваний и неявно увеличиваются размеры A, чтобы включить граничные векторы (это означает, что память, выделенная для A, в действительности содержит $(n+2)^2$ значений). Каждый оператор может быть реализован параллельно и завершает свою работу до того, как начинает выполняться следующий оператор. Операторы независимы, поэтому после каждого из них барьер не нужен.

Главный цикл для метода итераций Якоби можно записать на ZPL следующим образом.

```
[R]  repeat
       Temp := ( A@north + A@east + A@west +
                 A@south ) / 4;
       error := max<< abs(A-Temp);
       A := Temp;
     until error < EPSILON;
```

Префикс региона [R] вновь указывает, что операторы в цикле обрабатывают целые массивы. Первый оператор присваивает каждому элементу Temp среднее арифметическое значений четырех его ближайших соседей в A; заметьте, как для выражения индексов этих соседей используются направления. Второй оператор присваивает переменной error максимальное значение разностей пар значений в A и Temp; код max<< является оператором редукции.

12.3.4. Функциональные языки

В императивных языках процессы, взаимодействие и синхронизация выражаются явно. Языки с параллельностью по данным имеют неявную синхронизацию. В функциональных языках неявно все!

В функциональных языках программирования, по крайней мере "чистых", нет концепции видимого состояния программы и, следовательно, нет оператора присваивания, изменяющего состояние. Их называют *языками с одиночным присваиванием*, поскольку переменная может быть связана со значением только один раз. Программа записывается как композиция функций из некоторого набора, а не последовательность операторов.

В языках с одиночным присваиванием нет побочных эффектов, поэтому все аргументы в вызове функции могут вычисляться параллельно. Кроме того, вызываемую функцию можно вычислять, как только будут вычислены все ее аргументы. Таким образом, параллельность явно задавать не нужно; она присутствует автоматически. Процессы взаимодействуют неявно с помощью аргументов и возвращаемых значений. Наконец, синхронизация следует непосредственно из семантики вызова и возврата функции — тело функции не может выполняться, пока не будут вычислены ее аргументы, а результат функции не доступен до возврата из функции.

С другой стороны, семантика одиночного присваивания затрудняет эффективную реализацию функциональных программ. Даже если обновляется только один элемент массива, концептуально создается новая копия всего массива, хотя в нем изменилось всего одно значение. Таким образом, компилятор должен решать, в каких ситуациях безопасно обновлять массив на месте, а не создавать копию. Чтобы определить, перекрываются ли ссылки в массив, нужен анализ зависимости.

Обладая простыми, но мощными свойствами, функциональные языки давно популярны в последовательном программировании. Основанные на функциях, они особенно хороши в рекурсивном программировании. Одним из первых функциональных языков был Lisp; два других языка, Haskell и ML, популярны в настоящее время. В их реализации можно использовать параллельность. В их версиях программисту позволяется решать, где и в какой степени нужна параллельность. Например, Multilisp является версией Lisp, а Concurrent ML — стандартного ML (Standard ML).

Ниже описываются два функциональных языка, NESL и Sisal, разработанные специально для параллельного программирования.

NESL

NESL — функциональный язык с параллельностью по данным. Наиболее важными новыми идеями NESL являются: 1) вложенная параллельность по данным и 2) модель производительности, основанная на языке. NESL допускает применение функции параллельно к каждому элементу набора данных и вложенные параллельные вызовы. Модель производительности основана на концепциях работы и глубины, а не на времени работы. Неформально работа в вычислении — это общее число выполняемых операций, а глубина — самая длинная цепочка последовательных зависимостей.

Последовательные аспекты NESL основаны на ML, функциональном языке, и SETL, лаконичном языке программирования со множествами. Для иллюстрации языка рассмотрим функцию, которая реализует алгоритм быстрой сортировки.

```
function Quicksort(S) =
  if ( #S <= 1) then S
  else
    let a  = S[rand( #S )];
        S1 = {e in S | e < a};
        S2 = {e in S | e == a};
        S3 = {e in S | e > a};
        R  = {Quicksort(v); v in [S1,S3]};
    in R[0] ++ S2 ++ R[1];
```

Последовательности являются базовым типом данных в NESL, поэтому аргументом функции является последовательность `S`. Оператор `if` возвращает `S`, если длина `S` не больше `1`. В противном случае в теле функции выполняется пять присваиваний: 1) переменной `a` присваивается случайный элемент `S`, 2) в последовательность `S1` записываются все значения из `S`, которые меньше `a`, 3) в `S2` — все значения из `S`, равные `a`, 4) в `S3` — все значения из `S` больше `a` и 5) последовательности `R` присваивается результат рекурсивного вызова функции `Quicksort`.

В четырех последних операторах присваивания для параллельного вычисления значений используется оператор "применить ко всем" `{ ... }`. Например, выражение в присваивании `S1` означает "параллельно найти все элементы `e` из `S`, которые меньше `a`". Выражение в присваивании `R` означает "параллельно вызвать `Quicksort(v)` для `v`, равного `S1` и `v`, равного `S3`"; приведенный код является примером вложенной параллельности по данным. Результатами рекурсивных вызовов являются последовательности. В последней строке к первому результату `R[0]` дописываются последовательность `S2` и второй результат `R[1]`.

Sisal

Sisal (Streams and Iteration in a Single Assignment Language — потоки и итерация в языке с одиночным присваиванием) стал первым функциональным языком, разработанным специально для создания научных программ. Его основные понятия — функции, массивы, итерация и потоки. Функции используются, как обычно, для рекурсии и структурирования программы; итерация и массивы — для итеративного параллелизма (они будут рассмотрены ниже). Потоки — это последовательности значений, доступных по порядку; они используются в конвейерном параллелизме и в операциях ввода-вывода. Sisal больше не поддерживается его создателями из лаборатории Lawrence Livermore, однако он внес в программирование важные идеи и используется до сих пор.

Цикл `for` в языке Sisal является первичным механизмом для выражения итеративного параллелизма. Его можно применять, если итерации независимы. Например, в следующем коде для вычисления матрицы `C` как произведения матриц `A` и `B` используются два цикла.

```
C := for i in 1, n cross j in 1, n
       Element := for k in 1, n
         returns value of sum A[i,k] * B[k,j]
       end for
     returns array of Element
   end for;
```

Слово `cross` указывает, что внешний цикл выполняется параллельно для всех пар (cross-product), или комбинаций, образуемых `n` значениями `i` и `n` значениями `j`. Тело внешнего цикла образовано еще одним циклом, который возвращает скалярное произведение `A[i,*]` и `B[*,j]`; ключевое слово `sum` является оператором редукции. Внешний цикл возвращает массив элементов, вычисленных n^2 экземплярами внутреннего цикла. Отметим, что циклы расположены в правых частях двух операторов присваивания; этот синтаксис отражает семантику одиночного присваивания в функциональном языке.

В Sisal поддерживается еще одна циклическая конструкция, `for initial`. Ее используют, когда есть зависимость, создаваемая циклом. Она позволяет написать императивный цикл в функциональном стиле. Например, следующий цикл создает вектор `x[1:n]`, содержащий все частичные суммы вектора `y[1:n]`. (Эта параллельная префиксная проблема рассматривалась в разделе 3.5.)

```
for initial
    i := 1; x := y[1];
while i < n repeat
    i := old i + 1;
    x := old x + y[i];
  returns array of x
end for;
```

В первой части инициализируются две новые переменные и выполняется первая итерация. В части `while` к предыдущему значению `i` каждый раз добавляется `1` и вычисляется новое значение `x`, которое равно сумме `y[i]` и предыдущего значения `x`. Данный цикл возвращает массив, содержащий заново вычисленные значения `x`.

Реализация Sisal основана на модели потоков данных. Выражение можно вычислить, как только будут вычислены его операнды. В цикле умножения матриц, приведенном выше, матрицы `A` и `B` даны и зафиксированы, поэтому можно вычислить все произведения. Суммы произведений определяются по мере вычисления произведений. Наконец, массив элементов можно строить по мере того, как вычисляются все скалярные произведения. Таким образом, каждое значение переходит к тем операторам, которым оно нужно, а операторы порождают выходные значения, только получив все свои входные значения. Модель выполнения, основанная на потоках данных, применяется и в вызовах функций: аргументы независимы, поэтому их можно вычислить параллельно, а тело функции — после определения всех аргументов.

12.3.5. Абстрактные модели

Обычно для определения производительности параллельной программы измеряют время ее выполнения или учитывают каждую операцию. Очевидно, что измерения времени выполнения зависят от сгенерированного машинного кода и соответствующего аппаратного обеспечения. Подсчет операций основан на знании, какие операции можно выполнять параллельно, а какие должны быть выполнены последовательно, и на учете накладных расходов, вносимых синхронизацией. Оба подхода предоставляют подробную информацию, но только относительно одного набора предположений.

Модель параллельных вычислений обеспечивает высокоуровневый подход к характеризации и сравнению времени выполнения различных программ. Это делается с помощью абстракции аппаратного обеспечения и деталей выполнения. Первой важной моделью параллельных вычислений стала машина с параллельным случайным доступом (Parallel Random Access Machine — PRAM). Она обеспечивает абстракцию машины с разделяемой памятью. Модель BSP (Bulk Synchronous Parallel — массовая синхронная параллельная) объединяет абстракции и разделенной, и распределенной памяти. В LogP моделируются машины с распределенной памятью и специальным способом оценивается стоимость сетей и взаимодействия. Упоминавшаяся модель работы и глубины NESL основана на структуре программы и не связана с аппаратным обеспечением, на котором выполняется программа. Ниже дается обзор моделей PRAM, BSP и LogP.

PRAM

PRAM является идеализированной моделью синхронной машины с разделяемой памятью. Все процессоры выполняют команды синхронно. Если они выполняют одну и ту же команду, PRAM является абстрактной SIMD-машиной; однако процессоры могут выполнять и разные команды. Основными командами являются считывание из памяти, запись в память и обычные логические и арифметические операции.

Модель PRAM идеализирована в том смысле, что каждый процессор *в любой момент времени* может иметь доступ к *любой* ячейке памяти. Например, каждый процессор в PRAM может считывать данные из ячейки памяти *или* записывать данные в эту же ячейку. Очевидно, что на реальных параллельных машинах такого не бывает, поскольку модули памяти упорядочивают доступ к одной и той же ячейке памяти. Более того, время доступа к памяти на реальных машинах неоднородно из-за кэшей и возможной иерархической организации модулей памяти.

Базовая модель PRAM поддерживает параллельные считывание и запись (Concurrent Read, Concurrent Write — CRCW). Существуют две более реалистические версии модели PRAM.

- *Исключительное считывание, исключительная запись* (Exclusive Read, Exclusive Write — *EREW*). Каждая ячейка памяти в любой момент времени доступна только одному процессору.
- *Параллельное считывание, исключительная запись* (Concurrent Read, Exclusive Write — *CREW*). Из каждой ячейки памяти в любой момент времени данные могут считываться параллельно, но записываться только одним процессором.

Эти модели более ограничены и, следовательно, более реалистичны, однако и их нельзя реализовать на практике. Несмотря на это, модель PRAM и ее подмодели полезны для анализа и сравнения параллельных алгоритмов.

BSP

В модели массового синхронного параллелизма (BSP) синхронизация отделена от взаимодействия и учтены влияние иерархии памяти и обмена сообщениями. Модель BSP состоит из трех компонентов:

- *процессоры*, которые имеют локальную память и работают с одинаковой скоростью;
- *коммуникационная сеть*, позволяющая процессорам взаимодействовать друг с другом;
- механизм *синхронизации* всех процессоров через регулярные отрезки времени.

Параметрами модели являются число процессоров, их скорость, стоимость взаимодействия и период синхронизации.

Вычисление в BSP состоит из последовательности *сверхшагов*. На каждом отдельном сверхшаге процессор выполняет вычисления, которые обращаются к локальной памяти, и отправляет сообщения другим процессорам. Сообщения являются запросами на получение копии (чтение) или на обновление (запись) удаленных данных. В конце сверхшага процессоры выполняют барьерную синхронизацию и *затем* обрабатывают запросы, полученные в течение данного сверхшага. Далее процессоры начинают выполнять следующий сверхшаг.

Будучи интересной абстрактной моделью, BSP также стала моделью программирования. В частности, в Oxford Parallel Application Center реализованы библиотека взаимодействия и набор протоколирующих инструментов, основанные на модели BSP. Библиотека состоит примерно из 20 функций, в которых поддерживается BSP-стиль обмена сообщениями и удаленный доступ к памяти.

LogP

Модель LogP является более современной. Она учитывает характеристики машин с распределенной памятью и содержит больше деталей, связанных со свойствами выполнения в коммуникационных сетях, чем модель BSP. Процессоры в LogP асинхронные, а не син-

хронные. Компонентами модели являются процессоры, локальная память и соединительная сеть. Свое название модель получила от прописных букв своих параметров:

- L — верхняя граница *задержки* (*Latency*) при передаче сообщения, состоящего из одного слова, от одного процессора к другому;
- o — *накладные расходы* (*overhead*), которые несет процессор, передавая сообщение (в течение этого времени процессор не может выполнять другие операции);
- g — *минимальный временной интервал* (*gap*) между последовательными отправками или получениями сообщений в процессоре;
- P — число пар процессор-память.

Единицей измерения времени является длительность основного цикла процессоров. Предполагается, что длина сообщений невелика, а сеть имеет конечную пропускную способность, т.е. одновременно между процессорами передаются не более $\lceil L/g \rceil$ сообщений.

Модель LogP описывает свойства выполнения в коммуникационной сети, но не ее структуру. Таким образом, она позволяет моделировать взаимодействие в алгоритме. Однако промоделировать время локальных вычислений нельзя. Такой выбор был сделан, поскольку, во-первых, при этом сохраняется простота модели, и, во-вторых, локальное (последовательное) время выполнения алгоритмов в процессорах легко устанавливается и без этой модели.

12.3.6. Учебные примеры: быстродействующий Фортран (HPF)

Быстродействующий Фортран (High-Performance Fortran — HPF) — это самый новый представитель семейства языков, основанных на Фортране. Первая версия HPF была создана многими разработчиками из университетских, промышленных и правительственных лабораторий в 1992 г. Вторая версия была опубликована в начале 1997 г. Несколько компиляторов существуют и сейчас, а HPF-программы могут работать на основных типах быстродействующих машин.

HPF — это язык, параллельный по данным. Он является расширением Фортрана 90, последовательного языка, поддерживающего ряд операций с массивами и их частями. На проект HPF повлиял Фортран D, более ранний диалект Фортрана, параллельный по данным. Основные компоненты HPF: параллельное по данным присваивание массивов, директивы компилятора для управления распределением данных и операторы для записи и синхронизации параллельных циклов. Ниже рассматривается каждый из этих компонентов языка и приводится законченная программа для метода итераций Якоби.

Присваивания массивов

В HPF, подобно Фортрану 90, есть ряд операций, применяемых к целым массивам: присваивание, суммирование, умножение и т.д. Операции над массивами можно также применять к фрагментам массивов сравнимых размеров (так называемым согласованным секциям). Например, если `new` и `grid` — матрицы размера `n×n`, то следующий код реализует главный вычислительный цикл в итерациях Якоби.

```
do iters = 1, MAXITERS
  new(2:n-1,2:n-1) =
       ( grid(1:n-2,2:n-1) + grid(3:n,2:n-1) +
         grid(2:n-1,1:n-2) + grid(2:n-1,3:n) ) / 4
  grid = new
end do
```

Оба присваивания массивов имеют семантику параллельности по данным: сначала вычисляется правая часть, затем все значения присваиваются левой части. В первом присваивании значение в каждой внутренней точке `new` устанавливается равным среднему арифметическому значений ее четырех соседей в `grid`. Во втором присваивании массив `new` копируется обратно в `grid`. В действительности тело этого цикла можно было бы запрограммировать так.

```
grid(2:n-1,2:n-1) =
   (grid(1:n-2,2:n-1) + grid(3:n,2:n-1) +
    grid(2:n-1,1:n-2) + grid(2:n-1,3:n)) / 4
```

Один и тот же массив может появляться в обеих частях — это обусловлено параллельностью по данным семантики присваивания массивов. Однако в коде, сгенерированном компилятором для этого оператора, все равно придется использовать временную матрицу, например new.

Перед присваиванием массивов может стоять выражение WHERE, задающее условные операции с массивами, например приращение только положительных элементов. В языке HPF есть также операторы редукции, которые неделимым образом применяют какую-либо операцию ко всем элементам массива и возвращают скалярное значение. Наконец, HPF обеспечивает ряд так называемых *встроенных* (intrinsic) функций, которые работают с целыми массивами. Например, функция TRANSPOSE(a) вычисляет транспозицию массива a. Другая функция, CSHIFT, обычно применяется для выполнения циклического смещения данных в массиве; в последнем примере правая часть в присваивании grid представляет собой набор смещенных версий grid.

Отображение данных

В языке HPF директивы отображения данных позволяют программисту управлять расположением данных, в частности, их локализацией, особенно на машинах с распределенной памятью. Директивы являются рекомендациями компилятору HPF, т.е. не императивами, которым необходимо следовать, а советами, которые, по мнению программиста, улучшают производительность. В действительности удаление из программы всех директив отображения данных никак не повлияет на результат вычислений; просто программа, возможно, не будет работать столь эффективно.

Основные директивы — PROCESSORS, ALIGN и DISTRIBUTE. Директива PROCESSORS определяет форму и размер виртуальной машины процессоров Директива выравнивания ALIGN определяет взаимно однозначное соответствие между элементами двух массивов, указывая на то, что они должны быть выровнены и одинаково распределены. Директива DISTRIBUTE определяет, каким способом массив (и все выровненные с ним) должен отображаться в памяти виртуальной машины, определенной ранее директивой PROCESSORS; эти два способа обозначаются с помощью BLOCK (блоки) и CYCLIC (полосы).

В качестве примера предположим, что position и force — векторы, скажем, в задаче имитации *n* тел, и рассмотрим следующий код.

```
!HPF$ PROCESSORS pr(8)
!HPF$ ALIGN position (:) WITH force (:)
!HPF$ DISTRIBUTE position(CYCLIC) ONTO pr
```

Первая директива определяет абстрактную машину с восемью процессорами, вторая — задает выравнивание position относительно force. В третьей директиве указано, что вектор position должен отображаться на процессоры циклически (по полосам); соответственно, вектор force будет точно так же поделен на полосы между процессорами.

HPF поддерживает дополнительные директивы отображения данных. TEMPLATE — абстрактная область индексов, которую можно использовать в качестве целевой для директив выравнивания и источника для директивы распределения. Директива DYNAMIC указывает, что выравнивание или распределение массива может изменяться во время работы программы с помощью директивы REALIGN или REDISTRIBUTE.

Параллельные циклы

Присваивания массивов в HPF имеют параллельную по данным семантику и, следовательно, могут выполняться параллельно. HPF также обеспечивает два механизма задания параллельных циклов.

Оператор FORALL указывает, что тело цикла должно выполняться параллельно. Например, в следующем цикле параллельно вычисляются все новые значения в grid.

```
FORALL (i=2:n-1, j=2:n-1)
  new(i,j) = (grid(i-1,j) + grid(i+1,j) +
              grid(i,j-1) + grid(i,j+1)) / 4
```

Результат здесь такой же, как и при присваивании массивов. Однако тело цикла в операторе FORALL может состоять более, чем из одного оператора. Индексы цикла могут также иметь маску для задания предиката, которому должны удовлетворять индексные значения; это обеспечивает возможности, подобные тем, которые предоставляет оператор WHERE, но при этом отпадает необходимость окружать тело цикла операторами if.

Вторым механизмом написания параллельных циклов является директива INDEPENDENT. Программист, помещая ее перед циклом do, утверждает, что тела циклов независимы и, следовательно, могут выполняться параллельно. Например, в коде

```
!HPF$ INDEPENDENT
      do i = 1,n
        A(Index(i)) = B(i)
      end
```

программист утверждает, что все элементы Index(i) различны (не имеют псевдонимов) и A и B не перекрываются в памяти. Если B — функция, а не массив, программист может также использовать директиву PURE, чтобы объявить об отсутствии побочных эффектов в B.

Пример: метод итераций Якоби

В листинге 12.5 представлена законченная HPF-подпрограмма для метода итераций Якоби. В ней используется несколько механизмов, описанных выше. Первые три директивы определяют, что матрицы grid и new должны быть выровнены по отношению друг к другу и располагаться на PR процессорах блоками. Значение PR должно быть статической константой. В теле вычислительного цикла параллельно обновляются все точки матрицы, затем new также параллельно копируется в grid. Когда главный цикл завершается, в последнем присваивании вычисляется максимальная разница между соответствующими друг другу значениями в grid и new. Неявные барьеры установлены после оператора FORALL, оператора копирования массива и оператора редукции массива.

Листинг 12.5. Метод итераций Якоби на быстродействующем Фортране

```
      subroutine Jacobi(n)
      integer n           ! размер с границами
      integer i, j, iters
      real grid(n,n), new(n,n), maxdiff

!HPF$ PROCESSORS pr(PR)    !использовать PR процессоров
!HPF$ ALIGN grid(i,j) WITH new(i,j)
!HPF$ DISTRIBUTE grid(BLOCK) ONTO pr

      инициализация grid и new с границами
      do iters = 1, MAXITERS
        FORALL (i=2:n-1,j=2:n-1)
          new(i,j) = (grid(i-1,j) + grid(i+1,j) +
                      grid(i,j-1) + grid(i,j+1)) / 4
        grid=new     ! параллельное копирование массивов
      end do
      maxdiff = MAXVAL(ABS(grid-new))    ! редукция
      end
```

Сравните эту программу с явно параллельной программой в листинге 11.2 и программой, использующей библиотеку OpenMP (см. листинг 12.4). Данный код намного короче благодаря семантике параллельности по данным языка HPF. С другой стороны, явно параллельный код, подобный кодам в листингах 11.2 и 12.4, должен генерировать компилятор HPF. Это не слишком трудно для данного приложения и машин с разделенной памятью. Но для машины с распределенной памятью создать хороший код гораздо сложнее. Например, программа с явным обменом сообщениями для метода итераций Якоби (см. листинг 11.4) полностью отличается от приведенной выше программы. Компилятор HPF должен распределить данные между процессорами и сгенерировать код с обменом сообщениями. Директивы `ALIGN` и `DISTRIBUTE` дают ему указания, как это делать.

12.4. Инструментальные средства параллельного программирования

В предыдущих разделах данной главы рассматривалась роль библиотек, компиляторов и языков в создании параллельных программ для научных приложений. В процессе разработки, оценки и использования параллельных программ широко применяются также многочисленные вспомогательные инструментальные программные средства. К ним относятся: комплекты эталонных программ проверки производительности, библиотеки для классов приложений (например адаптивные сети, линейная алгебра, разреженные матрицы или неоднородные вычисления) и базовые инструментальные средства (отладчики, параллельные генераторы случайных чисел, библиотеки параллельного ввода-вывода и другие).

В этом разделе описаны два дополнительных вида программного инструментария: для измерения и визуализации производительности приложений и для создания географически распределенных метавычислений. В последнем разделе представлен учебный пример по метакомпьютерной инфраструктуре Globus — одному из самых новых и амбициозных наборов инструментальных средств. В конце главы в исторической справке приведены ссылки на более подробную информацию по всем видам инструментов.

12.4.1. Измерение и визуализация производительности

Цель параллельных вычислений — решить задачу быстрее. Общее время, затраченное на вычисления, подсчитать легко. Намного труднее определить, *где именно* тратится время на вычисления, и, следовательно, определить узкие места. Решать проблемы такого рода помогает инструментарий для визуализации и измерения производительности.

Одним из первых инструментальных средств измерения производительности в параллельных вычислениях, особенно на машинах с распределенной памятью, был Pablo. Проект Pablo возник с появлением первых гиперкубических машин. Его продолжают развивать для расширения возможностей и поддержки современных архитектур. Pablo является инфраструктурой для анализа производительности, позволяющей программисту исследовать приложения на различных уровнях аппаратной и программной реализации. Система проводит редукцию данных в реальном времени и представляет пользователю результаты несколькими способами. Для представления таблиц, схем, диаграмм и гистограмм используется статическая графика. Динамическая графика позволяет наблюдать за развитием во времени, например, за фазами вычислений и взаимодействия. Динамическая графика основана на трассах событий с отметками времени, которые можно отображать в реальном времени или сохранять для дальнейшего просмотра.

Paradyn — более новый инструментарий для измерения производительности параллельных программ. Новизна Paradyn заключается в том, что изучение характеристик приложений является динамическим; оно автоматически включается при запуске программы и уточняется по ходу ее

выполнения. Чтобы использовать Paradyn, разработчику приложения достаточно скомпоновать свою программу с библиотекой Paradyn. При запуске программы система Paradyn начинает поиск таких узких мест на высоком уровне, как чрезмерная блокировка при синхронизации, задержки в работе с памятью или при вводе-выводе. Обнаружив проблемы, Paradyn автоматически вставляет дополнительные средства, чтобы найти причины проблемы (вставка производится непосредственно в машинную программу). Paradyn представляет результаты пользователю в виде "консультации по производительности", в которой пытается ответить на три следующих вопроса. Почему программа выполняется медленно? Где именно в программе есть проблемы с производительностью? При каких условиях проблема возникает? Пользователь может сам искать ответы на эти вопросы или позволить Paradyn провести полное автоматическое исследование.

И в Pablo, и в Paradyn используется графика, позволяющая разработчику делать видимыми аспекты производительности при выполнении программы. Графические пакеты используются во многих приложениях, что позволяет программистам "увидеть" результаты по ходу вычислений. Например, при имитации движения *n* тел можно отображать на экране перемещение тел или при моделировании потока жидкости использовать линии и цвета, чтобы видеть структуру и скорость потоков. Еще один класс инструментов визуализации идет дальше и позволяет программисту *управлять* приложением, изменяя переменные программы по мере выполнения вычислений, и влиять на его дальнейшее поведение. Autopilot — пример недавно разработанного инструментария для управления вычислениями. Данные о производительности в реальном времени, которые дает Autopilot, используются в связанной с ним системе Virtue, реализующей среду погружения (виртуальную реальность). Autopilot и Virtue реализованы на основе частей набора инструментов Pablo. Они также используют систему Globus (описанную далее) для широкомасштабного взаимодействия.

12.4.2. Метакомпьютеры и метавычисления

Большинство параллельных вычислений выполняются на отдельных машинах с мультипроцессором или в группе машин, объединенных в локальную сеть. В идеале процессоры не выполняют другие приложения, и сеть изолирована от другой нагрузки. В локальных вычислительных сетях многие машины подолгу не заняты, например ночью. Периоды простоя можно продуктивно использовать, запуская "долгоиграющие" параллельные приложения, в частности, основанные на парадигме "управляющий-рабочие" (раздел 9.1). Такое использование поддерживается программной системой Condor, которая получила свое название от крупного хищника и ведет "охоту" на свободные машины в сети, захватывая их.

Метакомпьютер — это более общий и интегрированный набор вычислительных ресурсов. Метакомпьютер представляет собой группу компьютеров, объединенных с помощью высокоскоростных сетей и программной инфраструктуры, создающих иллюзию единого вычислительного ресурса. Эта концепция возникла в контексте высокопроизводительных вычислений, поэтому термин "метакомпьютер" считается синонимом *сетевого виртуального суперкомпьютера*. На основе метакомпьютеров возможно создание многочисленных региональных, национальных и интернациональных *вычислительных сетей*, которые будут обеспечивать повсеместную и надежную вычислительную мощность подобно тому, как электросети повсюду и бесперебойно обеспечивают электроэнергию.

Метакомпьютер (или вычислительная сеть) образуется некоторым уровнем программного обеспечения, которое объединяет компьютеры и коммуникационные сети, создавая иллюзию одного виртуального компьютера. Еще один уровень программного обеспечения на вершине этой инфраструктуры обеспечивает *метавычислительную среду*, позволяющую приложениям использовать возможности метакомпьютера. Сущность и роли этих уровней программного обеспечения подобны обычным операционным системам, которые реализуют виртуальную машину на вершине аппаратных ресурсов и поддерживают набор инструментов, используемых прикладными программистами.

Метавычисления обусловлены желанием некоторых пользователей иметь доступ к ресурсам, недоступным в среде одномашинных вычислений. Это свойство присуще некоторым типам приложений:[30]

- распределенные сверхвычисления, которые связаны с решением задач больших объемов и не помещаются на одном суперкомпьютере или могут выиграть от выполнения их разных частей на разных компьютерных архитектурах;
- настольные сверхвычисления, позволяющие пользователю, сидящему у рабочей станции, визуализировать и даже вмешиваться в вычисления, выполняемые на удаленном суперкомпьютере, или получать доступ к удаленным данным, возможно, используя их в качестве входных для прикладной программы;
- интеллектуальные программы, соединяющие пользователей с удаленными приборами (телескопами или электронными микроскопами), которые, в свою очередь, связаны с программой, запущенной на удаленном суперкомпьютере;
- совместные рабочие среды, соединяющие многочисленных пользователей из разных мест с помощью виртуальных пространств и эмуляций, запущенных на суперкомпьютерах.

Конечно, построение таких метавычислительных сред — далеко не тривиальная задача. Необходимо решать много сложных проблем. Перечислим некоторые из них: автономность разных сайтов, беспокойство пользователей по поводу защиты своих машин, многообразие архитектуры компьютеров и их постоянное обновление, отсутствие единого устойчивого пространства имен, ошибки в компонентах, несовместимость языков и операционных систем и т.д.

Одной из первых программных систем, поддерживающих метавычисления, была система Legion. Данный проект появился в 1993 г. и продолжает развиваться. Legion использует объектно-ориентированный подход и реализацию, направленную на решение перечисленных выше проблем. В частности, в Legion определен ряд классов, охватывающих компоненты и возможности системы. Каждый компонент, включая машины, файловые системы, библиотеки и компоненты прикладных программ, инкапсулируется в объект, который является экземпляром одного из классов Legion. Legion предназначен для поддержки всемирного виртуального компьютера и всех перечисленных выше типов приложений.

Более умеренным вариантом метавычислительной системы является Schooner. Она поддерживает настольные суперкомпьютерные приложения. Ключевым аспектом Schooner является язык определения интерфейса, не зависящий от языка программирования и машины; он используется для генерации интерфейсного кода, связывающего программные и аппаратные компоненты в приложении. Другой ключевой аспект Schooner — система времени выполнения, которая поддерживает и статическую, и динамическую конфигурации компонентов в приложении. Например, если удаленная машина оказывается перегруженной во время работы приложения или еще одна машина в сети становится доступной, пользователь может динамически перестроить приложение, чтобы адаптировать его к произошедшим изменениям.

12.4.3. Учебные примеры: инструментальный набор Globus

Globus — это новый, чрезвычайно амбициозный проект, позволяющий конструировать обширное множество инструментальных средств для построения метавычислительных приложений. Руководителями данного проекта являются Ян Фостер (Ian Foster) из Argonne National Labs и Карл Кессельман (Carl Kesselman) из USC's Information Sciences Institute. Их совместные разработки буквально охватывают весь земной шар.

Цель проекта Globus — обеспечить базовый набор инструментов для разработки переносимых высокопроизводительных сервисов, поддерживающих метавычислительные приложения.

[30] Список представлен руководителями проекта Globus Фостером и Кессельманом в [Foster and Kesselman, 1997]. Проект Globus описан в следующем разделе.

Таким образом, Globus основывается на возможностях таких более ранних систем, как PVM, MPI, Condor и Legion, значительно их расширяя. Проект также связан с разработкой способов применения высокоуровневых сервисов к изучению низкоуровневых механизмов и управлению ими.

Компоненты инструментального набора Globus изображены на рис. 12.4. Модули набора выполняются на верхнем уровне метакомпьютерной инфраструктуры и используются для реализации сервисов высокого уровня. Метакомпьютерная инфраструктура, или испытательная модель, реализована программами, соединяющими компьютеры. Группой Globus были построены два экземпляра такой инфраструктуры. Первый, сетевой эксперимент I-WAY, был создан в 1996 г. Он объединил 17 узлов в Северной Америке, его использовали 60 групп для разработки приложений каждого из четырех классов, описанных в предыдущем разделе. Вторая метакомпьютерная инфраструктура GUSTO (Globus Ubiquitous Supercomputing Testbed) была построена в 1997 г. как прототип вычислительной сети, состоящей из 15 узлов, и впоследствии премирована за развитие быстродействующих распределенных вычислений.

Сервисы высокого уровня
Adaptive Wide Area Resource Environment (AWARE) — адаптивная распределенная среда ресурсов
Интерфейс MPI, языковые интерфейсы,
CAVE-среды и другие
Другие сервисы
Legion, Corba и другие

Модули инструментального набора Globus
взаимодействие
размещение и распределение ресурсов
сервисы ресурсов информации
аутентификация
создание процессов
доступ к данным

Метакомпьютерная инфраструктура
I-WAY, GUSTO и другие

Рис. 12.4. Компоненты и структура инструментального набора Globus

Инструментальный набор Globus состоит из нескольких модулей (см. рис. 12.4).

- *Модуль взаимодействий* обеспечивает эффективную реализацию многих механизмов взаимодействия, описанных в части 2, в том числе обмен сообщениями, их многоабонентскую доставку, удаленный вызов процедур и распределенную разделяемую память. В основе его лежит библиотека взаимодействий Nexus.
- *Модуль размещения и распределения ресурсов* обеспечивает механизмы, позволяющие приложениям задавать запросы на ресурсы, распределять ресурсы, удовлетворяющие этим требованиям, и получать к ним доступ.
- *Модуль информационных ресурсов* поставляет справочный сервис, позволяющий приложениям получать текущую информацию о состоянии и структуре основного метакомпьютера.
- *Модуль аутентификации* обеспечивает механизмы, используемые для подтверждения подлинности пользователей и ресурсов. Эти механизмы, в свою очередь, используются как строительные блоки в сервисах, например, санкционирования доступа.
- *Модуль создания процессов* инициирует новые вычисления, объединяет их с уже идущими вычислениями и управляет их завершением.

- *Модуль доступа к данным* обеспечивает скоростной удаленный доступ к постоянной памяти, базам данных и параллельным файловым системам. Этот модуль использует механизмы библиотеки взаимодействий Nexus.

Модули набора Globus помогают реализовывать сервисы приложений высокого уровня. Одним из таких сервисов является так называемая адаптивная распределенная среда ресурсов — Adaptive Wide Area Resource Environment (AWARE). Она содержит интегрированный набор сервисов, в том числе "допустимые метавычислениями" интерфейсы для реализации библиотеки MPI, различные языки программирования и инструменты для создания виртуальных сред (constructing virtual environments — CAVE). Среди сервисов высокого уровня есть и разработанные другими, включая упомянутую выше систему метавычислений Legion и реализации CORBA (Common Object Request Broker Architecture).

Инструментальный набор Globus — это новый развивающийся проект, так что его описание изменяется по мере разработки приложений и поддержки сервисов. Заинтересованный читатель может получить информацию о текущем состоянии и последних достижениях проекта Globus, посетив его Web-сайт по адресу `www.globus.org`.

Историческая справка

Библиотеки параллельного программирования долгое время разрабатывались производителями параллельных машин. Однако, за редкими исключениями, все эти библиотеки были несовместимыми. В 1990-х годах стали появляться стандартные библиотеки, облегчавшие разработку переносимых кодов. Библиотека Pthreads рассматривалась в главе 4; там же в справке есть ссылки на другие источники информации о ней. Библиотека MPI была представлена в главе 7; реализации MPI и ее "близкой родственницы" библиотеки PVM описаны в исторической справке главы 7. Информацию по OpenMP, включая онлайновые обучающие программы, можно найти в Internet по адресу: `www.openmp.org`.

Основополагающая работа по анализу зависимости была проведена под руководством Д. Кука (David Kuck) сотрудниками Иллинойского университета У. Банерджи (Utpal Banerjee) и М. Вольфом (Michael Wolfe) и представлена в книге [Wolfe, 1989]. В работе [Bacon, Graham, Sharp, 1994] дан прекрасный обзор анализа зависимости и преобразований компилятора для быстродействующих вычислений; работа содержит множество ссылок. В статье [McKinley, Carr, Tseng, 1996] описаны эксперименты, использующие различные преобразования для улучшения распределения данных, и даются рекомендации по очередности выполнения оптимизаций.

Некоторые из языков, перечисленных на рис. 12.3, были объектами учебных примеров в предыдущих главах: Ada в главе 8, Java в главах 5, 7 и 8, SR в главе 8, CSP/Occam и Linda в главе 7. Другие источники информации по этим языкам указаны в исторических справках соответствующих глав.

Из объектно-ориентированных языков на рис. 12.3 указаны только Java и Orca. Для параллельного программирования разработано также много других объектно-ориентированных языков. Например, в книге [Wilson and Lu, 1996] представлено более дюжины языков и систем, основанных на C++. Отличное описание одного из этих языков — Compositional C++, а также Фортрана M, HPF и библиотеки MPI содержится в [Foster, 1995]. Обзор языков программирования для распределенных вычислений, включая объектно-ориентированные и функциональные языки, можно найти в статье [Bal, Steiner, Tanenbaum, 1989].

Многие исследователи используют Java в быстродействующих вычислениях из-за его популярности и широкого распространения. В специальных выпусках *Concurrency — Practice and Experience* (июнь и ноябрь 1997 г.) есть несколько статей, описывающих конкретные проекты.

Cilk, разработанный Ч. Лейзерсоном (Charles Leiserson) и его студентами в MIT, в настоящее время имеет версию 5. В работе [Frigo, Leiserson and Randall, 1998] дается обзор языка Cilk-5 и описывается его реализация. Эту и многие другие работы можно найти в Internet по

адресу supertech.lcs.mit.edu/cilk/. Там есть также введение в язык, полное справочное руководство и дистрибутив программного обеспечения.

Fortran M был разработан Я. Фостером (Ian Foster) из Argonne National Labs и М. Чанди (Mani Chandy) из California Institute of Technology. В статье [Foster and Chandy, 1995] описан язык и приведены многочисленные примеры его использования (см. также книгу [Foster, 1995]). Более обширная информация по языку HPF, дистрибутив программного обеспечения и последние работы по связыванию его с MPI есть на сайте www-fp.mcs.anl.gov/fortran-m/.

Язык Orca был разработан в 1989 году Х. Балом (Henri Bal) в Амстердамском Vrije Universiteit в его диссертации под руководством А. Таненбаума. В статье [Bal, Kaashoek and Tanenbaum, 1992] описан исходный язык, основанный на модели выполнения, параллельного по задачам, а в [Ben Hassen, Bal and Jacobs, 1998] — новая версия языка Orca, которая поддерживает параллельность и по задачам, и по данным. Более подробную информацию по этому языку можно найти в Internet на сайте www.cs.vu.nl/orca/.

Язык C* был разработан в середине 1980-х годов в Thinking Machines Corporation. В книгах [Kumar et al., 1994] и [Quinn, 1994] дан обзор языка и приведены примеры программ.

Язык ZPL является детищем Л. Снайдера (Larry Snyder) и его студентов из Вашингтонского университета. Первая версия ZPL появилась в начале 1990-х, хотя корни проекта ZPL уходят в 1980-е. В статье [Chamberlain et al., 1998] дано описание языка и приведены аргументы в пользу того, что он упрощает программирование вычислений, параллельных по данным, является переносимым и достаточно производительным. Web-сайт проекта ZPL размещен по адресу www.cs.washington.edu/research/zpl/. На сайте также есть дистрибутив программного обеспечения, обзор языка, несколько статей, тезисов и диссертаций.

Язык NESL был разработан Г. Блеллохом (Guy Blelloch) и его коллегами в Carnegie Mellon University. Язык, основные принципы его работы и базовая модель описаны в статье [Blelloch, 1996], а на Web-сайте с адресом www.cs.cmu.edu/~scandal/nesl.html находятся дистрибутив программного обеспечения, библиотека параллельных алгоритмов, интерактивные обучающие программы и многое другое.

Как уже отмечалось, первым функциональным языком, созданным специально для поддержки научных вычислений, стал Sisal. Он был разработан группой сотрудников Lawrence Livermore National Laboratory и основывался на Val, более раннем языке с потоками данных. В 1985 г. появилась первая версия Sisal. В статье [Feo, Cann and Oldehoeft, 1990] описан язык, его реализация и производительность. В середине 1990-х годов появилась вторая версия языка Sisal, но вскоре проект был закрыт. Существует Web-страница по адресу www.llnl.gov/sisal/, однако она уже не поддерживается.

В разделе 12.3 упоминались три дополнительных функциональных языка программирования: Lisp, Haskell и ML. В обзоре [Hudak, 1989] описаны эти и другие функциональные языки, а также основные принципы параллельного функционального программирования. Язык Haskell содержит средства управления параллельностью, а Multilisp и Concurrent ML представляют собой версии соответственно Lisp и ML для параллельного программирования. Информацию по этим языкам можно найти с помощью поисковых средств Internet, например www.yahoo.com.

Модель PRAM была представлена в работе [Fortune and Wyllie, 1978]. Модель PRAM описывается и иллюстрируется во многих книгах по параллельным алгоритмам, например [Kumar et al., 1994] и [Quinn, 1994]. Модель BSP была предложена в [Valiant, 1990]; текущую информацию о ней можно найти на узле www.bsp-worldwide.org/. Модель LogP описана в статье [Culler, 1996], а в обзоре [Skillicorn and Talia, 1998] рассмотрено очень много моделей и языков для параллельных вычислений. В работе [Juurlink and Wijshoff, 1998] представлено количественное сравнение трех моделей: BSP, E-BSP (расширение BSP, имеющее дело с несбалансированными схемами взаимодействия) и BPRAM (версия модели PRAM с обменом сообщениями).

Текущую информацию по быстродействующему Фортрану (HPF) можно найти на домашней страничке HPF по адресу

dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm.

Страница поддерживает связи с онлайновыми обучающими программами, спецификацией языка, другими публикациями и разработками. В статье [Brinch Hansen, 1998] дается критическая оценка HPF, а в [Offner, 1998] — отпор этой критике. Разработчики компилятора dHPF в Rice University являются одной из основных групп, развивающих новые технологии анализа программ и генерации кодов для языков, параллельных по данным, вообще и HPF в частности; их домашняя страница имеет адрес `www.cs.rice.edu/~dsystem/dhpf/`.

В разделе 12.4 был описан ряд инструментальных средств для параллельного программирования. Информацию о проекте Pablo, а также об инструментах визуализации Autopilot и Virtue можно найти по адресу `www-pablo.cs.uiuc.edu/`. В статье [Miller et al., 1995] дан обзор набора инструментов для измерения производительности Paradyn; домашняя страница этого проекта имеет адрес `http://www.cs.wisc.edu/~paradyn/`. В книге [Foster, 1995] описаны дополнительные инструменты для анализа производительности, визуализации и генерации случайных чисел.

В статье [Grimshaw et al., 1998] представлен обзор по метакомпьютерам и метавычислениям, а в [Grimshaw and Wulf, 1997] — система Legion, один из наиболее важных примеров системы метавычислений; домашняя страница проекта размещена по адресу `legion.virginia.edu`. В работе [Homer and Schlichting, 1994] описана система Schooner для распределенных научных вычислений; более подробную информацию можно найти по адресу `www.cs.arizona.edu/schooner/`.

В ноябрьском выпуске *Communications of the ACM* за 1997 г. есть несколько статей, в которых описываются вычислительные сети, способы их реализации и использования. Основным источником информации по этим вопросам является книга [Foster and Kesselman, 1999]. В главах 1 и 2 этой книги подробно описаны вычислительные сети, а в остальных 20 главах — различные приложения, инструменты программирования, сетевые сервисы и инфраструктуры аппаратного и программного обеспечения. Главы написаны экспертами, работающими над реализацией вычислительных сетей. Набор инструментов Globus, обсуждаемый в конце раздела 12.4, представлен в главе 11 данной книги, а впервые упоминается в статье [Foster and Kesselman, 1997]. Web-страница проекта Globus размещена по адресу `www.globus.org`.

В Интернет есть несколько глобальных информационных архивов по параллельным вычислениям. Например, архив Netlib по математическому программному обеспечению, статьи и базы данных находятся по адресу `www.netlib.org/`. Parallel Tools Consortium — это группа исследователей, разработчиков и пользователей, создающих инструменты для поддержки параллельных вычислений; адрес домашней странички консорциума — `www.ptools.org`. Хорошо организованный набор ссылок можно найти по адресу `www.cs.rit.edu/~ncs/parallel.html` (страничка Nan Schaller по параллельным вычислениям). На сайте IEEE Computer Society's ParaScope по адресу `computer.org/parascope/` находится постоянно обновляемая информация о журналах, конференциях, производителях, сайтах и проектах, относящихся к параллельным вычислениям. Информация на сайте ParaScope обширна, поэтому лучше использовать поиск по имени.

Литература

Bacon, D. F., S. L. Graham, and O. J. Sharp. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys* 26, 4 (December): 345–420.

Bal, H. E., J. G. Steiner, and A. S. Tanenbaum. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys* 21, 3 (September): 261–322.

Bal, H. E., M. E. Kaashoek, and A. S. Tanenbaum. 1992. Orca: A language for parallel programming on distributed systems. *IEEE Trans. Softw. Eng.* 18, 3 (March): 190–205.

Ben Hassen, S., H. E. Bal, and C. J. H. Jacobs. 1998. A task- and data-parallel programming language based on shared objects. ACM *Trans. on Programming Lang. and Systems* 20, 6 (November): 1131–1170.

Blelloch, G. E. 1996. Programming parallel algorithms. *Comm. of the ACM* 39, 3 (March): 85–97.

Brinch Hansen, P. 1998. An evaluation of High Performance Fortran. *ACM SIGPLAN Notices* 33, 3 (March): 57–64.

Chamberlain, B. L., S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. 1998. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engr.* 5, 3: 76–86.

Culler, D. E., et al. 1996. LogP: A practical model of parallel computation. *Comm. of the ACM* 39, 11 (November): 75–85.

Feo, J. T., D. C. Cann, and R. R. Oldehoeft. 1990. A report on the Sisal language project. *Journal of Parallel and Dist. Comp.* 10: 349–366.

Fortune, S., and J. Wyllie. 1978. Parallelism in random access machines. *Proc. 10th ACM Symp. on Theory of Computing*, pp. 114–118.

Foster, I. 1995. *Designing and Building Parallel Programs*. Reading, MA: Addison-Wesley.

Foster, I. T., and K. M. Candy. 1995. Fortran M: A language for modular parallel programming. *Journal of Parallel and Dist.Comp.* 26: 24–35.

Foster, I., and C. Kesselman. 1997. Globus: A metacomputing infrastructure toolkit. *Int. Journal of Supercomputing Applications* 11, 2; 115–128.

Foster, I., and C. Kesselman, eds. 1999. *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco: Morgan Kaufmann.

Frigo, M., C. E. Leiserson, and K. H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. *Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June: 212–223.

Grimshaw, A. S., and W. A. Wulf. 1997. The Legion vision of a worldwide virtual computer. *Comm. of the ACM* 40, 1 (January): 39–45.

Grimshaw, A., A. Ferrari, G. Lindahl, and K. Holcomb. 1998. Metasystems. *Comm. of the ACM* 41, 11 (November): 47–55.

Homer, P., and R. Schlichting. 1994. A software platform for constructing scientific applications from heterogeneous resources. *Journal of Parallel and Distributed Computing* 21, 3 (June): 301–315.

Hudak, O. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys* 21, 3 (September): 359–411.

Juurlink, B. H., and H. A. G. Wijshoff. 1998. A quantitative comparison of parallel computational models. *ACM Trans. on Computer Systems* 16, 3 (August): 271–318.

Kumar, V., A. Grama, A. Gupta, and G. Karypis. 1994. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Menlo Park, CA: Benjamin/Cummings.

McKinley, K. S., S. Carr, and C.-W. Tseng. 1996. Improving data locality with loop transformations. *ACM Trans. of Programming Lang. And Systems* 18, 4 (July): 4214–4253.

Miller, B. P., et al. 1995. The Paradyn parallel performance measurement tool. *IEEE Computer* 28, 11 (November): 37–46.

Offner, C. D. 1998. Per Brinch Hansen's concerns about High Performance Fortran. *ACM SIGPLAN Notices* 33, 8 (August): 34–39.

Quinn, M. J. 1994. *Parallel Computing: Theory and Practice*. New York: McGraw-Hill.

Skillicorn, D. B., and D. Talia. 1998. Models and languages for parallel computation. *ACM Computing Surveys* 30, 2 (June): 123–169.

Valiant, L. G. 1990. A bridging model for parallel computation. *Comm. of the ACM* 33, 8 (August): 103–111.

Wilson, G. V., and P. Lu, eds. 1996. *Parallel Programming Using C++*. Cambridge, MA: MIT Press.

Wolfe, M. 1989. *Optimizing Supercompilers for Supercomputers*. Cambridge, MA: MIT Press.

Упражнения

12.1. В разделе 12.1 представлены реализации метода итераций Якоби, в которых используются библиотеки Pthreads, MPI и OpenMP. С помощью библиотек Pthreads, MPI и OpenMP составьте параллельные программы:

а) для задачи *n* тел;

б) для реализации LU-разложения.

Для программ с Pthreads и OpenMP используйте разделяемые переменные, а с MPI — обмен сообщениями. Последовательные части своих программ напишите на C для Pthreads и на C или Фортране для MPI и OpenMP.

12.2. Рассмотрите последовательные программы для следующих задач:

а) метод итераций Якоби (листинг 11.1);

б) задача *n* тел (листинг 11.6);

в) LU-разложение (листинг 11.11);

г) прямой и обратный ход (листинг 11.12).

В каждой программе выделите все случаи: 1) потоковых зависимостей, 2) антизависимостей, 3) зависимостей по выходу. Укажите, какие из зависимостей находятся в циклах, а какие создаются циклами.

12.3. Рассмотрим следующие последовательные программы:

а) метод итераций Якоби (листинг 11.1);

б) задача *n* тел (листинг 11.6);

в) LU-разложение (листинг 11.11);

г) прямой и обратный ход (листинг 11.12).

На рис. 12.1 перечислены некоторые типы преобразований программ в распараллеливающих компиляторах. Для каждого преобразования и каждой из перечисленных выше последовательных программ определите, можно ли применить преобразование к программе, и, если можно, покажите, как это сделать. Каждое преобразование рассматривайте независимо от других.

12.4. Повторите предыдущее упражнение, применяя в каждой программе несколько преобразований. Предположим, что программа пишется для машины с разделяемой памятью, имеющей восемь процессоров, а размер задачи n кратен восьми. Ваша цель — создать программу, которую совсем просто превратить в параллельную программу со сбалансированной вычислительной нагрузкой, удачным распределением данных и небольшими накладными расходами синхронизации. Для каждой программы: 1) подберите последовательность преобразований, 2) покажите, как изменяется программа после каждого преобразования, и 3) объясните, почему вы считаете, что либо полученный код приведет к хорошей параллельной программе, либо из исходной программы получить хорошую параллельную программу нельзя.

12.5. В конце раздела 12.2 было показано, что алгоритм Гаусса–Зейделя обладает так называемой *параллельностью фронта волны*:

а) напишите параллельную программу для метода Гаусса–Зейделя. Для взаимодействия используйте разделяемые переменные, а для параллельности и синхронизации фронта волны — оператор `co`;

б) напишите еще одну параллельную программу для метода Гаусса–Зейделя. Вновь используйте разделяемые переменные, но создайте процессы один раз и для синхронизации фронта волны используйте флаговые переменные;

в) оцените и сравните производительность своих программ из пунктов *а* и *б*. Всегда ли одна из программ лучше или это зависит от ситуации?

12.6. В разделе 12.3 дано краткое описание некоторых языков параллельного программирования:

а) выберите язык, соберите все ссылки на него и напишите полный "учебный пример" по нему. Степень детализации должна быть аналогичной учебным примерам в тексте по языкам Linda, CPS, Ada и SR;

б) выберите язык, соберите по нему информацию. Приобретите и установите для него программное обеспечение (если необходимо). Затем на этом языке напишите программы для метода итераций Якоби, задачи *n* тел, LU-разложения. Возможно, вы захотите написать одну или несколько рекурсивных параллельных программ. Составьте отчет с описанием языка, программ и опыта программирования. Легким или тяжелым был язык при изучении? Насколько легко или трудно им пользоваться? Насколько хорошо или плохо работают ваши программы?

в) повторите пункт *б* с другим языком. К отчету добавьте сравнение языков. Какой из языков вы предпочитаете и почему?

12.7. Приобретите и установите компилятор для HPF (если необходимо). Реализуйте и протестируйте программу для метода итераций Якоби (листинг 12.5), затем — для задачи *n* тел и LU-разложения. Если возможно, запустите свои программы на машинах с разделяемой и с распределенной памятью. Составьте отчет с описанием ваших программ, опыта программирования и их производительности.

12.8. В разделе 12.4 описано несколько инструментальных средств и наборов инструментов для параллельного программирования, а в исторической справке даны ссылки на многие другие. Соберите подробную информацию о каком-либо инструментальном средстве, приобретите и установите его. Затем поэкспериментируйте с ним на новых и на составленных ранее программах. Составьте отчет, описывающий инструментальное средство, а также ваши программы, эксперименты и опыт по его использованию. Насколько легко было пользоваться этим инструментом? Чему вы научились при работе с ним?

Словарь

Активная блокировка (Spin Lock)

Булева переменная, используемая в конъюнкции с активным ожиданием для защиты критической секции. Стремясь войти в критическую секцию, процесс зацикливается до тех пор, пока блокировка не будет освобождена.

Активное (занятое) ожидание (Busy Waiting)

Реализация синхронизации, в которой процесс многократно выполняет цикл, ожидая, что некоторое булево условие `B` станет истинным. Обычно это программируется как `while (!B) skip;`. Если процесс находится в состоянии занятого ожидания, говорят также, что он *зациклен* (*spinning*).

Алгоритм "зонд-эхо" (Probe/Echo Algorithm)

Парадигма взаимодействия процессов в распределенных программах. *Зонд* используется для рассылки информации от одного процесса всем остальным, а *эхо* — для сбора информации.

Алгоритм передачи маркера (Token-Passing Algorithm)

Схема взаимодействия процессов в распределенной программе, использующая маркеры для передачи разрешения или сбора информации о глобальном состоянии.

Алгоритм пульсации (Heartbeat Algorithm)

Парадигма взаимодействия процессов в распределенных программах. Каждый процесс периодически выполняет три фазы: 1) передает сообщения другим процессам, 2) принимает сообщения от других процессов, 3) выполняет вычисления с локальными данными и данными, полученными в сообщениях.

Алгоритм рассылки (Broadcast Algorithm)

Метод распространения информации или принятия решений в распределенной программе. Для принятия решения каждый процесс отправляет запросы и подтверждения всем остальным процессам и обслуживает упорядоченную очередь сообщений, по которой определяется наиболее давний запрос.

Балансировка нагрузки (Load Balancing)

Назначение каждому процессу (и процессору) в параллельной программе приблизительно одинакового количества работы.

Барьер (Barrier)

Точка синхронизации, которую должны достичь все процессы перед тем, как некоторым из них будет разрешено дальнейшее выполнение.

Безусловное неделимое действие (Unconditional Atomic Action)

Неделимое действие, не имеющее условия задержки. Программируется как `<S;>` и может быть реализовано одной машинной командой.

Блокировка (Lock)

Переменная, используемая для защиты критической секции. Блокировка устанавливается, только если некоторый процесс выполняется в критической секции.

Взаимная блокировка (Deadlock)

Состояние, в котором несколько процессов ожидают друг друга в так называемом клинче, или "смертельных объятиях" (deadly embrace).

Взаимное влияние (Interference)

Результат чтения или записи разделяемых переменных в двух процессах в непредсказуемом порядке и, следовательно, с непредсказуемыми результатами; см. также **Невмешательство**

Взаимное исключение (Mutual Exclusion)

Тип синхронизации, обеспечивающей, что операторы в разных процессах не могут выполняться одновременно; см. также **Критическая секция**.

Взаимодействие производителей и потребителей (Producers and Consumers Interaction)

Взаимодействие двух процессов, при котором один из них создает данные, используемые в другом; см. также **Процесс-фильтр** и **Конвейер**.

Взаимодействующие равные (Interacting Peers)

Модель взаимодействия процессов в распределенных программах, в которой процессы равны, т.е. выполняют один и тот же код и взаимодействуют друг с другом для обмена информацией.

Вложенный вызов монитора (Nested Monitor Call)

Вызов одного монитора из другого. Когда процесс выполняет вложенный вызов, то говорят, что вызов *открытый*, если процесс освобождает исключение в первом (вызывающем) мониторе. Вызов называется *закрытым*, если процесс сохраняет исключение в первом мониторе.

Глобальный инвариант (Global Invariant)

Предикат, истинный в каждом видимом состоянии программы, а именно, до и после каждого неделимого действия.

Живая блокировка (Livelock)

Ситуация, в которой процесс зациклен в ожидании условия, которое никогда не станет истинным. Живая блокировка является "активно ожидающим" аналогом взаимной блокировки.

Инвариант цикла (Loop Invariant)

Предикат, который является истинным до и после каждого выполнения операторов цикла.

Исключение конфигураций (Exclusion of Configurations)

Метод доказательства таких свойств безопасности, как, например, взаимное исключение и отсутствие взаимной блокировки. Процесс P_i не может находиться в состоянии, удовлетворяющем условию A_i, одновременно с пребыванием процесса P_j в состоянии, удовлетворяющем условию A_j, если $(A_i \wedge A_j) == false$.

История (History)

Последовательность состояний или действий, полученная при одном выполнении программы; история также называется *трассой*, или *следом* (*trace*).

Итерационный параллелизм (Iterative Parallelism)

Тип параллелизма, в котором каждый процесс выполняет цикл, обрабатывающий часть данных программы. Часто возникает в результате распараллеливания циклов последовательной программы.

Конвейер (Pipeline)

Множество процессов, соединенных в ряд так, что выход одного процесса является входом для следующего. Конвейер бывает: *а*) *открытым*, если его концы не связаны с другими процессами; *б*) *круговым*, если концы замыкаются друг на друга, образуя круг; *в*) *закрытым*, если он круговой и есть внешний управляющий процесс, который поставляет (производит) вход для первого рабочего процесса в конвейере и потребляет выход последнего.

Корректность (Total Correctness)

Свойство программы вычислять желаемый результат и завершаться.

Критическая секция (Critical Section)

Последовательность операторов, которая должна выполняться со взаимным исключением по отношению к критическим секциям других процессов, ссылающихся на те же разделяемые переменные. Протоколы критической секции обычно используются для реализации крупномодульных неделимых действий.

Ложное разделение данных (False Sharing)

Ситуация, в которой два процесса ссылаются на различные переменные, расположенные в одной строке кэша или странице DSM, и хотя бы один из процессов записывает в "свою" переменную.

Метакомпьютер (Metacomputer)

Группа компьютеров, объединенных высокоскоростной сетью и инфраструктурой программного обеспечения, создающей иллюзию одного компьютера. Иногда называется *сетевым виртуальным суперкомпьютером.*

Многопоточная программа (Multithreaded Program)

Программа, содержащая множество процессов или потоков. Является синонимом параллельной (concurrent) программы, хотя понятие "многопоточная программа" часто означает, что в программе потоков больше, чем процессоров, поэтому потоки поочередно выполняются на процессорах.

Мультикомпьютер (Multicomputer)

Мультипроцессор со многими потоками команд и многими потоками данных (MIMD-мультипроцессор) с распределенной памятью, например, гиперкубическая машина.

Мультипроцессор с одним потоком команд и многими потоками данных (SIMD-мультипроцессор) (Single Instruction, Multiple Data Multiprocessor)

Аппаратная архитектура, в которой один поток команд выполняется синхронно (in lockstep) каждым процессором, обрабатывающим свои локальные данные; также называется *синхронным мультипроцессором.*

Мультипроцессор со многими потоками команд и многими потоками данных (MIMD-мультипроцессор) (Multiple Instruction, Miltiple Data Multiprocessor)

Аппаратная архитектура со множеством независимых процессоров, каждый из которых занят выполнением своей программы; также называется *асинхронным мультипроцессором.*

Невмешательство (Noninterference)

Отношение между неделимым действием a в одном процессе и критическим утверждением C в другом. Выполнение a не вмешивается в C, если оставляет C истинным при условии, что C уже истинно.

Неделимое действие (Atomic Action)

Последовательность из одного или нескольких операторов, которая выполняется неделимым образом. *Мелкомодульное неделимое действие* (*fine-grained atomic action*) может быть реализовано непосредственно одной машинной командой. *Крупномодульное неделимое действие* (*coarse-grained atomic action*) реализуется с помощью протоколов критической секции.

Независимые операторы (Independent Statements)

Два оператора в разных процессах, которые не записывают данные в одни и те же переменные, причем каждый из них не считывает переменные, в которые записывает другой оператор. Независимые операторы никогда не влияют друг на друга при параллельном выполнении.

"Одна программа — много данных" (Single Program, Multiple Data — SPMD)

Стиль программирования, при котором кодируется один процесс. Копия процесса выполняется на всех процессорах; каждая копия имеет свои собственные данные. Обычно существует способ, по которому процесс может определить свой идентификатор (иногда называемый его рангом).

Парадигма дублируемых серверов (Replicated Servers Paradigm)

Схема взаимодействия процессов в распределенной программе, имеющей много экземпляров серверного процесса. Каждый сервер управляет частью или копией некоторого разделяемого ресурса; серверы взаимодействуют между собой, чтобы поддерживать согласованное состояние ресурса.

Парадигма "портфель задач" (Bag-of-Task Paradigm)

Метод параллельных вычислений, при котором все задачи помещаются в портфель, разделяемый рабочими процессами. Каждый процесс многократно берет задачу из портфеля, выполняет ее и, возможно, порождает новые задачи, которые помещает в портфель. Вычисления завершаются, когда портфель пуст и рабочие процессы незаняты.

Парадигма "управляющий-рабочие" (Manager/Workers Paradigm)

Распределенная реализация парадигмы "портфель задач". Управляющий процесс реализует портфель задач и собирает результаты; рабочие процессы взаимодействуют с управляющим, чтобы получать от него задачи и возвращать ему результаты.

Параллельная программа (Concurrent Program)

Программа, состоящая из нескольких процессов. Каждый процесс — это последовательная программа; см. также **Распределенная программа** и **Синхронная параллельная программа**.

Передача эстафеты (Passing the Baton)

Техника синхронизации, используемая с семафорами. Когда процесс решает, что следует активизировать еще один процесс, он передает сигнал семафору, у которого ожидает этот процесс. Действие этого сигнала подобно передаче эстафеты второму процессу; она передает разрешение на выполнение с помощью взаимного исключения.

Переключение контекста (Context Switch)

Переключение процессора с выполнения одного процесса на выполнение другого. Оно называется переключением контекста, поскольку состояние каждого процесса называется его контекстом. Переключение контекста выполняется в ядре программой, которая называется диспетчером или планировщиком.

Покрывающее условие (Covering Condition)

Механизм синхронизации, используемый с мониторами. Процесс передает сигнал условной переменной, когда можно продолжить выполнение ожидающих процессов. Условие, связанное с этой переменной, "покрывает" настоящие условия, которых ожидают процессы.

Постусловие (Postcondition)

Условие, истинное после выполнения оператора.

Поток (Thread)

Последовательная программа, которая имеет свой собственный поток управления (контекст) и может выполняться одновременно с другими потоками. Потоки соревнуются за время одного и того же процессора или выполняются параллельно на разных процессорах.

Предусловие (Precondition)

Условие, истинное перед выполнением оператора.

Программа, параллельная по данным (Data Parallel Program)

Программа, в которой все процессы выполняют (обычно одновременно) одни и те же действия над разными частями разделяемых данных.

Программа, параллельная по задачам (Task Parallel Program)

Программа, в которой каждый процесс выполняет отдельную задачу и, следовательно, является отдельной последовательной программой.

Программа типа "клиент-сервер" (Client/Server Program)

Схема взаимодействия процессов в распределенной программе. Процесс-сервер управляет ресурсом и реализует операции над этим ресурсом. Клиент посылает запрос на сервер, активизируя одну из операций сервера.

Процесс-фильтр (Filter Process)

Процесс, который получает (считывает) данные из одного или нескольких входных каналов, вычисляет функцию и отправляет (записывает) результаты в один или несколько выходных каналов. Процесс-фильтр является одновременно производителем и потребителем и может использоваться в конвейере.

Распределенная программа (Distributed Program)

Программа, в которой процессы взаимодействуют с помощью передачи сообщений, удаленного вызова процедур или рандеву. Обычно процессы выполняются на разных процессорах.

Распределенная разделяемая память (Distributed Shared Memory — DSM)

Программная реализация разделяемого адресного пространства на мультипроцессоре с распределенной памятью или на сети процессоров.

Рекурсивный параллелизм (Recursive Parallelism)

Тип параллелизма, возникающего в результате совершения параллельных рекурсивных вызовов. Часто это происходит при распараллеливании последовательных программ, использующих парадигму "разделяй и властвуй" для решения все уменьшающихся задач.

Свойство безопасности (Safety Property)

Свойство программы, утверждающее, что ничего опасного не произойдет, т.е. что программа никогда не достигнет опасного состояния. Частичная корректность, взаимное исключение и отсутствие взаимных блокировок — типичные примеры свойства безопасности.

Свойство живучести (Liveness Property)

Свойство программы, утверждающее, что нечто надлежащее в конце концов произойдет, т.е. программа в конце концов достигнет "хорошего" состояния. Примерами данного свойства являются завершение и возможный вход в критическую секцию.

Свойство "не более одного" (At-Most-Once Property)

Свойство оператора присваивания `x = e`, в котором либо *а*) `x` не читается другим процессом, а `e` содержит не более одной ссылки на переменную, изменяемую другим процессом, либо *б*) `x` не изменяется другими процессами, а `e` не содержит ссылок на переменные, изменяемые другими процессами. Такой оператор присваивания выполняется неделимым образом.

Симметричный мультипроцессор (Symmetric Multiprocessor — SMP)

Архитектура мультипроцессоров с разделяемой памятью, в которой все процессоры идентичны, и каждый получает доступ к любому слову памяти за одно и то же время.

Синхронизация (Synchronization)

Взаимодействие между процессами, управляющее порядком их выполнения; см. также **Взаимное исключение** и **Условная синхронизация**.

Синхронная параллельная программа (Parallel Program)

Программа, в которой каждый процесс выполняется на своем собственном процессоре, т.е. процессы выполняются параллельно. (Вместе с тем, этот термин иногда употребляется по отношению к любой "многопроцессной" программе.)

Состояние гонок (Race Condition)

Ситуация в параллельной программе с разделяемыми переменными, когда один процесс записывает в переменную, которая должна читаться в другом процессе, но продолжает выполнение ("вырывается вперед") и вновь изменяет эту переменную до того, как второй процесс увидит результат первого изменения. Обычно это приводит к некорректно синхронизированной программе.

Состояние программы (State of a Program)

Значения всех переменных программы в некоторый момент времени.

Справедливость (Fairness)

Свойство диспетчера или алгоритма, гарантирующее, что каждый отложенный процесс получит шанс на продолжение: см. также **Стратегия планирования**.

Стратегия планирования (Scheduling Policy)

Алгоритм, по которому определяется очередность действий, т.е. порядок выполнения процессов. Стратегия планирования бывает: *а) безусловно справедливой*, если безусловные неделимые действия в конце концов выполняются; *б) справедливой в слабом смысле*, если условные неделимые действия в конце концов выполняются после того, как условие задержки стало и остается истинным, *в) справедливой в сильном смысле*, если условные неделимые действия в конце концов выполняются в ситуации, в которой условие задержки бывает истинным бесконечно часто.

Схема доказательства (Proof Outline)

Программа с добавленными утверждениями, достаточными для того, чтобы убедить читателя в ее корректности. В полной схеме доказательства утверждение ставится до и после каждого оператора.

Тройка (Triple)

Формула логики программирования вида `{ P } S { Q }`, где `P` и `Q` — предикаты, а `S` — список операторов. Интерпретация тройки следующая: если выполнение `S` начинается в состоянии, удовлетворяющем `P`, и заканчивается, то заключительное состояние удовлетворяет `Q`.

Ускорение (Speedup)

Отношение T_1/T_p, где T_1 — время выполнения последовательной программы на одном процессоре, T_p — параллельной программы на `p` процессорах.

Условная синхронизация (Condition Synchronization)

Тип синхронизации, при котором выполнение процесса приостанавливается до тех пор, пока не станет истинным некоторое булево условие `B`. Обычно это реализуется с помощью ожидания одним процессом события, о котором сигнализирует другой процесс.

Условное неделимое действие (Conditional Atomic Action)

Неделимое действие, которое должно быть отложено, пока некоторое булево условие `B` не станет истинным. Обычно оно программируется как `<await (B) S;>`.

Утверждение (Assertion)

Предикат, характеризующий состояние программы. Утверждение, помещенное перед оператором в программе, определяет, что этот предикат должен быть истинным каждый раз, когда программа готова к выполнению этого оператора.

Частичная корректность (Partial Correctness)

Свойство программы вычислять желаемый результат при условии, что она завершается.

Ядро (Kernel)

Набор структур данных и примитивных операций (непрерываемых процедур), который управляет процессами, распределяет их между процессорами и реализует взаимодействие высокого уровня и синхронизацию операций типа семафоров или обмен сообщениями.

Предметный указатель

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

P

С

Научно-популярное издание

Грегори Р. Эндрюс

Основы многопоточного, параллельного и распределенного программирования

Литературный редактор	*О.Ю. Белозовская*
Верстка	*В.И. Бордюк*
Художественный редактор	*А.А. Линник (мл.)*
Корректоры	*Л.А. Гордиенко, Т.А. Корзун, Л.В. Коровкина, О.В. Мишутина*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 31.12.2002. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 44,48. Уч.-изд. л. 37,5.
Тираж 3500 экз. Заказ № 2317.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.