СРЕДНЕЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Н. А. Староверова

ЛАНЬ

E.LANBOOK.COM

Н. А. СТАРОВЕРОВА

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебник

ЛАНЬ
· САНКТ-ПЕТЕРБУРГ · МОСКВА · КРАСНОДАР ·
· 2021 ·

УДК 004.451
ББК 32.972.11я723

С 77 **Староверова Н. А.** Операционные системы : учебник для СПО / Н. А. Староверова. — Санкт-Петербург : Лань, 2021. — 412 с. : ил. — Текст : непосредственный.

ISBN 978-5-8114-6385-5

В данном учебнике рассматриваются основные вопросы, связанные со структурой и развитием операционных систем. В основу учебника легли статьи, лекции и лабораторные работы, разрабатываемые в рамках дисциплин «Операционные системы» и «Системное программное обеспечение».

Внимание уделено таким темам, как история и перспективы развития операционных систем, структура, вопросы диспетчеризации, многопоточности.

В рамках лабораторных работ рассматривались принципы работы в операционной системе UNIX.

УДК 004.451
ББК 32.972.11я723

Обложка
П. И. ПОЛЯКОВА

ВВЕДЕНИЕ

Тенденции развития операционных систем

Как известно, операционные системы (Далее ОС), с этапа их появления еще в 60-х гг. минувшего века, изменились до неузнаваемости. Изменилась оболочка, манера программирования ОС, функциональность и возможность использовать, как для личных нужд, так и для систем управления на всевозможных производствах. Но, как и прежде, ОС — это неотъемлемая часть вычислительных машин и их систем — вычислительных систем.

Обычный пользователь даже и не представляет, что скрывается за всеми иконками на рабочем столе. Какие сложные процессы происходят одновременно в системе, позволяя читать новости в просторах интернета. Поэтому, прежде чем говорить о путях развития ОС, стоит упомянуть всю сложность создания новой и неповторимой ОС.

ОС — это громоздкое программное обеспечение, которое состоит из миллионов строк. Над разработкой новой системы трудятся целые команды специалистов в этой области, проводя за работой огромное количество времени. Это и не удивительно, так как ОС по большей степени это огромный инженерный проект, сравнимый с постройкой космической станции или авианосца. Огромные блоки программного кода, зачастую, не могут быть обособленными единицами и постоянно взаимодействуют с другими блоками информации, усложняя задачи, стоящие перед разработчиками.

На сегодняшний день существует довольно большое количество различных операционных систем. От долгожителей, таких как UNIX и его клоны, до совсем новых и малоизвестных систем. Вот далеко не полный список исследовательских и коммерческих операционных систем, созданных рядом крупных фирм:

- В 1960-х — 1970-х гг. фирма IBM разработала OS IBM 360/370. Следующей разработкой стала OS/2, предназначенная для персональных компьютеров. В настоящее время наиболее современными ОС этой фирмы являются z/OS и z/VM.
- Семейство ОС MacOS развивается с начала 1980-х гг. фирмой Apple. Оно характеризуется улучшенным графическим пользовательским интерфейсом.
- ОС Solaris развивается с начала 1980-х гг. фирмой Oracle/Sun. Эта ОС является развитием UNIX.
- Фирма Hewlett-Packard развивает собственную версию UNIX — систему HP/UX.
- Novell — одна из ведущих фирм в области сетевых технологий; развивает семейство сетевых операционных систем: NetWare; в настоящее время — Open Enterprise Server (сетевая ОС, включающая все сетевые возможности NetWare и возможности распространенного диалекта Linux — openSUSE) [1].

Исследовав различные современные операционные системы, можно выделить следующие основные направления развития ОС:

Графические оболочки. Графический пользовательский интерфейс имеют все современные ОС. Стоит отметить, что графические оболочки для всех ОС по возможностям приблизительно одинаковы, связано это с обостренной конкуренцией между фирмами-разработчиками. Из-за того, что оболочки для разных ОС похожи, пользователю порой бывает даже сложно определить, в какой именно ОС он работает. С одной стороны, эта схожесть удобна для конечных пользователей, так как она упрощает изучение рабочей среды. Но с другой стороны, использование только графических оболочек (без изучения командных языков и конфигурационных файлов) является минусом для системных программистов, так как снижает их уровень подготовки.

Основные возможности, предоставляемые графическими оболочками ОС:

- Удобный графический пользовательский интерфейс.
- Возможность выполнить любые системные настройки с помощью GUI. Особенно следует отметить в этом отношении графические оболочки ОС Linux.
- Поддержка новых тенденций в развитии интерфейсов — multi-touch, Tablet PC и др.
- Унификация графических оболочек для различных ОС. В разных ОС используются графические оболочки CDE, KDE, GNOME. Вероятно, в ближайшем будущем их список пополнится.

Поддержка новых сетевых технологий и Web-технологий. В настоящее время активно развиваются сети и Интернет, появляются новые стандарты и протоколы — Ipv6, HTML 5 (для облачных вычислений) и т. д. Цель современных ОС — возможность поддерживания всех новых сетевых технологий [2].

Развитие беспроводных сетей. Развитие высокопроизводительных беспроводных сетей отражается и в развитии операционных систем. Выделим следующие передовые сетевые технологии.

WiMAX — телекоммуникационная технология, разработанная для предоставления универсальной беспроводной связи на больших расстояниях для широкого спектра устройств. Максимальная производительность до 1 Гбит/с. Основан WiMAX на технологии Wi-Fi, но отличается своим дальнодействием.

3G — технологии мобильной связи 3 поколения с быстродействием до 14 Мбит/с. Основоположник 3G в России — компания «СкайЛинк».

4G — поколение мобильной связи с повышенными требованиями. К нему относятся перспективные технологии, позволяющие осуществлять передачу данных со скоростью, превышающей 100 Мбит/с — подвижным и 1 Гбит/с — стационарным абонентам, это позволяет обеспечить повышенное качество голосовой связи.

Также на сегодняшний день для выхода в Интернет используют цифровые телевизионные каналы, это происходит с помощью специальных устройств set-top boxes.

Усиленное внимание к механизмам безопасности и защиты. Безопасности уделяют большое внимание все современные операционные системы. Это отражается, например, в том, что браузеры при загрузке веб-страниц вы-

полняют их анализ на отсутствие фишинг (вид интернет-мошенничества, который стремится заполучить доступ к конфиденциальным данным пользователей: логинам и паролям), также при загрузках и установках программ из сети требуется только явное согласие пользователя. Уделяется такое внимание безопасности во многом благодаря Trustworthy Computing Initiative (инициативе надежных и безопасных вычислений) фирмы Microsoft, объявленной в 2002 из-за постоянно усиливающейся кибер-преступности [3].

Поддержка многопоточности и многоядерных процессоров. На сегодняшний день многоядерные процессоры распространились очень широко, в связи с этим все современные ОС имеют библиотеки программ, которые поддерживают эту возможность аппаратуры. Параллельное выполнение потоков становится возможным именно благодаря многоядерной архитектуре.

Поддержка распределенных и параллельных вычислений. Современные ОС имеют высокоуровневые библиотеки, которые позволяют разрабатывать параллельные алгоритмы решения задач с использованием возможностей аппаратуры. Они поддерживают следующие основные виды, стандарты и инструменты параллелизма:

- OpenMP — механизм написания параллельных программ для систем с общей памятью.
- MPI (Message Passing Interface) — программный интерфейс, необходимый для параллельного выполнения программ, взаимодействующих с помощью передачи сообщений.

Виртуализация ресурсов и аппаратуры. Современные ОС содержат средства виртуализации. Процесс виртуализации представляет собой запуск специализированного программного обеспечения под операционной системой, называемой хостом (Host OS), дающего возможность создавать виртуальные машины (Virtual Machine), обладающие заданными характеристиками реальных компьютеров, и запускать на них независимо друг от друга различные гостевые операционные системы (Guest OS). Виртуализация удобна тем, что ресурсы между операционными системами распределятся довольно быстро [4, 5].

Развитие файловых систем. Это направление необходимо для защиты информации и существенного увеличения размера файлов (для мультимедиа). Мультимедийная информация обрабатывается таким образом, что старые файловые системы не могут вместить в себя мультимедийные файлы для хранения. Например, максимальный размер файла в системе FAT — 4 гигабайта — легко может быть превышен при переписи на компьютер цифровой видеопленки длительностью 10–15 минут. В связи с этим разрабатываются такие новые файловые системы, которые допускают хранение очень больших файлов, например, система ZFS в ОС Solaris.

Поддержка облачных вычислений — это новое направление в развитии ОС, его основоположником является «облачная» ОС Windows Azure фирмы Microsoft [6].

Перспективы развития операционных систем

Цель создания ОС — это основа дальнейшей разработки проекта. Если нет четкого представления о том, чего разработчики хотят получить, то очень сложно говорить об успешности готового продукта. Но на данный момент ОС используются на вычислительных машинах в разных областях и науки, и производства, поэтому сложно предугадать, как будет использована система в дальнейшем. Можно сделать вывод, что система должна быть универсальна и на это стоит уделять больше внимания и сил.

Как уже говорилось ранее, ОС — сложная система, в которой выполняется множество различных операций одновременно. Особенно операций ввода-вывода для нескольких устройств. Если раньше в системе MS-DOC нельзя было подключить одновременно звуковую карту и сетевую плату, так как это вызывало конфликт системы, то сейчас в каждом современном персональном компьютере находится десяток таких плат расширения возможностей. Поэтому очень важно, чтобы в системе был отлично проработан параллелизм. Он позволит выполнять множество операций, предотвращая тупиковые ситуации и соперничество за используемую память.

К сожалению, одной из причин сложного создания новой ОС являются недоброжелательные пользователи, готовые взломать систему для получения своей собственной выгоды. Такая ситуация ставит перед разработчиками еще одну задачу — защита системы и пользовательских данных. Однако не всегда это помогает, так как ежедневно в сети все сильнее развиваются вирусы, способные пройти даже через самые изощренные системы защиты.

При разработке новой системы также учитывают совместимость с предыдущими версиями. Многие системные характеристики, как длина имени файлов и папок, считаются проектировщиками ОС уже устаревшими, но отказаться от них не представляется возможным.

Не стоит забывать и о конкуренции. Известные гиганты в ОС Linux и Microsoft уже не один год занимают лидирующее положение. Их преимущество — это взгляд в будущее с возможностью широкого изменения структуры системы, на что некоторые разработчики даже не обращают внимания. Поэтому многие молодые системы не долго находятся на арене перспективных ОС.

Все это и еще другие не менее важные аспекты усложняют появление новых ОС.

Остановимся на ОС 2000-х гг. В чем они преуспели, а чего еще нужно достичь?

Взглянув на ОС WINDOWS 8, можно с уверенностью сказать, что разработчики из Рэдмонда решились изменить привычное представление об ОС, начиная с перемены классического интерфейса. Теперь рабочее пространство пользователя больше напоминает веб-сайт, чем привычный рабочий стол. Заметно увеличилась и расширилась интеграция всего программного стека Microsoft с Сетью. Office и сама учетная Windows Live становятся облачной платформой, а также появляется центральный сервис по хранению данных SkyDrive.

Самое главное то, что компания Microsoft намекает нам на скорый отказ от классических программных технологий десктопного программирования и продвигают более новые, которые раньше использовались только в веб-программировании — это JavaScript и HTML. Это очень сильно настораживает разработчиков настольных приложений на платформе Windows, так как многие из них теряют свои позиции на рынке и становятся новичками, теряя весь наработанный годами опыт. Такое развитие событий касается не только индивидуальных разработчиков, но и огромных компаний.

С чем связана такая политика Microsoft? Неужели разработчики, как и в компаниях Google или Facebook, понимают, что уже в скором будущем сама суть классического десктопного прикладного программирования и ОС в целом в корне изменится?

Сначала все прикладные приложения и программы были стационарными. Они не требовали обязательного подключения к Интернету для работы с ними. Обработка и хранение данных были ограничены лишь ресурсами персонального компьютера, на котором происходил запуск данной программы. Поэтому ОС была единственным возможным создателем (хостом) приложения, целиком и полностью предоставляя среду для работы с программой, запущенной пользователем.

В таком случае выбор ОС имел колоссальное значение, так как все приложения разрабатывались только под определенную ОС. Пользователь, при данной ситуации, выбирал определенный набор программ, доступный для использования вместе с ОС.

Развитие локальных сетей должно было изменить такую ситуацию. Однако, ничего не изменилось. Обработка пользовательской задачи распределялась на некоторое количество машин, для которых стоило выбирать ОС и набор программ по аналогичному принципу.

Продвижение в ходе сложившегося случая появилось вместе с развитием скоростного Интернета. Появились новые веб-сервисы, направленные на замену десктопного софта. Связь между серверами и пользователями преграждалась лишь стандартным набором протоколов Интернета. Такое нововведение позволило снизить системные требования к пользовательской машине, а также выбор ОС больше не влиял на выбор сервисов, которые стали работать на технологиях, не зависящих от платформы — это JavaScript, XML и HTML+CSS.

Хост приложений разделился на две части: клиентская и серверная. ОС перестала быть клиентским хостом, ее место занял веб-браузер, который не сильно зависит от характеристик самой ОС. Это расширило возможность пользователям выбирать свою ОС. Именно по этой причине значительную часть рынка смогла захватить ОС X.

В свою очередь, перенос части хоста на серверную платформу способствовал упрощению работы самим разработчикам. Больше не нужно было брать в расчет мощность персонального компьютера пользователя, что позволяло пользоваться огромным объемом информации каждому клиенту в частности. Вместе с этим стали создаваться приложения, существование которых было ранее сложным или вообще невозможным. Изменился и критерий выбора ОС, т. е.

отпала необходимость выбирать разработчикам такую же ОС, что и у пользователей.

Чтобы наглядно представить перемены, рассмотрим следующую ситуацию. Досуг для среднестатистического пользователя домашнего персонального компьютера. Что ему необходимо? Несложный текстовый редактор для написания какой-либо статьи в свой блог или заметок на будущее. Табличный процессор для ведения домашней бухгалтерии и расчета семейной зарплаты за месяц. Приложения для прослушивания аудиозаписей, просмотра фотографии и видео, для общения с другими людьми. Конечно же, браузер для просмотра новостей.

Когда эти приложения и программы были десктопными и запускались на персональном компьютере пользователя, очень важна была и задача работы с файлами. Такая ситуация ушла в прошлое, весь перечень приложений доступен в веб-пространстве, причем бесплатно. Это очень сильно упрощает работу с ОС и экономит приличное количество денежных средств.

Стоит отметить, что при переустановке ОС также отпадает надобность в перенастройке всех приложений веб-сервиса, увеличивая мобильность системы.

Работа с файлами уходит на задний план с веб-ресурсами. Все файлы теперь хранятся на серверах определенных сервисов или в едином пользовательском облачном пространстве, например, Яндекс.Диск. В итоге получается, что локальная файловая система может понадобиться только для временного хранения файлов.

Еще остается возможность открыть браузер, чтобы использовать облачные сервисы. На этом основные потребности среднестатистического пользователя от ОС заканчиваются.

Исключая трату денежных средств, пользователю остается взять бесплатную ОС. Именно такую нишу ОС пытается заполучить Google с их Chrome OS — главным конкурентом Windows.

Чтобы не проиграть, компания Microsoft пробует создать из ОС онлайн-портал. Новый плиточный интерфейс нацелен на то, чтобы выводить на такой плитке актуальную информацию из сети Интернет. Приложение занимает полностью весь экран и не требует дополнительного инструментария, а процесс работы напоминает работу с сайтом.

Microsoft добивается сотворить то же, что и Google, — ОС-браузер, являющуюся хостом для веб-приложений, интегрированных с Сетью. Это и есть верное и перспективное направление ОС.

Как и переход от консольных приложений к GUI, переход всех приложений в облачный сервис очень долгий и сложный. Но рано или поздно веб-технологии смогут достичь определенного уровня и такой переход осуществится. Кто успеет подготовиться к такому, тот займет лидирующие позиции, иные останутся не у дел. Такие переходы в истории программной индустрии неоднократны.

Исходя из всего описанного, можно сделать вывод, что ОС продолжают жить и развиваться. Переход от одних методов к другим изменяют ОС, но по-прежнему ОС будут оставаться основой каждой вычислительной машины.

Перспективы развития ОС. Таким образом, операционные системы — это перспективное, активно развивающееся направление. Перечислим основные перспективы развития операционных систем.

Развитие в направлении к интеграции ОС (на уровне графических оболочек, а так же на уровне общего ядра); развитие семейств ОС на основе модулей общего кода.

Значительное повышение надежности, безопасности и отказоустойчивости ОС; разработка ОС на управляемом коде или его аналогах.

Дальнейшее развитие проектов по ОС с открытым кодом.

Развитие виртуализации: необходимо обеспечить возможность выполнить или эмулировать любое приложение в среде любой современной ОС.

Дальнейшее сближение по возможностям ОС для настольных компьютеров и ОС для мобильных устройств.

Дальнейшее объединение ОС и сетей.

Перенос ОС и базовых инструментов в среды для облачных вычислений.

Обзор современных компьютерных архитектур

Когда мы говорим об операционных системах, необходимо помнить, что ОС непосредственно связана с архитектурой компьютера, и поэтому хоть наш курс и посвящен рассмотрению операционных систем, нам необходимо посвятить время тому, чтобы рассмотреть, хотя бы обзорно, современные компьютерные аритектуры.

Что же собой представляет архитектура компьютера? Это набор характеристик, операций и типов данных каждого уровня. Можно выделить шесть уровней организации компьютера. Первый, он же логический, уровень представляет собой аппаратное обеспечение машины, состоящей из вентилей. Второй уровень, микроархитектурный, это интерпретация (микропрограммы) или непосредственное выполнение. Совокупность регистров процессора составляет локальную память, а электронные схемы выполняют машинно-зависимые программы. Третий уровень — это уровень архитектуры системы команд, трансляция (ассемблер). Четвертый уровень связан с третьим, так как он состоит из операционной системы и трансляции (ассемблера). Еще его называют гибридным уровнем, так как одна часть команд объясняется операционной системой, а другая — микропрограммой. Пятый уровень — уровень языка ассемблера, трансляция (компилятор). Этот уровень и выше предназначены для написания прикладных программ, с 1–3 — системных программ, для человека удобен как раз этот вид программ. И наконец, шестой уровень, язык высокого уровня, программы на этом уровне транслируются обычно на уровнях 3 и 4. Сама архитектура связана программными аспектами. В свою очередь, аспекты реализации не являются архитектурой компьютера. Например, технология, применяемая при реализации памяти.

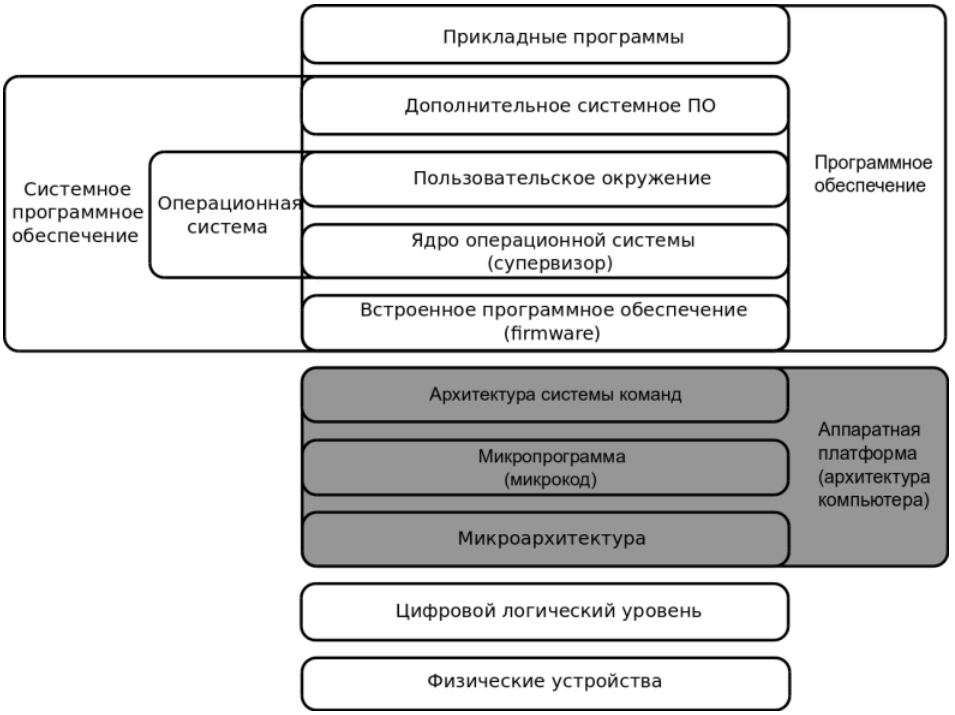


Рис. 1
Схема, иллюстрирующая многоуровневую структуру компьютера

Классификация компьютерных архитектур

CISC (Complicated Instruction Set Computers — компьютеры с усложненной системой команд) — данный подход является одним из первых к архитектуре компьютера. Он подразумевает то, что в систему команд компьютера включаются трудные по семантике операции, которые реализуют типовые воздействия, которые, в свою очередь, часто применяются при программировании и при реализации языков. Например, вызов рекурсивных процедур и автоматическое обновление дисплей-регистров, массовые операции пересылки строк и массивов. Стандартными адептами **CISC**-компьютеров были машины серии **IBM 360/370** и вычислительные комплексы (МВК) **«Эльбрус»**. В IBM 360 была использована команда MVC (move characters), создающая пересылку массива знаков (строки) из одной области памяти в иную, при этом адреса источника, получателя и длина пересылаемой строки задавались в регистрах. В «Эльбрусе» был аппаратно реализован в общем виде вход в функцию с последовательной передачей сквозь стек характеристик, обновлением дисплей-регистров, которые указывают на дешевые процедуры области локальных данных. Также в «Эльбрусе» команда считывания в стек смысла по данному адресу воплотила в жизнь самодействующий проход «косвенной цепочки» заблаговременно незнакомой длины. То есть если значение вдруг оказывалось и адресом, то происходило считывание в стек значения

по нему и так продолжается до тех пор, пока считанный в стек размер не окажется значением, а не адресом. На мой взгляд, стремление создателей CISC-архитектур устроить аппаратуру как можно более «умной» — это удача. Но строгое «вшивание» трудных алгоритмов выполнения команд в «железо» приводило к тому, что аппаратура исполняла каждый раз некоторый общий алгоритм команды, требовавший десятков или же сотен тактов микропроцессора. Что усложняет работу компьютера в целом. К сожалению, как-нибудь улучшить выполнение этих команд с внедрением определенной информации о длине строки, косвенной цепочки и тому подобному способности не было. Еще один провал CISC-архитектур в том, что похожие массовые операции на время их выполнения практически останавливали работу потока (pipeline) — реализованной в любой компьютерной архитектуре аппаратной оптимизации, параллельного выполнения нескольких примыкающих команд при условии их независимости друг от друга по сведениям.

RISC (Reduced Instruction Set Computers — компьютеры с облегченной системой команд) — самый простой подход к компьютерной архитектуре, который предложили вначале 1980-х гг. доктор Дэвид Паттерсон и его студент Дэвид Дитцел. Примеры семейств RISC-компьютеров: SPARC, MIPS, PA-RISC, PowerPC. Основа данного расклада такова. Во-первых, упрощение семантики команд. Во-вторых, отсутствие трудных массовых операций. В-третьих, одна и та же длина команд. В-четвертых, выполнение арифметических операций только в регистрах и внедрение особых команд считывания из памяти в регистр и записи из регистра в память. В-пятых, недоступность предназначенных регистров. В-шестых, внедрение большущего комплекта регистров (регистрового файла) совместного назначения — 512, 1024, 2048 регистров и т. д., в зависимости от определенной модели процессора. И в заключении, предоставление при вызове процедур характеристик сквозь регистры. Точно такая же архитектура выделяет размашистый простор для оптимизаций, производимых компиляторами, то есть, что и показывают компиляторы Sun Studio. RISC — такая архитектура, которая до сих времен применяется при разработке свежих компьютеров.

VLIW (Very Long Instruction Word — компьютеры с широким командным словом) — данный подход образовался в 1980–1990-х гг. Главная мысль предоставленного расклада — это статическое планирование параллельных вычислений компилятором на уровне отдельных последовательностей команд и подкоманд. При предоставленной архитектуре любая команда обозначается «широкой» (long) и содержит какое-либо количество подкоманд, производимых параллельно за один машинный такт на нескольких однотипных устройствах микропроцессора. Задачей для компилятора считается подходящее планирование загрузки всех данных приборов в любом машинном такте и генерация команд, которые позволили бы нормально загрузить на любом такте любой из приборов. Достоинство этой архитектуры — вероятность распараллеливания вычислений, а дефект — сложность. Примеры компьютеров этих архитектур: Cray X/MP, Cray Y/MP, многопроцессорный вычислительный ансамбль «Эльбрус-3».

EPIC (Explicit Parallelism Instruction Computers — компьютеры с очевидным распараллеливанием) — по архитектуре подобны VLIW, но при условии с наличием ряда ярчайших усовершенствований. Например, спекулятивных вычислений — параллельного выполнения обеих ветвей относительной системы с вычислением условия. Расклад сформировался и применяется с 1990-х гг. Примеры микропроцессоров предоставленной архитектуры — Intel IA-64, AMD-64.

Multi-core computers (многоядерные компьютеры) — компьютерная архитектура, в которой любой микропроцессор содержит некоторое количество ядер (cores), соединенных в одном кристалле и параллельно работающих на одной и той же совместной памяти, что предоставляет широкие способности для параллельных вычислений. Сейчас популярны многоядерные процессоры компании Intel (Core 2 Duo, Dual Core и др.) и массивные многоядерные процессоры компании Sun/Oracle: Ultra SPARC-T1 («Niagara») — 16-ядерный процессор; Ultra SPARC-T2 («Niagara2») — 32-ядерный микропроцессор. Разработчики операционных систем для этих компьютеров создают базисные библиотеки программ, дозволяющие в абсолютной мере применить способности параллельного выполнения на многоядерных процессорах.

Hybrid processor computers (компьютеры с гибридными процессорами) — свежий, все обширнее распространяющийся подход к архитектуре компьютеров, при котором микропроцессор содержит гибридную структуру — произведено из (многоядерного) центрального микропроцессора (CPU) и (также многоядерного) графического микропроцессора (GPU — Graphical Processor Unit). Эта архитектура была разработана в связи с потребностью параллельной обработки графической и мультимедийной информации, тем более животрепещуще для компьютерных игр, при просмотре на компьютере качественного цифрового видео и др. Гибридная архитектура считается свежим «умственным вызовом» для создателей компиляторов, коим нужно создать и воплотить в жизнь адекватный комплект оптимизаций как для центральных, так и для графических микропроцессоров. Примерами этих архитектур считаются свежие микропроцессоры компании AMD, а еще графические микропроцессоры серии Tesla компании Nvidia.

Микроядерные ОС

Микроядерным Операционным Системам (ОС) уделяется большое внимание разработчиками и исследователями. Эти системы дают жесткое разделение исполняемых модулей, вынесение всех служб из ядра в пространство пользователя, позволяют повышать отказ системы и саму ее безопасность. Хотя существуют и издержки, которые проявляются в том, что поддержка такой структуры требует дополнительных вычислительных ресурсов. Данное противоречие обязывает ученых и разработчиков находить все новые и новые архитектурные решения для минимизации негативных и усиления позитивных сторон микроядерных ОС.

За последнее время в технологиях вычислительных систем произошел колоссальный рывок вперед, так, например, в мобильных устройствах присутствуют процессоры с частотой большей 1 GHz и модули оперативной памяти объемом 512–1024 MB. Помимо этого, внутренняя архитектура программного обеспечения стала значительно сложнее, появились большие программные стеки изолированных модулей, осуществляющих интенсивный обмен данными. При просмотре видео в Интернете, в браузере выполняется код flash проигрывателя. Этот модуль взаимодействует с библиотеками кодека, декодирующего видео, с оконным менеджером, с графической подсистемой, с ядром ОС.

Данная архитектура основана на идее жесткого разделения исполнимых модулей и коммуникации между ними. Если в системе данного типа ядро будет построено в соответствии с микроядерной идеологией, то нужно будет добавить дополнительный уровень абстракции, связанный с аппаратным вводом-выводом. Дополнительный IPC, возникающий от микроядра, незначителен. Если учитывать архитектуру системы в целом, а не только ядра и его компонентов, можно сказать, что идея использования микроядерных ОС может стать вновь актуальной.

Первое поколение микроядер. Самым первым поколением микроядерных ОС называют системы, созданные в 1980-х гг. Известными проектами являются ядро Mach и ОС QNX. Первый проект был в области архитектуры ОС, осуществленный командой ученых в конце 1980-х — начале 1990-х. Данное ядро было медленным и поддерживало только архитектуру x86. В настоящее время GNU/Mach является частью ОС GNU/Hurd и развивается при поддержке компании Debian.

Второе поколение микроядер: L4. Jochen Liedtke в конце 1990-х предложил новую архитектуру микроядра, сначала L3, потом L4. Архитектура подразумевала синхронный IPC, минимальный функционал ядра, вынесение всех политик в пространство пользователя. Стремление уменьшить размер ядра ставило задачу исполнения кода ядра из кеша процессора.

L4 — это идеология, и она оформлена в документах L4 eXperimental reference manual. На базе данной архитектуры может быть запрограммирована ОС.

Появилось много проектов, за последнее десятилетие было больше двух десятков.

L4Ka, Karlsruhe Institute of Technology (KIT). ОС L4Ka разрабатывалась командой Йохена Лидтке на базе архитектуры L4. Ядро в этой ОС называлось Hazelnut и построено в соответствии с версией X.0 L4 API. Потом ядро, построенное в соответствии с ревизией X.2 получило название Pistachio. Данное ядро существует и в наше время, но его развитием никто сейчас не занимается, можно сказать, что оно умерло. В дальнейшем ядро Pistachio получило распространение, и на его базе было создано несколько коммерческих проектов.

Обзор операционных систем, разработанных по принципу микроядра

Вычислительную систему, которая работает под управлением операционной системы, основанной на ядре, как правило, принято рассматривать как сис-

тему из трех слоев, расположенных иерархически. Первый и внутренний слой — аппаратура, второй и промежуточный слой — ядро ОС, третий и, соответственно, внешний слой — утилиты, системные обрабатывающие программы, библиотеки.

Каждый из слоев имеет возможность вести взаимодействие лишь с соседними, то есть ближайшими слоями. Таким образом, при такой организации операционной системы приложения не имеют возможности напрямую связываться с аппаратурой, а лишь только через слой ядра.

Так как ядро считается сложным многофункциональным комплексом, то и на структуру ядра распространяется многослойный подход.

Ядро условно может состоять из следующих слоев:

Первый слой — средства аппаратной поддержки ОС — это та часть аппаратуры, которая работает напрямую с ОС (средства поддержки привилегированного режима, система прерываний, процессов, средства защиты областей памяти).

Второй слой — машинно-зависимые части ОС. На данном слое происходит формирование программных модулей, которые отражают особенности компьютерных аппаратных платформ. Этот слой осуществляет защиту верхних слоев ядра от аппаратной зависимости.

Третий слой — основные механизмы ядра. Задачей этого слоя является выполнение самых массовых операций ядра, таких как процессы переключения программного обеспечения, прерывания планирования, перемещения страниц из памяти на диск и обратно и т. п.

Четвертый слой — менеджеры ресурсов. Модули, из которых состоит этот слой ядра, реализуют стратегические цели для управления основными ресурсами вычислительной системы.

Пятый и самый верхний слой ядра — интерфейс системных вызовов. Этот слой напрямую ведет взаимодействие с приложениями и системными утилитами, тем самым создавая интерфейс прикладного программирования операционной системы.

Микроядерная архитектура считается альтернативой традиционному способу построения операционной системы, в которой мультислойное ядро выполняется в привилегированном режиме.

Микроядро — это важнейший принцип разработки операционной системы. Данный принцип заключается в том, что из системного в пользовательское «пространство» переносится такое число модулей, которое только максимально возможно. Иными словами, большинство модулей операционной системы выполняются в пользовательском режиме. Процесс обмена информацией между модулями в пользовательском режиме осуществляется путем отправки сообщений.

У микроядерных операционных систем существует целый ряд достоинств и недостатков. К *преимуществам* операционных систем, созданных по принципу микроядра, можно отнести переносимость, расширяемость, надежность и поддержку распределенных приложений. А основным *недостатком* таких операционных систем будет считаться низкая производительность.

В микроядре обособлен весь машинно-зависимый код. Вследствие чего, для того чтобы перенести систему на новый процессор, будет требоваться меньше изменений, и все они вместе будут логически объединены в группы. Это связано с высокой степенью переносимости микроядерных операционных систем.

Высокая степень расширяемости — это одно из достоинств операционных систем с микроядерной организацией. В традиционных системах с мультислойной структурой довольно непросто, а быть может, даже достаточно трудно удалить один слой или заменить его другим. Для того чтобы добавить новые функции и изменить уже ранее существующие, требуются хорошие знания операционной системы и большие затраты времени. В то же время локальный набор определенных интерфейсов микроядра имеет основополагающее значение для организованного роста и развития ОС. Для добавления новой подсистемы требуется разработка нового приложения. Это никоим образом не воздействует на целостность микроядра. Микроядерная архитектура позволяет добавлять и уменьшать количество компонентов операционной системы.

Надежность операционной системы повышается за счет применения микроядерной модели. В традиционных операционных системах все модули ядра имеют возможность воздействовать друг на друга, а в микроядерных ОС каждый сервер способен выполняться в виде отдельного процесса в своей собственной области памяти. Это позволяет серверу защититься от других серверов операционной системы. При выходе из строя одного сервера, он перезапускается без ущерба для остальных серверов. Все серверы работают в режиме пользователя, вследствие чего они не имеют способности прямого доступа к аппаратуре и не способны вносить изменения в память, в которой хранится и работает само микроядро непосредственно. Другим способом повышения надежности операционной системы может являться объем кода микроядра по сравнению с традиционным ядром.

Поскольку структура с микроядром использует механизмы подобные сетевым: клиенты и серверы ведут взаимодействие методом обмена сообщениями, — микроядерные операционные системы отлично подходят для поддержки распределенных вычислений. Серверы такой операционной системы могут работать на одном или на разных компьютерах. В таком случае, когда микроядро получает сообщение от приложения, оно может обработать его само и передать локальному серверу или же отправить по сети микроядру, которое работает на другом компьютере. Для перехода к распределенной обработке требуется минимум изменений в работе ОС. В таком случае локальный транспорт заменяется на сетевой.

При традиционной организации операционной системы выполнение системного вызова будет сопровождаться двумя переключениями режимов, а при микроядерной организации — четырьмя. Иначе говоря, операционная система, которая разрабатывается по принципу микроядра, всегда будет менее производительной, чем ОС с классическим ядром.

Различные микроядра имеют разные возможности и размеры, поэтому невозможно сформулировать общие правила, которые смогли бы определить

структуру микроядра и набор его функций. Микроядро включает в себя те функции, которые напрямую зависят от аппаратного обеспечения, а также те, которые необходимы для поддержки серверов и приложений, работающих в пользовательском режиме. Данные функции относятся к общим категориям функций управления памятью низкого уровня, связи между процессами, и управления вводом-выводом и прерываниями.

Управление памятью низкого уровня. В микроядре должен обеспечиваться контроль над аппаратным представлением адресного пространства для того, чтобы в нем было возможно осуществление защиты на уровне процессов. В случае если микроядро станет отвечать за отображение каждой виртуальной страницы в физический страничный блок, то блок управления памятью, охватывая систему защиты адресного пространства одного процесса от другого, а также способ замены страниц и другие логические схемы страничной организации памяти, можно реализовать за пределами ядра.

Взаимодействие между процессами. Взаимодействие между процессами и потоками в микроядерной операционной системе осуществляется при помощи сообщений. Такие сообщения состоят из заголовка с указанием идентификатора процесса, который будет являться отправителем, и процесса, который будет являться адресатом, то есть получателем, а также из тела с передаваемыми данными, указателем на блок данных или некоторыми управляющими сведениями о процессе. Чаще всего взаимодействие между процессами основано на портах, которые относятся к этим процессам. В данном случае порт будет выступать очередью сообщений, которые предназначены для какого-то определенного процесса. С портом связан список возможностей, указывающий, с какими процессами у данного процесса возможен обмен информацией. После того, как процесс разрешает доступ к себе, он отправляет сообщение ядру, в котором указана новая возможность порта.

Одна область памяти копируется в другую при передаче сообщения от одного процесса другому, если выполняется следующее условие: адресные пространства двух этих процессов не должны перекрываться. Отсюда следует, что скорость такой передачи сообщения будет связана со скоростью работы памяти, которая несоизмеримо меньше, чем скорость процессора. Вследствие чего в современных исследованиях наблюдается интерес к межпроцессному взаимодействию, основанному на потоках, и к таким схемам разделения памяти, как многократное отображение страниц.

Управление вводом-выводом и прерываниями. В архитектуре микроядра возможна обработка аппаратных прерываний аналогично сообщениям, а также возможно включение портов ввода-вывода в адресное пространство. Такое микроядро распознает прерывания, но не обрабатывает их. Оно создает сообщение процессу, который совершает работу на пользовательском уровне и связан с данным прерыванием. Отсюда следует, что ядром поддерживается такое отображение, когда разрешенное прерывание сопоставляется с процессом на пользовательском уровне. Микроядром должно выполняться превращение прерываний в сообщения, однако ядро не принимает участия в обработке аппаратно-зависимых прерываний.

Проекты NICTA Group

NICTA Group — Information and Communications Technology (ICT) Research Centre of Excellence в Австралии. Nicta это есть государственная организация, объединяющая ведущие вузы Австралии и специализирующаяся на узком направлении исследований. Сюда входят научные проекты в области системного программирования, управления, искусственного интеллекта и распознавания образов, оптики и наноэлектроники — наиболее перспективных направлений начала XXI-го века. Была создана компания OKL (Open Kernel Lab), занимавшаяся разработкой коммерческих продуктов на основе микроядра Pistachio-embedded. В 2007 г. компания OKL анонсировала собственно микроядро OKL4, и это микроядро получило большое распространение в коммерческих продуктах в партнерстве с Qualcomm. В 2012 г. OKL4 продали компании General Dynamics.

L4.verified — верифицированное ядро на архитектуре L4. Научная разработка NICTA Group, трек у этого проекта аналогичен проекту OKL. Идея данного проекта есть в математическом доказательстве корректности реализации ядра. Микроядерные операционные системы, имеющие минимальный размер, могут быть верифицированы. По каждой строчке кода можно будет написать выражение, функции перевести в теоремы, и дальше идет доказательство. Для L4.verified было доказано соответствие реализации модели и отсутствие вечных циклов. Стоимость данного проекта была $ 6 млн, продолжительность — 7 лет. Объем работы оценили в 25 человеческих лет.

seL4 — еще один проект NICTA, и он является ОС для систем безопасности. Информации мало, но есть возможность скачать данные образы.

Проекты Technische Universitat Dresden (TUD)

Группа под руководством Hermann Hartig в TUD участвовала в разработке микроядерных ОС совместно с Йохеном Лидтке. Данной группой была создана собственная реализация архитектуры L4 в микроядрах L4.Sec, Fiasco/L4.Fiasco и Fiasco.OC. Ими была разработана ОС реального времени DROPS (Dresden Real-Time Operating System Project).

Современные микроядра

Fiasco.OC микроядро, написанное на языке программирования C. Поддерживает архитектуры ARM, x86, ppc, sparc v8, современные отладочные средства и платформы Pandaboard и Beagleboard. Самостоятельно микроядро действовать не может, значит, у него должно быть окружение — набор прикладных программ и модулей, реализующих полезный функционал. Его окружением является L4Re (L4 Runtime Environment). В его состав входят паравиртуализированный L4Linux, который позволяет использовать скомпилированное для Linux прикладное ПО, DDEKit — набор исходного кода (wrapper), который позволяет использовать драйверы ОС Linux, большой набор библиотек и прикладных программ, адаптированных для запуска поверх микроядра. Стоит выделить TCP/IP Стек LwIP, Nitpicker и Mag. Они позволяют создавать на основе микроядра Fiasco.OC современные аппаратно-программные комплексы.

Genode Framework является Open Source проектом выпускников TUD. Перед разработчиками стояла задача создания унифицированного окружения для множества разных микроядерных проектов. Важными качествами являются наличие доверенного графического интерфейса пользователя (GUI), поддержка Qt. Аналогично присутствует поддержка L4Linux для ядра OKL4. Оно содержит собственное ядро с минимальным функционалом, называемым base-hw. Genode стал способен совершать сборку и отладку окружения. Genode поддерживает механизмы аппаратной виртуализации ARM TrustZone, это позволяет использовать этот фреймворк для создания доверенных решений.

Микрогипервизор NOVA (NOVA OS Virtualization Architecture) является гипервизором первого порядка, запускается напрямую на аппаратной платформе. Разработан выпускниками TUD и развивается в Intel. В его основе находится микроядерная ОС архитектуры L4. Архитектура системы такова, что позволяет исполнять прикладное ПО параллельно с виртуальными машинами. На основе гипервизора NOVA можно создавать доверенные решения, в которых доверенное ПО будет исполняться в отдельном окружении. Genode NOVA поддерживает собственное окружение NOVA User Land (NUL).

Проект MINIX3. MINIX3 являлся учебным проектом по разработке ОС. Эндрю Таненбаум, разработчик системы, является автором множества учебников по архитектуре вычислительных и ОС. Особое внимание в микроядре MINIX3 сделан на отказоустойчивости и безопасности. Данная система обладает многофункциональным графическим интерфейсом пользователя (GUI) и может быть использована в качестве настольной (Desktop) системы. Сейчас MINIX3 продолжает свое развитие.

Принцип уровней абстракций при разработке операционных систем

В современном мире происходит интенсивная разработка программного обеспечения, что повышает требования к подготовке будущих инженеров-программистов. Один из аспектов, который требует пристального внимания, это изучение рефлексивных и абстрактных способов мышления, особенно в курсах, посвященных гуманитарным аспектам программирования. В предлагаемой статье рассматривается уровень абстракций, который достаточно широко применяется в программировании.

Один из путей приучения будущих специалистов к абстрактному мышлению — это изучение конкретных примеров программного обеспечения с целью понимания имеющихся там уровней абстракции.

Рассмотрим первоначально общие понятия, а затем, конкретный пример использования данного механизма в разработке операционных систем.

Уровень абстракции представляет собой способ сокрытия деталей реализации определенного набора функциональных возможностей, который применяется для управления сложной проектируемой системы. При этом система подразделяется на несколько уровней. Детали нижних уровней скрываются, дабы обеспечить верхним уровням более простые модели [4].

На самом нижнем уровне общение программиста с машиной осуществляется с помощью нулей и единиц. Так как это не совсем удобно, был изобретен более высокий уровень абстракции — язык ассемблера [5]. Но и этот язык оказался не совсем удобным в использовании, так как представлял собой символическую форму двоичного языка компьютера. Поэтому был создан еще более высокий уровень абстракции — язык программирования высокого уровня (ЯВУ). Сейчас уже насчитывается около сотни таких языков. К примеру, Basic, C, C++, Cobol и RPG.

И все-таки, первые языки программирования позволяли программисту абстрагироваться от конкретных машинных деталей и уменьшить количество уровней абстракции. Именно в этом причина их успеха. Фактически эти языки позволяли программисту работать почти на том же уровне абстракции, какой был в прикладной задаче. Дело в том, что задачи решались хорошо математически определенные, сложные в основном по логике, а математические объекты были самые элементарные — переменная, вектор, матрица. Для тех времен такой абстракции было вполне достаточно — и достаточно сейчас для решения большинства вычислительных задач.

При таком подходе программирование — это перенесение логики алгоритма на язык программирования, не более того. Все сводится к логике.

Крупным достижением тех времен были подпрограммы [6]. Вызывая подпрограммы, программист абстрагируется от логики их выполнения. Если удачно разбить программу на подпрограммы, в каждой из них есть возможность работать на одном уровне абстракции — скажем, подпрограмма перемножения матриц имеет дело только с элементами. В головной же программе программист работает уже с матрицами, не опускаясь до уровня элементов, т. е. количество уровней абстракции уменьшается (точнее говоря, уменьшается не само количество этих уровней, а количество уровней, с которыми работает в данный момент программист). Именно в этом главное преимущество подпрограмм, но не все это понимают, и потому выделяют в подпрограмму обычно просто повторяющиеся действия, не связывая их с понятием абстракции. Возможно, это тоже преимущество подпрограмм, но экономия сил в данном случае — не главный выигрыш. Если не выделен абстрактный объект, который «прячется» за подпрограммой, то эту подпрограмму ждет нелегкое будущее — она либо не будет использоваться, либо будет бесконечно переделываться.

Если же понимать подпрограмму с точки зрения абстрагирования, то мы неизбежно приходим к иерархическому построению программы, соответствующему последовательному раскрытию абстрактных объектов. Майерс приводит следующие свойства этих уровней [7]:

1. На каждом уровне не известно о свойствах и даже о существовании более высоких уровней.

2. На каждом уровне ничего не известно о внутреннем строении других уровней.

3. Каждый уровень представляется группой модулей.

4. Каждый уровень располагает определенными ресурсами, которые скрывает от других уровней или представляет им некоторые их абстракции.

5. Каждый уровень может обеспечивать некоторую абстракцию данных в системе.

6. Предположения, которые на каждом уровне делаются относительно других уровней, должны быть минимальны.

7. Связи между уровнями ограничены явными аргументами.

8. Каждый уровень должен иметь высокую прочность и слабое сцепление.

Таким образом, прежде чем проектировать систему, нужно представить ее иерархически. Это прямо вытекает из организации человеческого мозга. Он может манипулировать ограниченным количеством объектов (не более семи) и к тому же способен эффективно понимать одновременно не более двух уровней абстракции — текущий и предыдущий. Поэтому, если при написании программы углубиться в детали, чтобы реализовать некий сегмент, то потом, выйдя из него, с трудом можно вспомнить, зачем это было нужно. Нисходящий подход родился именно из этих соображений — нужно было найти путь такой, чтобы во время всего процесса проектирования находиться на одном уровне абстракции и манипулировать, возможно, меньшим количеством объектов. Подход хорош, но не идеален, так как выделять абстрактные объекты человек по природе не способен, и потому уровни абстракции распределяются в соответствии с логикой программы. Человек по-прежнему абстрагируется от логики программы, но, как правило, вынужден точно определять данные, проходящие через несколько сегментов программы. Поэтому уровни абстракции не получаются, получаются просто уровни подпрограмм. И, если в хорошо определенных математических задачах этот фокус проходит, так как там абстракция идет в основном по логике, то для других задач — нет. В области численных задач был даже такой период, когда намечалось написание универсальных подпрограмм, которые останется только вызывать в определенном порядке.

Однако в целом этот метод не оправдался, так как программирование ушло от вычислительных задач и шагнуло к информационным, а тут уже на первое место вышли не логика, а данные. И если раньше задача была — при бедном наборе типов данных реализовать как можно больше функций над ними, то теперь — при довольно бедном наборе функций охватить как можно больше типов данных.

Все усилия в основном направлены на то, чтобы одну и ту же функцию распространить на разные типы данных, например, посчитать одномерную таблицу не только по дискретным предположениям, но и по непрерывным, и по символьным, причем уметь задавать какие-то структуры внутри этих данных (интервалы, автоматические и ручные) — вот теперь группировка альтернатив и разные классы эквивалентности; при всем при этом функция остается та же — напечатать таблицу распределения частот.

На этом пути создаются всяческие языки, ориентированные на данные. Понятно, что используя достижения предыдущего этапа — от логики человек привычно абстрагируется подпрограммами. Однако и тут не предел развития, потому что на самом деле (без машины) человек (разработчик) работает не в терминах логики и/или данных — он оперирует с объектами, имеющими определенные свойства.

Можно предположить, что современное программирование идет именно к этому. Уже достаточно очевидно, что фактически речь идет об абстрактных типах данных, но предпочтительнее говорить о более широком понятии — объектно-ориентированном подходе к программированию. Это значительно шире, так как, кроме технических средств (языков и т. д.), здесь речь идет и об изменении мышления программиста.

Разработка программы на языке высокого уровня является наглядной иллюстрацией многоуровневой абстракции. Компилятор преобразует программы на ЯВУ в язык ассемблера, который затем переводит команды в двоичный код, понятный для процессора. Необходимо заметить, что некоторые компиляторы генерируют команды уже непосредственно на машинном языке, минуя уровень ассемблера.

Перед выполнением программы на ЯВУ компилятор и ассемблер транслируют ее в команды машинного языка. Эта операция выполняется всего раз, и нет необходимости ее повторять при очередном запуске, если только исходный текст программы не был изменен. Наличие нескольких уровней позволяет скрыть детали нижележащего машинного языка от программиста, что уже было сказано ранее, и обеспечить более производительный и простой интерфейс.

Примерами моделей программного обеспечения, использующих уровни абстракции, являются семиуровневая модель OSI для протоколов передачи данных компьютерных сетей, библиотека графических примитивов OpenGL, модель ввода-вывода на основе потоков байтов из Unix, адаптированная MS DOS, Linux и многие другие современные операционные системы [8].

К примеру, в сетевой модели OSI содержится семь уровней абстракции:

Физический — самый нижний уровень абстракции этой модели, который отвечает за передачу данных, которые представлены в двоичном коде, от устройства к устройству. К нему относятся физические, механические и электрические интерфейсы между двумя системами. Все функции этого уровня реализуются на устройствах, подключенных к сети.

Канальный — уровень, предназначенный для обеспечения взаимодействия сетей на физическом уровне, а также выявления и исправления ошибок, которые могут возникнуть при этом. Полученные с нижнего (физического) уровня данные упаковываются в кадры, проверяются на целостность, исправляются, если это необходимо, и только потом отправляются на уровень выше, на сетевой уровень.

Сетевой — уровень, необходимый для определения пути передачи данных, т. е. определения кратчайшего маршрута передачи данных от источника к получателю, отслеживания ошибок и «заторов» в сети, а также коммутацию и маршрутизацию.

Транспортный — уровень абстракции, необходимый для обеспечения надежной передачи данных. Но при этом необходимо учитывать, что уровень этой надежности может находиться в широких пределах. К примеру, существуют протоколы, осуществляющие только основные транспортные функции, например, без подтверждения приема данных, и протоколы, осуществляющие не только основные, но и дополнительные функции, такие как доставка не-

скольких пакетов данных в определенной последовательности, достоверность принятых данных и управление этими потоками данных.

Сеансовый — уровень, обеспечивающий поддержание сеанса связи и позволяющий приложениям взаимодействовать друг с другом длительное время. Также на этом уровне создается/завершается сеанс, осуществляется синхронизация задач, определяются права на передачу данных и поддержание неактивного сеанса.

Представительский — уровень, обеспечивающий преобразование протоколов и кодирование/декодирование данных. Этот уровень представляется как промежуточный протокол для преобразования информации, полученной из соседних уровней. То есть запросы приложений с прикладного уровня здесь преобразуются в формат для передачи по сети, а данные, полученные из сети, преобразуются в формат приложений. Также этот уровень абстракции форматирует и преобразовывает код, чтобы обеспечить приложению ту информацию, с которой он мог бы работать и которая имела бы смысл, переводит данные из одного формата в другой. Еще одной функцией этого уровня является защита информации от несанкционированного доступа при пересылке. Для этого на этом уровне существуют подпрограммы, сжимающие текст и преобразовывающие графические файлы в битовые потоки для дальнейшей передачи по сети. Соответственно, уровень представления отвечает за организацию и защиту данных при пересылке.

Прикладной — самый верхний уровень абстракции модели, который обеспечивает взаимодействие пользователя с сетью:

– осуществляет удаленный доступ к файлам и базам данных и пересылку электронной почты;

– отвечает за передачу служебной информации;

– предоставляет приложениям информацию об ошибках;

– формирует запросы к представительскому уровню.

Этот уровень содержит набор популярных протоколов, необходимых пользователям, к примеру протокол, передачи гипертекста HTTP.

Типичное представление с точки зрения архитектуры компьютера системы в виде последовательности уровней абстракции:

– компьютерная техника;

– прошивка;

– язык ассемблера;

– ядро операционной системы;

– приложения.

При этом необходимо упомянуть, что любой из протоколов модели OSI может взаимодействовать только с протоколами своего уровня и с протоколами на единицу выше и/или ниже своего уровня. Также любой протокол модели OSI может выполнять только функции своего уровня и никакого другого больше.

Причин, по которым модель OSI не была реализована, не так уж и мало:

– в связи с затянувшейся разработкой протоколов OSI, основным используемым стеком протоколов стал TCP/IP, разработанный еще до принятия моде-

ли OSI, и вне связи с ней ни одна из компаний не стала поддерживать очередной стек протокол, ожидая, что это сделает кто-то другой;

– несовершенство разработанной модели, в том числе и ее протоколов. В результате этого два уровня (сеансовый и уровень представления) оказались пусты, в то время как два других перегружены (сетевой и передачи данных);

– сложность модели и протоколов, а также медлительность первых и последующих реализаций;

– неудачная политика тоже сыграла свою роль, так как разработчики пытались навязать исследователям и программистам неудавшийся в техническом отношении стандарт, что не способствовало продвижению этой модели.

Проанализировав вышесказанное, можно сделать вывод, что использование уровней абстракции в разработке операционных систем имеет особое место, так как декомпозиция системы весьма удобна для обработки и восприятия. В первую очередь, восприятия разработчика. Также необходимо учесть, что число уровней не должно превышать семи, так как на примере модели OSI можно убедиться, насколько усложнится задача при увеличении числа уровней абстракции.

Но все-таки, несмотря на все свои недостатки, модель OSI (кроме сеансового уровня и уровня представления) показала себя исключительно полезной для теоретических дискуссий о компьютерных сетях.

Таким образом мы вновь возвращаемся к утверждению, что уровни абстракции, как было сказано ранее, подталкивают нас к более обширному понятию как ООП (объектно-ориентированное программирование), где необходимо не только объектное мышление, но и верно выбранные уровни абстракции. Сделав это, разработчик уже может свободно представлять вещи в виде объектов и их взаимодействия, будь то реализация протокола, бизнес-процесса или слоя доступа к базе данных. Соответственно, уровни абстракции весьма полезны для правильного представления и последующего решения поставленной задачи.

ЗНАКОМСТВО С ОПЕРАЦИОННОЙ СИСТЕМОЙ UNIX

Администрирование

Администратор и суперпользователь

Суперпользователь

Во всех системах на базе Linux всегда есть один привилегированный пользователь, который называется root, или суперпользователь. Полномочия этого пользователя не ограничены ничем, он может делать в системе абсолютно все, что угодно. Кроме того, большинство системных процессов работают от имени root. Обычный же пользователь в Linux не может устанавливать и удалять программы, управлять системными настройками и изменять файлы вне своего домашнего каталога. Поскольку использование суперпользователя крайне опасно, в UNIX-системах он спрятан внутри системы, а управлением занимаются обычные пользователи со специальными административными привилегиями. В суперпользователя можно превратиться. Для этого необходимо выполнить команду:

```
su # Super User
```

Администратор

Администратор по умолчанию может по запросу делать все то же самое, что и суперпользователь, однако случайно что-то испортить из-под администратора нельзя, так как перед выполнением каждого опасного действия система спрашивает у пользователя-администратора его пароль. Администратор является обычным пользователем, однако при необходимости он может вмешаться в работу системы, но для этого ему потребуется ввести свой пароль. Главное отличие администратора от суперпользователя заключается в необходимости вводить пароль для выполнения любого потенциально опасного действия.

Команда регистрации нового пользователя

Добавление пользователя осуществляется при помощи команды useradd. Пример использования:

```
sudo useradd <имя пользователя>
```

Эта команда создаст в системе нового пользователя. Чтобы изменить настройки создаваемого пользователя, вы можете использовать следующие ключи:

Таблица 1

Ключи для изменения настроек создаваемого пользователя

Ключ	Описание
-b	Базовый каталог. Это каталог, в котором будет создана домашняя папка пользователя. По умолчанию/home
-c	Комментарий. В нем вы можете напечатать любой текст
-d	Название домашнего каталога. По умолчанию название совпадает с именем создаваемого пользователя
-e	Дата, после которой пользователь будет отключен. Задается в формате ГГГГ-ММ-ДД. По умолчанию отключено

Продолжение табл. 1

Ключ	Описание
-f	Количество дней, которые должны пройти после устаревания пароля до блокировки пользователя, если пароль не будет изменен (период неактивности). Если значение равно 0, то запись блокируется сразу после устаревания пароля, при –1 — не блокируется. По умолчанию –1
-g	Первичная группа пользователя. Можно указывать как GID, так и имя группы. Если параметр не задан, будет создана новая группа, название которой совпадает с именем пользователя
-G	Список вторичных групп, в которых будет находиться создаваемый пользователь
-k	Каталог шаблонов. Файлы и папки из этого каталога будут помещены в домашнюю папку пользователя. По умолчанию/etc/skel
-m	Ключ, указывающий, что необходимо создать домашнюю папку. По умолчанию домашняя папка **не создается**
-p	Зашифрованный пароль пользователя. По умолчанию пароль не задается, но пользователь будет заблокирован до установки пароля
-s	Оболочка, используемая пользователем. По умолчанию/bin/sh
-u	Вручную задать UID пользователю

Установка программ

В UNIX-системах, как и в других операционных системах, есть понятие зависимостей. Это значит, что программу можно установить, только если уже установлены пакеты, от которых она зависит. Такая схема позволяет избежать дублирования данных в пакетах (например, если несколько программ зависят от одной и той же библиотеки, то она установится один раз отдельным пакетом). В отличие от, например, Slackware или Windows, в UNIX зависимости разрешаются пакетным менеджером (Synaptic, apt, Центр приложений, apt-get, aptitude) — он автоматически установит зависимости из репозитория. Зависимости придется устанавливать вручную, если нужный репозиторий не подключен, недоступен, если нужного пакета нет в репозитории, если вы устанавливаете пакеты без использования пакетного менеджера (используете Gdebi или dpkg), если вы устанавливаете программу не из пакета (компилируете из исходников, запускаете установочный run/sh скрипт).

Установка из репозиториев

Репозиторий — место централизованного хранения пакетов программного обеспечения. Использование репозиториев позволяет упростить установку программ и обновление системы. Пользователь волен выбирать, какими репозиториями будет пользоваться, и может создать собственный. Список используемых репозиториев содержится в файле /etc/apt/sources.list и в файлах каталога /etc/apt/sources.list.d/, проще всего его посмотреть через специальное приложение, которое можно вызвать через главное меню: *Система → Администрирование → Источники Приложений*, или через Менеджер пакетов Synaptic.

Если не добавлялись локальные репозитории (например, CD/DVD диски), то для установки программ из репозиториев необходим будет Интернет.

У такого метода установки программ есть масса преимуществ: это удобно, устанавливаются уже протестированные программы, которые гарантированно будут работать, зависимости между пакетами будут решаться автоматически, информирование о появлении в репозитории новых версий установленных программ.

С использованием графического интерфейса

Выберите *Система → Администрирование → Менеджер пакетов Synaptic* и получите более функциональный инструмент для работы с пакетами. В частности, можно, например, устанавливать программы частично. Запустите программу Менеджер пакетов Synaptic: *Системные → Менеджер пакетов Synaptic*. По запросу введите свой пароль. В запустившейся программе нажмите кнопку «Обновить», подождите, пока система обновит данные о доступных программах.

В списке доступных программ сделайте двойной клик на нужной программе (либо клик правой кнопкой — пункт «Отметить для установки»). После того, как все нужные программы помечены для установки, нажмите кнопку «Применить». Подождите, пока необходимые пакеты будут скачаны и установлены. Схожие функции выполняет программа «Установка и удаление приложений», ее можно легко найти в меню *Приложения → Установка/удаление*...

С использованием командной строки

Установка из командной строки позволяет получить больше информации о процессе установки и позволяет гибко его настраивать, хотя и может показаться неудобной начинающему пользователю.

Обновить данные о доступных в репозиториях программах можно командой:

```
sudo apt-get update
```

Для установки нужной программы:

```
sudo apt-get install имя-программы
```

Например:

```
sudo apt-get install libsexymm2
```

Если нужно установить несколько программ, то их можно перечислить через пробел, например:

```
sudo apt-get install libsexymm2 nmap
```

Если потребуется — ответьте на задаваемые вопросы (для положительного ответа нужно ввести Y или Д). Программа будет установлена, если она уже установлена — она будет обновлена.

Для поиска программы в списке доступных пакетов:

```
sudo apt-cache search keyword
```

где **keyword** — название программы, часть названия программы или слово из ее описания.

Установка из deb-пакета

Если нужной программы нет в основном репозитории и у автора программы нет своего репозитория, либо если репозитории недоступны (например, нет Интернета), то программу можно установить из deb-пакета (скачанного за-

ранее / принесенного на USB накопителе). Если deb-пакет есть в официальном репозитории, то его можно скачать с сайта http://packages.ubuntu.com. Часто deb-пакет можно скачать с сайта самой программы. Можно также воспользоваться поиском на сайте http://getdeb.net. Минус такого подхода — менеджер обновлений не будет отслеживать появление новых версий установленной программы.

С использованием графического интерфейса

Перейдите в папку, где находится deb-пакет, откройте свойства файла (правая клавиша → Свойства), во вкладке «Права» разрешите выполнение файла (галочка у «Разрешить исполнение файла как программы»). Далее закрываем свойства файла, и по двойному щелчку Nautilus предложит нам открыть код или выполнить файл. Запускаем. Либо возможно это сделать специальным установщиком GDebi. Установить можно из Центра приложений, вписав в поиск GDebi, либо вписав в командную строку:

```
sudo apt-get install GDebi
```

После установки запускаем deb-пакет с помощью установщика программ GDebi; все, что от вас потребуется — это просто нажать кнопку «Установить пакет».

Возможные ошибки

Пакет не может быть установлен. Например, он предназначен для другой архитектуры.

В системе отсутствуют необходимые устанавливаемому приложению пакеты. В таком случае «Установщик программ GDebi» автоматически попытается получить нужные пакеты из репозиториев. Или же вы можете самостоятельно скачать требуемые пакеты и установить их.

С использованием командной строки

Установка выполняется с помощью программы dpkg:

```
sudo dpkg -i /home/user/soft/ntlmaps_0.9.9.0.1-10_all.deb
```

При использовании dpkg нужно ввести полное имя файла (а не только название программы). Можно одной командой установить сразу несколько пакетов, например, следующая команда установит все deb-пакеты в директории:

```
sudo dpkg -i /home/user/soft/ntlmaps_*.deb
```

Это бывает полезно для установки пакета программы вместе с пакетами зависимостей.

Установка программ с собственным инсталлятором из файлов sh, run

Иногда программы могут распространяться с собственным инсталлятором. Это ничем не отличается от ситуации в Windows. Только здесь, распаковав **tar.gz** архив с дистрибутивом программы, вы вместо **setup.exe** увидите что-то наподобие **install.sh**. Это заранее собранный пакет ПО, он берет на себя работу по размещению файлов в нужных местах и прописыванию нужных параметров. При этом пропадает возможность управлять таким ПО с помощью пакетного

менеджера. Пользоваться такими пакетами нежелательно, но если выбора нет, то переходим в директорию с файлом, например:

```
cd ~/soft
```

Разрешение выполнять этот файл:

```
chmod +x install.sh
```

Запуск файла:

```
sudo ./install.sh
```

Иногда программу можно установить и без прав суперпользователя (без sudo), но это, скорее, исключение.

Иногда дистрибутив программы распространяется в виде самораспаковывающегося архива. В таком случае это будет просто один единственный файл .sh, который и нужно запустить. Дальше нужно будет ответить на ряд вопросов, как это делается в Windows. Так устанавливаются официальные драйверы nVidia, ATI, среда разработчика NetBeans и т. п.

Есть программы, которые не нуждаются в инсталляции и распространяются в виде обычного архива tar.gz, который просто достаточно куда-то распаковать. В Windows тоже есть такие программы, их еще часто называют словом Portable. Такие программы обычно можно устанавливать в любой каталог, но обычно стандартное место — это каталог/opt. Конечно, пункты на запуск в меню придется добавлять вручную, для этого нужно щелкнуть правой кнопкой по заголовку меню «Программы» и выбрать «Правка меню».

Установка из исходных кодов

Если для системы нигде нет deb-пакетов, то программу можно скомпилировать самостоятельно из исходных кодов, которые можно скачать на официальном сайте любой Open Source программы либо из source — репозитория дистрибутива.

Основное, что понадобится — это средства для компиляции, поэтому сначала нужно установить пакет build-essential. Далее нужно распаковать архив с кодами программы в какую-то временную папку. Потом нужно найти файл README или INSTALL, прочитать его и выполнить то, что там написано. Чаще установка программ таким способом ограничивается последовательным выполнениям следующих команд:

```
./configure
make
sudo make install
```

Но в некоторых случаях могут быть отличия. Кроме того, после выполнения скрипта ./configure можно получить сообщение о том, что в системе не установлено библиотек, нужных для компиляции программы. В таком случае нужно будет установить их самостоятельно и повторить процесс. Обычно процесс компиляции занимает определенное время и напрямую зависит от мощности вашего компьютера.

Автоматическая установка зависимостей при сборке из исходников

Такой тип установки лучше, чем просто ./configure && make && make install, и подходит для установки программ, отсутствующих в репозиториях.

Установка auto-apt:

```
sudo apt-get install auto-apt
```

Переходим в папку с распакованными исходниками и командуем:

```
sudo auto-apt update && auto-apt -y run ./configure
```

Команда auto-apt сама доставит необходимые пакеты для сборки и уменьшит количество вопросов. Создание deb-пакета для более простой работы в дальнейшем (установка, удаление и прочее):

```
checkinstall -D
```

Архивирование. Копирование файлов на стример

Архиватор tar

tar — наиболее распространенный архиватор, используемый в Linux-системах. Сам по себе tar не является архиватором в привычном понимании этого слова, так как он самостоятельно не использует сжатие. В то же время многие архиваторы (например, Gzip или bzip2) не умеют сжимать несколько файлов, а работают только с одним файлом или входным потоком. Поэтому чаще всего эти программы используются вместе. **tar** создает несжатый архив, в который помещаются выбранные файлы и каталоги, при этом сохраняя некоторые их атрибуты (такие, как права доступа). После этого полученный файл *.tar сжимается архиватором, например, gzip. Вот почему архивы обычно имеют расширение .tar.gz или .tar.bz2 (для архиваторов gzip и bzip2 соответственно).

Использование

tar запускается с обязательным указанием одного из основных действий, самые распространенные из которых — создание и распаковка архивов. Далее задаются прочие параметры, зависящие от конкретной операции.

Создание архива

Для создания архива нужно указать **tar** соответствующее действие, что делается с помощью ключа -c. Кроме того, для упаковки содержимого в файл необходим ключ -f. Далее указываются сначала имя будущего архива, а затем те файлы, которые необходимо упаковать:

```
tar -cf txt.tar *.txt
```

Эта команда упакует все файлы с расширением txt в архив txt.tar. Так и создается простейший архив без сжатия. Для использования сжатия не нужно запускать что-либо еще, достаточно указать **tar**, каким архиватором следует сжать архив. Для двух самых популярных архиваторов gzip и bzip2 ключи будут -z и -j соответственно:

```
tar -cvzf files.tar.gz ~/files
```

Упакует папку ~/files со всем содержимым в сжатый с помощью gzip архив:

```
tar -cvjf files.tar.bz2 ~/files
```

Создаст аналогичный архив, используя для сжатия bzip2. Ключ -v включает вывод списка упакованных файлов в процессе работы. Помимо gzip и bzip2 можно использовать, например, lzma (ключ -lzma) или xz (ключ -J), при этом соответствующий архиватор должен быть *установлен* в системе.

Распаковка архива

Действие «распаковка» задается с помощью ключа -x. И тут снова потребуется ключ -f для указания имени файла архива. Также необходим ключ -v для визуального отображения хода процесса:

```
tar -xvf /path/to/archive.tar.bz2
```

Распакует содержимое архива в текущую папку. Альтернативное место для распаковки можно указать с помощью ключа -C:

```
tar -xvf archive.tar.bz2 -C /path/to/folder
```

Просмотр содержимого архива

Для просмотра содержимого архива используется следующая команда:

```
tar -tf archive.tar.gz
```

Она выведет простой список файлов и каталогов в архиве. Если же добавить ключ -v, будет выведен подробный список с указанием размера, прав доступа и прочих параметров (так же, как по ls -l).

Прочие возможности

tar предоставляет множество полезных возможностей. Например, можно указать файлы и каталоги, которые не будут включены в архив, добавить файлы в существующий архив, взять список объектов для запаковки из текстового файла и многое другое. Во всем многообразии опций, как всегда, поможет разобраться

```
man tar
```

или же

```
tar --help
```

Копирование файлов на стример. Команда CPIO (Copy In/Out)

Команда ***cpio -o*** берет с системного ввода список имен и склеивает эти файлы вместе в один архив, выталкивая его на свой системный вывод.

Сбросить на ленту файлы по списку: ***o*** - (output) — создавать архив.

H odc — записывать в «совместимом формате» (чтобы архив можно было считать на Besta или Sun); c — записывать в «престарелом» совместимом формате:

```
cat spisok | cpio -ovB -H odc > /dev/rmt/ctape1
find katalog -print | cpio -ovc > arhiwnyj-fajl.cpio
```

Команда ***cpio -i*** читает с системного ввода cpio-архив и извлекает из него файлы.

Просмотреть содержание стримера:

```
cpio -itB < /dev/rmt/ctape
```

Извлечь файлы со стримера:

```
cpio -idmvB [«шаблон» ...] < /dev/rmt/ctape
```

-B — размер блока 5120 байт — стримерный формат.

-d — создавать каталоги в случае необходимости.

-v — вывести список имен обработанных файлов.

-m — сохранять прежнее время последней модификации.

-f — брать все файлы, кроме указанного шаблоном.

-u — безусловно заменять существующий файл архивным.

-l — где можно, не копировать, а делать ссылки.

Архивация со сжатием

Архиваторы tar и cpio, в отличие от DOS-архиваторов, не занимаются компрессией. Чтобы получить сжатый архив, нужно воспользоваться специализированной командой **compress** или **gzip**. Команда **compress** читает свой системный ввод, а на свой системный выход подает «прожатые» данные. Команда zcat («сжатый cat») читает с системного входа «прожатый» файл, а на выход подает «разжатые» данные.

Создать сжатый tar-архив**:**

```
tar -cvf - emacs-19.28 | compress > emacs-19.28.tar.Z
```

Прочитать оглавление сжатого tar-архива:

```
zcat < emacs-19.28.tar.Z | tar -tvf -
```

Обратите внимание на ключ минус «-» на том месте, где в tar нужно указывать имя файла с архивом. Он означает «брать данные со стандартного входа» (или выводить архив на стандартный выход).

GNU Zip — достаточно известный упаковщик, имеет степень сжатия более высокую, чем у compress, почти как у arj или pkzip.

Создать сжатый cpio-архив, используя «компрессор» gzip:

```
find . -print | cpio -ovcaB | gzip > arhiw.gz
```

Извлечь файлы из сжатого cpio-архива:

```
gunzip < arhiw.gz | cpio -idmv
```

Другие утилиты архивации

В зависимости от версии UNIX могут существовать и другие программы для бэкапирования и создания архивов:

```
backup/restore
 dump
 fbackup/frestore (HP/UX)
 pax
 ...
```

Менеджер пакетов Synaptic

Менеджер пакетов Synaptic позволяет полностью управлять отдельными пакетами в системе. Основное его отличие от *Центра приложений*, кроме более богатого функционала, в том, что он работает на уровне пакетов, а не приложений (приложение и пакет — это не одно и то же. Каждое приложение состоит из одного или более пакетов).

Найти Synaptic можно в меню *Системные →Менеджер пакетов Synaptic*. Для запуска понадобится ввести свой административный пароль. При первом входе появляется краткая справка (рис. 2).

Интерфейс Synaptic немного напоминает Центр приложений: слева находится колонка с категориями пакетов, под ней — переключатель способа сортировки по категориям, а справа находится собственно список пакетов и под ним описание текущего выбранного пакета (рис. 3).

Также на верхней панели есть строка поиска, а кроме нее кнопки, позволяющие совершать некоторые операции. При нажатии на кнопку «Обновить» будет произведено обновление индексов всех репозиториев, при нажатии на

кнопку «Отметить для обновления» собственно будут отмечены для обновления все пакеты, для которых доступны новые версии, ну а кнопка «Применить» нужна для применения всех внесенных изменений.

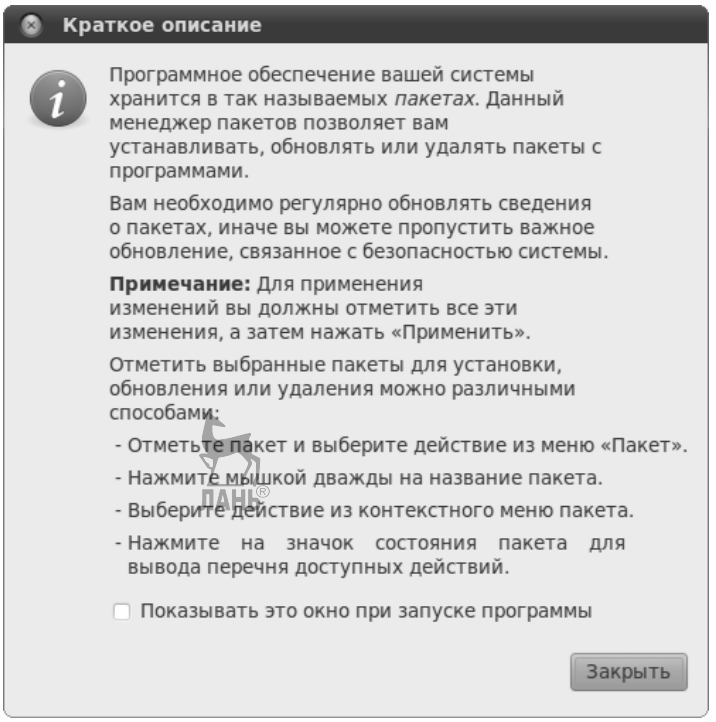


Рис. 2
Справка

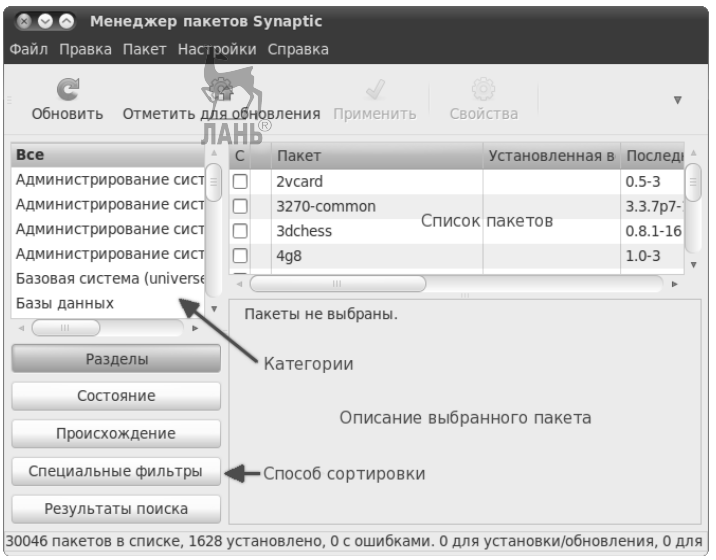


Рис. 3
Интерфейс менеджера пакетов Synaptic

Установленные пакеты помечаются зелеными квадратиками, а неустановленные — белыми. Изменить состояние того или иного пакета можно, нажав правой кнопкой мыши на его название в списке и выбрав нужное действие (рис. 4).

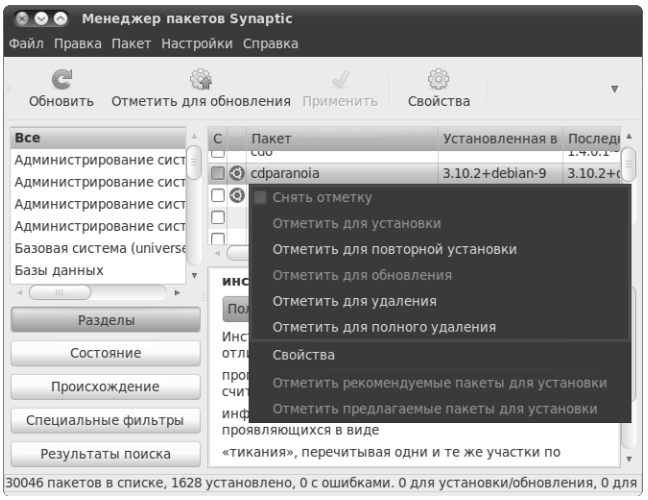


Рис. 4
Менеджер пакетов Synaptic. Изменение состояния пакетов

В отличие от Центра приложений внесенные через Synaptic изменения вступают в силу только после нажатия на кнопку «Применить» на панели инструментов.

Удалить пакет можно одним из двух способов: либо просто удалить файлы пакета, либо удалить их вместе со всеми пользовательскими настройками, относящимися к удаляемому пакету. Отличаются эти способы вот чем: многие программы создают в домашних папках пользователей файлы со своими настройками, а эти программы при простом удалении удаляются без пользовательских настроек, а при полном — с ними.

Synaptic, как и остальные инструменты управления пакетами, автоматически следит за разрешением всех зависимостей и ликвидацией различных конфликтов. Мало того, при совершении любых действий Synaptic выдаст вам окно с подробным описанием вносимых изменений.

Консольные инструменты управления пакетами

Утилита dpkg-

Существуют два основных инструмента работы с пакетами: **aptitude** и **dpkg**. **dpkg** — это низкоуровневая программа управления пакетами, единственная полезная ее функция для обычного пользователя — это прямая установка пакета из deb-файла. Выполняется она командой

```
sudo dpkg -i имя_пакета.deb
```

Для того чтобы команда успешно выполнилась, в системе должны присутствовать все зависимости устанавливаемого пакета, поскольку **dpkg** не уме-

ет их разрешать и скачивать что-либо из репозиториев. Можно также ставить несколько пакетов за раз, передавая их все как аргументы команде **dpkg -i**.

Утилита aptitude

Основным же консольным инструментом работы с пакетами является **aptitude**. В некотором смысле это консольный аналог менеджера пакетов Synaptic, хотя **aptitude** на самом деле обладает куда как большим функционалом.

Установить пакеты из репозиториев можно командой

```
sudo aptitude install имя_пакета1 [имя_пакета2 ...]
```

Сколько бы ни было указано пакетов, **aptitude** автоматически разрешит все зависимости и предложит конечный вариант необходимых действий, останется только лишь согласиться, нажав Enter:

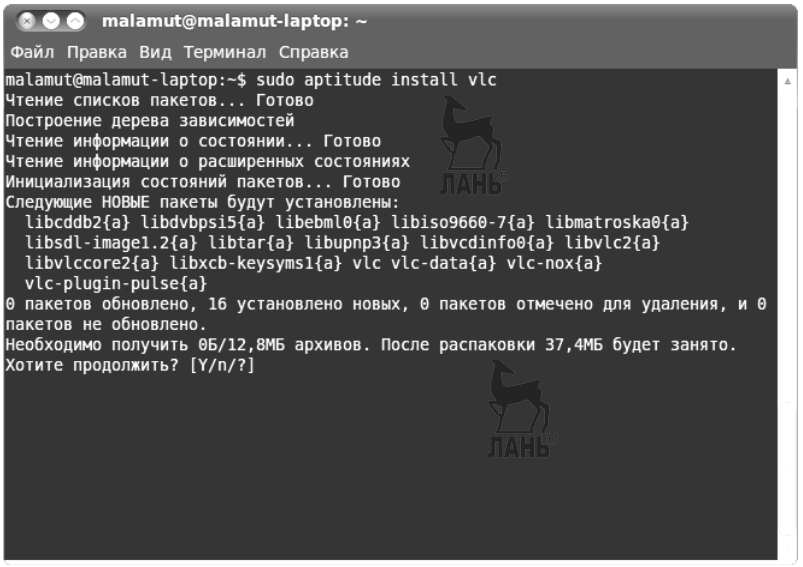


Рис. 5
Установка пакетов из репозиториев

Обратите внимание, **aptitude** предлагает вам в квадратных скобочках три возможных варианта ответа на поставленный вопрос: [Y/n/?]. Y означает *Yes*, то есть согласие, n — это *No*, то есть отказ, а ? — это просьба вывести справку. Вам нужно ввести символ, соответствующий вашему выбору и нажать Enter. Однако часто есть вариант по умолчанию, выделенный в списке большой буквой, и если вам нужен именно он, то вы можете ничего не вводить, просто нажать Enter.

Аналогично установке, удалить пакеты можно одной из двух команд:

```
sudo aptitude remove имя_пакета1 [имя_пакета2 ...]
sudo aptitude purge имя_пакета1 [имя_пакета2 ...]
```

Первая удаляет только файлы пакета, оставляя пользовательские настройки нетронутыми, вторая же удаляет пакет полностью.

Посмотреть описание конкретного пакета можно командой

```
aptitude show имя_пакета
```

Произвести поиск нужного пакета по доступным источникам приложений можно командой

```
aptitude search фраза
```

По умолчанию поиск производится по именам пакетов, для поиска по описаниям надо перед искомой фразой добавить символ ~d:

```
aptitude search ~d фраза
```

aptitude имеет мощный графический интерфейс, попасть в него можно, набрав в терминале просто aptitude. Вот как это выглядит:

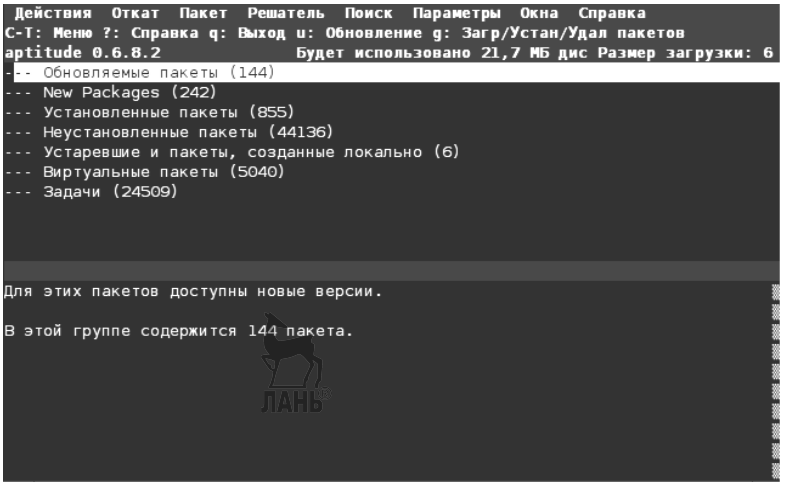


Рис. 6
Интерфейс утилиты aptitude

Несмотря на неприглядный вид, работать с этим очень удобно.

Обычно в разнообразных инструкциях для установки пакетов предлагается как раз использовать команду

```
sudo aptitude install имя_пакета
```

Это ни в коей мере не значит, что обязательно надо исполнять эту команду. Вы спокойно можете поставить указанные пакеты через тот же **Synaptic**. Просто авторы инструкций обычно экономят свое время на объяснении, куда и как надо нажимать в **Synaptic**, давая вместо всего этого одну маленькую команду. Но в конечном итоге **aptitude** и **Synaptic** выполняют одни и те же действия.

Утилита apt-get

В инструкциях гораздо чаще вместо **aptitude** используется **apt-get**. **apt-get** — это стандартная утилита управления пакетами, используется она ровно так же, как и **aptitude**, только у нее нет графического интерфейса и поиска. То есть во всех командах с **install**, **remove**, **purge** можно вместо **aptitude** писать **apt-get**.

Командный интерпретатор SHELL

Вход с системной консоли

Вход в систему осуществляется с системной консоли, которая представляет собой монитор и клавиатуру, связанные непосредственно с системой. Как многопользовательская система UNIX предоставляет возможность работы в нескольких виртуальных символьных терминалах (виртуальных консолях), которые позволяют запускать программы в разных терминалах и от имени разных пользователей работать одновременно под несколькими именами или под одним именем и т. п.

Максимально возможное количество виртуальных терминалов — 12, по умолчанию установленная система представляет 6 виртуальных символьных терминалов и один графический. Переключение между терминалами осуществляется комбинацией клавиш ***<Alt> - <F1>*** — первый терминал, ***<Alt> - <F2>*** — второй терминал и т. д. Переключение из графического терминала в символьный осуществляется сочетанием трех функциональных клавиш ***<Ctrl> - <Alt> - <F#>***, где **#** — номер символьного терминала.

При входе в систему на конкретном терминале пользователь видит приглашение **hostname login**, где **hostname** — имя машины, на которой регистрируется пользователь.

После успешного ввода имени пользователя и пароля система выводит приглашение к вводу команды:

— для суперпользователя **root**;

$ — для всех остальных пользователей.

Система готова к вводу команды, и пользователь может запустить утилиту mc, которая является удобной оболочкой работы с файловой системой.

```
$ mc
```

Часто при первом входе в систему пользователя требуется поменять пароль, назначенный пользователю администратором, — используйте команду **passwd**.

```
$ passwd
```

Выход из терминала осуществляется по команде **exit**

```
$ exit
```

Понятия login и password

Операционная система UNIX является многопользовательской операционной системой. Для обеспечения безопасной работы пользователей и целостности системы доступ к ней должен быть санкционирован. Для каждого пользователя, которому разрешен вход в систему, заводится специальное регистрационное имя — ***username*** или ***login*** — и сохраняется специальный пароль — ***password***, соответствующий этому имени. Как правило, при регистрации нового пользователя начальное значение пароля для него задает системный администратор. После первого входа в систему пользователь должен изменить начальное значение пароля с помощью специальной команды. В дальнейшем он может в любой момент изменить пароль по своему желанию.

Упрощенное понятие об устройстве файловой системы в UNIX. Полные и относительные имена файлов

Понятие «файл» характеризует статическую сторону вычислительной системы. Все файлы, доступные в операционной системе UNIX, как и в уже известных вам операционных системах, объединяются в древовидную логическую структуру. Файлы могут объединяться в каталоги или директории. Не существует файлов, которые не входили бы в состав какой-либо директории. Директории, в свою очередь, могут входить в состав других директорий. Допускается существование пустых директорий, в которые не входит ни один файл, и ни одна другая директория (рис. 7). Среди всех директорий существует только одна директория, которая не входит в состав других директорий, — ее принято называть корневой. На настоящем уровне недостаточного знания UNIX можно заключить, что в файловой системе UNIX присутствует, по крайней мере, два типа файлов: обычные файлы, которые могут содержать тексты программ, исполняемый код, данные и т. д. (их принято называть регулярными файлами), и директории.

Рис. 7
Пример структуры файловой системы

Каждому файлу (регулярному или директории) должно быть присвоено имя. В различных версиях операционной системы UNIX существуют те или иные ограничения на построение имени файла. В стандарте POSIX на интерфейс системных вызовов для операционной системы UNIX содержится лишь три явных ограничения:

- Нельзя создавать имена большей длины, чем это предусмотрено операционной системой (для Linux — 255 символов).
- Нельзя использовать символ NUL (не путать с указателем NULL!) — он же символ с нулевым кодом, он же признак конца строки в языке Си.

• Нельзя использовать символ '/'.

• Единственным исключением является корневая директория, которая всегда имеет имя «/». Эта же директория представляет собой единственный файл, который должен иметь уникальное имя во всей файловой системе. Для всех остальных файлов имена должны быть уникальными только в рамках той директории, в которую они непосредственно входят. Каким же образом отличить два файла с именами «aaa.c», входящими в директории «b» и «d» на рисунке 7, чтобы было понятно, о каком из них идет речь? Здесь на помощь приходит понятие полного имени файла. Мысленно построим путь от корневой вершины дерева файлов к интересующему нас файлу и выпишем все имена файлов (т. е. узлов дерева), встречающиеся на нашем пути, например, «/ usr b aaa.c». В этой последовательности первым будет всегда стоять имя корневой директории, а последним — имя интересующего нас файла. Отделим имена узлов друг от друга в этой записи не пробелами, а символами «/», за исключением имени корневой директории и следующего за ним имени («/usr/b/aaa.c»). Полученная запись однозначно идентифицирует файл во всей логической конструкции файловой системы. Такая запись и получила название полного имени файла.

Понятие текущей директории. Команда pwd. Относительные имена файлов

Для каждой работающей программы в операционной системе, включая командный интерпретатор (shell), который обрабатывает вводимые команды и высвечивает приглашение к их вводу, одна из директорий в логической структуре файловой системы назначается текущей или рабочей для данной программы. Узнать, какая директория является текущей для вашего командного интерпретатора, можно с помощью команды операционной системы *pwd.*

Домашняя директория пользователя и ее определение. Для каждого нового пользователя в системе заводится специальная директория, которая становится текущей сразу после его входа в систему. Эта директория получила название домашней директории пользователя. Воспользуйтесь командой *pwd* для определения своей домашней директории.

Команда man — универсальный справочник

По ходу изучения операционной системы UNIX вам часто будет требоваться информация о том, что делает та или иная команда или системный вызов, какие у них параметры и опции, для чего предназначены некоторые системные файлы, каков их формат и т. д. Большая часть информации в UNIX Manual доступна в интерактивном режиме с помощью утилиты man.

Пользоваться утилитой man достаточно просто: наберите команду

```
$ man имя,
```

где «имя» — это имя интересующей вас команды, утилиты, системного вызова, библиотечной функции или файла. Посмотрите с ее помощью информацию о команде pwd.

Чтобы пролистать страницу полученного описания, если оно не поместилось на экране полностью, следует нажать клавишу <***пробел***>. Для прокрутки одной строки воспользуйтесь клавишей <**Enter**>. Вернуться на страницу назад

позволит одновременное нажатие клавиш <***Ctrl***> *и* <***b***>. Выйти из режима просмотра информации можно с помощью клавиши <***q***>.

Команды cd для смены текущей директории и ls для просмотра состава директории

Для смены текущей директории командного интерпретатора можно воспользоваться командой cd (change directory). Для этого необходимо набрать команду в виде

```
$ cd имя_директории,
```

где «имя_директории» — полное или относительное имя директории, которую вы хотите сделать текущей. Команда cd без параметров сделает текущей директорией вашу домашнюю директорию.

Просмотреть содержимое текущей или любой другой директории можно, воспользовавшись командой ls (от list). Если ввести ее без параметров, эта команда распечатает вам список файлов, находящихся в текущей директории. Если же в качестве параметра задать полное или относительное имя директории:

```
$ ls имя_директории,
```

— она распечатает список файлов в указанной директории. Надо отметить, что в полученный список не войдут файлы, имена которых начинаются с символа «точка» — «.». Такие файлы обычно создаются различными системными программами для своих целей (например, для настройки).

Посмотреть полный список файлов можно, дополнительно указав команде ls опцию -a, т. е. набрав ее в виде

```
$ ls -a или $ ls -a имя_директории
```

У команды ls существует и много других опций.

Команда ls с опциями -al позволяет получить подробную информацию о файлах в некоторой директории, включая имена хозяина, группы хозяев и права доступа, можно с помощью уже известной нам команды ls с опциями -al.

```
$ ls -al
```

В выдаче этой команды третья колонка слева содержит имена пользователей хозяев файлов, а четвертая колонка слева — имена групп хозяев файла. Крайняя левая колонка содержит типы файлов и права доступа к ним. Тип файла определяет первый символ в наборе символов. Если это символ 'd', то тип файла — директория, если там стоит символ '-', то это регулярный файл. Следующие три символа определяют права доступа для хозяина файла, следующие три — для пользователей, входящих в группу хозяев файла, и последние три — для всех остальных пользователей. Наличие символа (r, w или x), соответствующего праву, для некоторой категории пользователей означает, что данная категория пользователей обладает этим правом.

Для получения полной информации о команде ls воспользуйтесь утилитой man.

Команда cat и создание файла. Перенаправление ввода и вывода

Вы уже умеете перемещаться по логической структуре файловой системы и рассматривать ее содержимое. Следует уметь также и просматривать содержимое файлов, и создавать их.

Для просмотра содержимого небольшого текстового файла на экране можно воспользоваться командой cat.

Если набрать ее в виде

```
$ cat имя_файла,
```

то на экран выведется все его содержимое.

Если ваш текстовый файл большой, то вы увидите только его последнюю страницу. Большой текстовый файл удобнее рассматривать с помощью утилиты more (описание ее использования вы найдете в UNIX Manual).

Если в качестве параметров для команды cat задать не одно имя, а имена нескольких файлов:

```
$ cat файл1 файл2 ... файлN,
```

— система выдаст на экран их содержимое в указанном порядке. Вывод команды cat можно перенаправить с экрана терминала в какой-нибудь файл, воспользовавшись символом перенаправления выходного потока данных — знаком «>» («больше»).

Команда

```
$ cat файл1 файл2 ... файлN > файл_результата
```

сольет содержимое всех файлов, чьи имена стоят перед знаком «>», воедино в файл_результата — конкатенирует их (от англ. 'concatenate' — объединять — и произошло название команды).

Прием перенаправления выходных данных со стандартного потока вывода (экрана) в файл является стандартным для всех команд, выполняемых командным интерпретатором. Вы можете получить файл, содержащий список всех файлов текущей директории, если выполните команду ls-a с перенаправлением выходных данных:

```
$ ls -a > новый_файл
```

Если имена входных файлов для команды cat не заданы, то она будет использовать в качестве входных данных информацию, которая вводится с клавиатуры, до тех пор, пока вы не наберете признак окончания ввода — комбинацию клавиш <***CTRL***> *и* <***d***>.

Таким образом, команда

```
$ cat > новый_файл
```

позволяет создать новый текстовый файл с именем «новый_файл» и содержимым, которое пользователь введет с клавиатуры.

У команды cat существует множество различных опций. Посмотреть ее полное описание можно в UNIX Manual.

Заметим, что наряду с перенаправлением выходных данных существует другой способ перенаправить входные данные. Если во время выполнения некоторой команды требуется ввести данные с клавиатуры, можно положить их заранее в файл, а затем перенаправить стандартный ввод этой команды с помощью знака «меньше» — «<» — и следующего за ним имени файла с входными данными.

Шаблоны имен файлов

Шаблоны имен файлов могут применяться в качестве параметра для задания набора имен файлов во многих командах операционной системы. При использовании шаблона просматривается вся совокупность имен файлов, находящихся в файловой системе, и те имена, которые удовлетворяют шаблону, включаются в набор. В общем случае шаблоны могут задаваться с использованием следующих метасимволов:

– * — соответствует всем цепочкам литер, включая пустую;

– ? — соответствует всем одиночным литерам;

– […] — соответствует любой литере, заключенной в скобки.

Пара литер, разделенных знаком минус, задает диапазон литер.

Так, например, шаблону *.c удовлетворяют все файлы текущей директории, чьи имена заканчиваются на .c. Шаблону [a-d]* удовлетворяют все файлы текущей директории, чьи имена начинаются с букв a, b, c, d. Существует одно ограничение на использование метасимвола * в начале имени файла, например, в случае шаблона *c. Для таких шаблонов имена файлов, начинающиеся с символа точка, считаются не соответствующими шаблону.

Простейшие команды работы с файлами — cp, rm, mkdir, mv

Для нормальной работы с файлами необходимо не только уметь создавать файлы, просматривать их содержимое и перемещаться по логическому дереву файловой системы. Нужно уметь создавать собственные поддиректории, копировать и удалять файлы, переименовывать их. Это минимальный набор операций, не владея которым, нельзя чувствовать себя уверенно при работе с компьютером.

Для создания новой поддиректории используется команда mkdir (сокращение от make directory). В простейшем виде команда выглядит следующим образом:

```
$ mkdir имя_директории,
```

— «имя_директории» — полное или относительное имя создаваемой директории. У команды mkdir имеется набор опций, описание которых можно просмотреть с помощью утилиты man.

Для копирования файлов и директорий применяется команда cp. Данная команда может применяться в следующих формах:

```
$ cp файл_источник файл_назначения
```

— служит для копирования одного файла с именем «файл_источник» в файл с именем «файл_назначения».

Команда cp в форме

```
$ cp файл1 файл2 ... файлN дир_назначения
```

— служит для копирования файла или файлов с именами «файл1», «файл2», ... «файлN» в уже существующую директорию с именем «дир_назначения» под своими именами. Вместо имен копируемых файлов могут использоваться их шаблоны.

```
$ cp -r дир_источник дир_назначения
```

— служит для рекурсивного копирования одной директории с именем «дир_источник» в новую директорию с именем «дир_назначения». Если директория «дир_назначения» уже существует, то мы получаем команду cp в следующей форме:

```
$ cp -r дир1 дир2 ... дирN дир_назначения
```

— служит для рекурсивного копирования директории или директорий с именами «дир1», «дир2», ... «дирN» в уже существующую директорию с именем «дир_назначения» под своими собственными именами. Вместо имен копируемых директорий могут использоваться их шаблоны.

Для удаления файлов или директорий применяется команда rm (сокращение от remove). Если вы хотите удалить один или несколько регулярных файлов, то простейший вид команды rm будет выглядеть следующим образом:

```
$ rm файл1 файл2 ... файлN
```

— «файл1», «файл2», ... «файлN» — полные или относительные имена регулярных файлов, которые требуется удалить. Вместо имен файлов могут использоваться их шаблоны. Если вы хотите удалить одну или несколько директорий вместе с их содержимым (рекурсивное удаление), то к команде добавляется опция -r:

```
$ rm -r дир1 дир2 ... дирN
```

— «дир1», «дир2», ... «дирN» — полные или относительные имена директорий, которые нужно удалить. Вместо непосредственно имен директорий также могут использоваться их шаблоны.

У команды rm есть еще набор полезных опций, которые описаны в UNIXManual и могут быть просмотрены с помощью команды man. Командой удаления файлов и директорий следует пользоваться с осторожностью. Удаленную информацию восстановить невозможно. Если вы системный администратор и ваша текущая директория — это корневая директория, пожалуйста, не выполняйте команду rm -r *!

Для перемещения файлов и директорий используется команда mv (сокращение от move). Данная команда может применяться в следующих формах:

```
$ mv имя_источника имя_назначения
```

— для переименования или перемещения одного файла (неважно, регулярного или директории) с именем «имя_источника» в файл с именем «имя_назначения». При этом перед выполнением команды файла с именем «имя_назначения» существовать не должно.

```
$ mv имя1 имя2 ... имяN дир_назначения
```

— служит для перемещения файла или файлов (неважно, регулярных файлов или директорий) с именами «имя1», «имя2», ... «имяN» в уже существующую директорию с именем «дир_назначения» под собственными именами. Вместо имен перемещаемых файлов могут использоваться их шаблоны.

Пользователь и группа. Команды chown и chgrp. Права доступа к файлу

Как уже говорилось, для входа в операционную систему UNIX каждый пользователь должен быть зарегистрирован в ней под определенным именем.

Вычислительные системы не умеют оперировать именами, поэтому каждому имени пользователя в системе соответствует некоторое числовое значение — его идентификатор UID (user identificator).

Все пользователи в системе делятся на группы. Например, студенты одной учебной группы могут составлять отдельную группу пользователей. Группы пользователей также получают свои имена и соответствующие идентификационные номера — GID (group identificator). В одних версиях UNIX каждый пользователь может входить только в одну группу, в других — в несколько групп.

Команда chown предназначена для изменения собственника (хозяина) файлов. Нового собственника файла могут назначить только предыдущий собственник файла или системный администратор.

```
$ chown owner файл1 файл2 ... файлN
```

*— параметр owner задает нового собственника файла в символьном виде, как его username, или в числовом виде, как его UID. Параметры «файл1», «файл2», ... «файлN» — это имена файлов, для которых производится изменение соб*ственника. Вместо имен могут использоваться их шаблоны.

Для каждого файла, созданного в файловой системе, запоминаются имена его хозяина и группы хозяев. Заметим, что группа хозяев не обязательно должна быть группой, в которую входит хозяин. Упрощенно можно считать, что в операционной системе Linux при создании файла его хозяином становится пользователь, создавший файл, а его группой хозяев — группа, к которой этот пользователь принадлежит. Впоследствии хозяин файла или системный администратор могут передать его в собственность другому пользователю или изменить его группу хозяев с помощью команд chown и chgrp.

Команда chgrp предназначена для изменения группы собственников (хозяев) файлов.

```
$ chgrp group файл1 файл2 ... файлN
```

— новую группу собственников файла могут назначить только собственник файла или системный администратор. Параметр group задает новую группу собственников файла в символьном виде, как имя группы, или в числовом виде, как ее GID. Параметры «файл1», «файл2», ... «файлN» — это имена файлов, для которых производится изменение группы собственников. Вместо имен могут использоваться их шаблоны.

Для каждого файла выделяется три категории пользователей:

– пользователь, являющийся хозяином файла;

– пользователи, относящиеся к группе хозяев файла;

– все остальные пользователи.

Для каждой из этих категорий хозяин файла может определить различные права доступа к файлу.

Различают три вида прав доступа: право на чтение файла — r (от слова read), право на модификацию файла — w (от слова write) и право на исполнение файла — x (от слова execute).

Команда chmod предназначена для изменения прав доступа к одному или нескольким файлам.

`$ chmod [who] { + | - | = } [perm] файл1 файл2 ... файлN`
— права доступа к файлу могут менять только собственник (хозяин) файла или системный администратор.

Параметр ***who*** определяет, для каких категорий пользователей устанавливаются права доступа. Он может представлять собой один или несколько символов:

a — установка прав доступа для всех категорий пользователей. Если параметр *who* не задан, то по умолчанию применяется *a*. При определении прав доступа с этим значением заданные права устанавливаются с учетом значения маски создания файлов;

u — установка прав доступа для собственника файла;

g — установка прав доступа для пользователей, входящих в группу собственников файла;

O — установка прав доступа для всех остальных пользователей.

Операция, выполняемая над правами доступа для заданной категории пользователей, определяется одним из следующих символов:

+ — добавление прав доступа;

- — отмена прав доступа;

= — замена прав доступа, т. е. отмена всех существовавших и добавление перечисленных.

Если параметр ***perm*** не определен, то все существовавшие права доступа отменяются.

Параметр ***perm*** определяет права доступа, которые будут добавлены, отменены или установлены взамен соответствующей командой. Он представляет собой комбинацию следующих символов или один из них:

r — право на чтение;

w — право на модификацию;

x — право на исполнение.

Параметры ***файл1, файл2, ... файлN*** — это имена файлов, для которых производится изменение прав доступа. Вместо имен могут использоваться их шаблоны.

Хозяин файла может изменять права доступа к нему, пользуясь командой chmod.

Вход удаленным пользователем

Для входа удаленным пользователем в систему UNIX используется утилита ssh (security shell). Для доступа к другим UNIX-системам с UNIX машины:

```
$ ssh -l <Имя пользователя> <IP адрес удаленной машины>
```

Пользователь может набрать команду

```
$ ssh -l <Имя пользователя> localhost — для доступа по ssh к «своей» (локальной) машине.
```

Команды write и wall

Очень часто устанавливают многопользовательскую поддержку, многие люди работают на том же сервере через различные удаленные доступные рабочие варианты. Однако все эти пользователи системы Linux могут работать на соответствующем проекте или в команде, и даже если они не связаны, они при-

надлежат к одному рабочему месту. Таким образом, вполне вероятно, им придется общаться в любой момент времени. С утилитой Linux **write** Linux пользователи имеют удобный способ общения друг с другом. Можно послать сообщение любому другому пользователю и разрешить войти в Linux машину. Более того, можно посылать сообщения любому пользователю в той же сети, даже на другой машине хозяина.

Синтаксис выглядит так:

```
$ write person
```

Здесь **person** — имя пользователя, если он находится на той же машине, машины, или username@hostname, в случае, если пользователь принадлежит к другой машине хозяина и необходима идентификация, когда один пользователь заходит более чем один раз. Рассмотрим, как передаются сообщения среди пользователей на той же машине.

```
$ w -s
```

— данная команда выводит всех активных пользователей системы;

```
$write person
```

— вместо **person** мы можем вписать имя любого активного пользователя. После нажатия клавиши ввода появляется возможность писать наше сообщение.

Команда **wall** используется для передачи сообщения всем пользователям системы. Однако для получения этого сообщения пользователи должны установить разрешение их mesg на «да». В использовании довольно проста, что легко понять на следующем примере:

```
$ wall
```

— как только ввод сообщения завершен, нужно нажать комбинацию клавиш <**CTRL> + <D>**.

После этого все пользователи получают сообщение.

Таблица 2

Основные информационные команды

Команды	Описание
hostname	Вывести или изменить сетевое имя машины
whoami	Вывести имя, под которым я зарегистрирован
date	Вывести или изменить дату и время
time	Получить информацию о времени, нужном для выполнения процесса
who	Определить, кто из пользователей работает на машине
rwho -a	Определение всех пользователей, подключившихся к вашей сети. Для выполнения этой команды требуется, чтобы был запущен процесс rwho. Если такого нет — запустите «setup» под суперпользователем
finger [имя_пользователя]	Системная информация о зарегистрированном пользователе. Попробуйте: finger root
ps -a	Список текущих процессов
df -h	(=место на диске). Вывести информацию о свободном и используемом месте на дисках (в читабельном виде)
Arch или uname -m uname -r	Отобразить архитектуру компьютера. Отобразить используемую версию ядра

Продолжение табл. 2

Команды	Описание
find / -name file1 find / -user user1	Найти файлы и директории с именем file1. Поиск начать с корня (/). Найти файл и директорию, принадлежащие пользователю user1. Поиск начать с корня (/)
top	Отобразить запущенные процессы, используемые ими ресурсы и другую полезную информацию (с автоматическим обновлением данных)
Kill -9 98989 или kill -KILL 98989	«Убить» процесс с PID 98989 «на смерть» (без соблюдения целостности данных)

Командный интерпретатор Shell

Командный интерпретатор в среде UNIX выполняет две основные функции:

– представляет интерактивный интерфейс с пользователем, т. е. выдает приглашение и обрабатывает вводимые пользователем команды;

– обрабатывает и исполняет текстовые файлы, содержащие команды интерпретатора (командные файлы).

В последнем случае операционная система позволяет рассматривать командные файлы как разновидность исполняемых файлов. Соответственно, различают два режима работы интерпретатора: интерактивный и командный.

Существует несколько типов оболочек в мире UNIX. Две главные — это «Bourne shell» и «C shell». Bourne shell (или просто shell) использует командный синтаксис, похожий на первоначальный для UNIX. В большинстве UNIX-систем Bourne shell имеет имя /bin/sh (где sh сокращение от «shell»). C shell используется иной синтаксис, чем-то напоминающий синтаксис языка программирования Си. В большинстве UNIX-систем он имеет имя /bin/csh.

В Linux есть несколько вариаций этих оболочек. Две наиболее часто используемые — это Новый Bourne shell (Bourne Again Shell) или «Bash» (/bin/bash) и Tcsh (/bin/tcsh). Bash — это развитие прежнего shell с добавлением многих полезных возможностей, частично содержащихся в C shell.

Поскольку Bash можно рассматривать как надмножество синтаксиса прежнего shell, любая программа, написанная на sh shell, должна работать и в Bash. Tcsh является расширенной версией C shell.

При входе в систему пользователю загружается командный интерпретатор по умолчанию. Информация о том, какой интерпретатор использовать для конкретного пользователя, находится в файле /etc/passwd.

Настройка Shell. Файлы инициализации, используемые в bash: /etc/profile (устанавливается системным администратором, выполняется всеми экземплярами начальных пользовательских bash, вызванными при входе пользователей в систему), $HOME/.bash_profile (выполняется при входе пользователя) и $HOME/.bashrc (выполняемый всеми прочими не начальными экземплярами bash). Если .bash_profile отсутствует, вместо него используется .profile. Переменная HOME указывает на домашний каталог пользователя.

tcsh использует следующие сценарии инициализации: /etc/csh.login (выполняется всеми пользовательскими tcsh в момент входа в систему), $HOME/.tcshrc (выполняется во время входа в систему и всеми новыми экземплярами tcsh) и $HOME/.login (выполняется во время входа после .tcshrc). Если .tcshrc отсутствует, вместо него используется .cshrc.

Командные файлы. Командный файл в UNIX представляет собой обычный текстовый файл, содержащий набор команд UNIX и команд Shell.

Для того чтобы командный интерпретатор воспринимал этот текстовый файл как командный, необходимо установить атрибут на исполнение.

Установку атрибута на исполнение можно осуществить командой chmod или через mc по клавише F9, выйти в меню и выбрать вкладку File, далее выбрать изменение атрибутов файла.

Например:

```
$ echo « ps -af » > commandfile
$ chmod +x commandfile
$ ./commandfile
```

В представленном примере команда echo « ps –af » > commandfile создаст файл с одной строкой « ps -af », команда chmod +x commandfile установит атрибут на исполнение для этого файла, команда ./commandfile осуществит запуск этого файла.

Переменные shell. Имя shell-переменной — это начинающаяся с буквы последовательность букв, цифр и подчеркиваний. Значение shell-переменной — строка символов.

Например: Var = «String» или Var = String.

Команда ***echo $Var*** выведет на экран содержимое переменной Var, т. е. строку ‘String’, на то, что мы выводим содержимое переменной, указывает символ ‘$’.

Так, команда echo Var выведет на экран просто строку ‘Var’.

Еще один вариант присвоения значения переменной ***Var = ` набор команд UNIX `***

Обратные кавычки говорят о том, что сначала должна быть выполнена заключенная в них команда, а результат ее выполнения, вместо выдачи на стандартный выход, приписывается в качестве значения переменной.

CurrentDate = `date`

– Переменной CurrentDate будет присвоен результат выполнения команды date.

Можно присвоить значение переменной и с помощью команды «read», которая обеспечивает прием значения переменной с (клавиатуры) дисплея в диалоговом режиме.

Например:

```
echo «Введите число»
read X1
echo «вы ввели -» $X1
```

Несмотря на то что shell-переменные в общем случае воспринимаются как строки, т. е. «35» — это не число, а строка из двух символов «3» и «5», в ряде случаев они могут интерпретироваться иначе, например, как целые числа.

Разнообразные возможности имеет команда «expr».

Например, командный файл:

```
x=7
y=2
rez=`expr $x + $y`
echo результат=$rez
```

выдаст на экран результат = 9.

ОБЗОР МЕТОДОВ И СТРАТЕГИЙ ДИСПЕТЧЕРИЗАЦИИ ПРОЦЕССОРА В ОС

Основным понятием, которое связано с операционными системами, является процесс — теоретическое понятие, которое описывает работу программы. Все остальное основывается на этом суждении.

Все современные компьютеры могут делать несколько вещей одновременно. К примеру, в то время как работает запущенная пользователем программа, имеет возможность производиться копирование файлов с диска на флеш-карту. В многозадачной системе процессор переключается между программами, давая каждой от десятков до сотен миллисекунд. В то же время в любой данный момент времени процессор занимается только одной программой, но за секунду он может успеть поработать с несколькими программами, что создает у пользователей иллюзию параллельной работы со всеми программами.

Следить за работой параллельных процессов достаточно непросто, поэтому создатели операционных систем разработали концептуальную модель последовательных процессов, которая упрощает данную работу.

В данной модели все функционирующее на компьютере программное обеспечение скооперировано в виде набора последовательных процессов. Процессом считается выполняемая программа, охватывая текущие значения счетчика команд, регистров и переменных. Процессор же в свою очередь переключается от программы к программе. Это переключение и называется многозадачностью или же мультипрограммированием. Один процессор может переходить от одного процесса к другому, применяя некоторый алгоритм планирования, чтобы определить момент переключения от одного процесса к другому.

Часть операционной системы, которая отвечает за это, называется планировщиком.

Планирование и диспетчеризация процессора — это одна из самых важных функций операционной системы.

Диспетчеризация процессора — это рассредоточение его времени между процессами в системе. Ведущей целью диспетчеризации считается предельная загрузка процессора, которая достигается как раз с помощью мультипрограммирования.

Главным вопросом планирования считается выбор момента принятия решений. Существует большое количество ситуаций, в которых требуется планирование.

Во-первых, когда формируется новый процесс, нужно решить, какой процесс запустить: родительский или же дочерний. Так как оба процесса находятся в состоянии готовности, данная ситуация не выходит за рамки обычного и планировщик имеет возможность запустить всякий из этих двух процессов.

Во-вторых, планирование требуется при завершении работы процесса. Данный процесс уже не существует, значит, нужно из списка готовых процессов выбрать и запустить следующий. В случае если процессов, которые нахо-

дятся в состоянии готовности, нет, как правило, запускается холостой процесс, поставляемый системой.

В-третьих, при блокировании процесса на операции ввода-вывода, семафоре, или же по какой-то иной причине, требуется выбрать и запустить другой процесс. Временами причина блокировки влияет на выбор. Сложность заключается в том, что планировщик, как правило, не владеет информацией, которая необходима для принятия правильного решения.

В-четвертых, необходимость планирования появляется при возникновении прерывания ввода-вывода. В случае если прерывание пришло от устройства ввода-вывода, который завершил работу, можно запустить процесс, блокированный в ожидании этого события. Планировщик обязан решить, какой процесс запустить: новый, тот, который был временно остановлен прерыванием, или же какой-то другой.

Чтобы создать метод планирования, нужно владеть информацией о том, что должен делать хороший алгоритм. Поэтому следует учитывать некоторые критерии диспетчеризации. К основным критериям относятся:

– использование процессора — это поддержание его в режиме занятости максимально возможный период времени. Критерий оптимизации: максимизация данного показателя;

– пропускная способность системы — это (среднее) количество процессов, завершающих свое выполнение за единицу времени. Критерий оптимизации: максимизация;

– среднее время обработки одного процесса — время, требуемое для исполнения какого-либо процесса. Критерий оптимизации: минимизация;

– среднее время ожидания одним процессом — время, которое процесс ждет в очереди процессов, готовых к выполнению. Критерий оптимизации: минимизация;

– среднее время ответа системы — время, необходимое от этапа первого запроса до первого ответа (данный показатель наиболее важен для среды разделения времени). Критерий оптимизации: минимизация.

Как и при любой оптимизации, независимо от стратегии, в одно и то же время удовлетворить всем критериям нельзя. Существует множество стратегий диспетчеризации, имеющие как достоинства, так и недостатки, с точки зрения достижения оптимальности указанных критериев.

1. Стратегия First — Come — First — Served

(FCFS — «первым пришел — первым обслужен») — стратегия диспетчеризации, которая, пожалуй, считается самой простой в реализации. Процессы получают доступ к процессору в том порядке, в котором они поступили в систему, независимо от потребляемых ими ресурсов. Чаще всего образуется единая очередь ожидающих процессов. Как только появляется первая задача, она незамедлительно запускается и работает столько, сколько нужно. Другие задачи ставятся в конец очереди. Когда нынешний процесс блокируется, запускается следующий в очереди, а когда блокировка снимается, процесс переходит в конец очереди.

Главным плюсом данной стратегии является то, что ее просто понять и также просто программировать. В рамках данной стратегии все процессы, готовые к исполнению, контролируются одним связным списком. Для того чтобы выбрать процесс для запуска, нужно просто выбрать первый элемент из этого списка и удалить его.

Однако существует и недостаток, а именно то, что, когда увеличивается загрузка вычислительной системы, увеличивается и среднее время ожидания обслуживания, при этом короткие задания, которые требуют маленьких затрат машинного времени, вынуждены ожидать столько же, сколько трудозатратные задания.

2. Стратегия Shortest Job First

(SJF — «кратчайшая задача — первая») — стратегия диспетчеризации процессора, в которой процессор в первую очередь предоставляется самому короткому процессу из доступных в системе. Данная стратегия требует, чтобы для каждого задания была известна оценка в потребностях машинного времени. Необходимость информирования ОС о характеристиках задач с описанием потребностей в ресурсах вычислительной системы привела к тому, что были разработаны и соответствующие языковые средства.

Стратегия диспетчеризации SJN подразумевает, что существует единственная очередь готовых к выполнению заданий.

3. Стратегия SRT

(Shortest Remaining Time — наименьшее оставшееся время выполнения) — эта стратегия «кратчайшая задача — первая» только с переключениями. В соответствии с этой стратегией диспетчеризации планировщик каждый раз делает выбор в пользу процесса с наименьшим количеством оставшегося времени выполнения. В данном случае также требуется заранее знать время выполнения задач. При поступлении новой задачи, сравнивается ее полное время выполнения с оставшимся временем выполнения нынешней задачи. В случае если время выполнения новой задачи меньше, нынешний процесс временно останавливается и управление переходит к новой задаче. Данная схема дает возможность быстро обслуживать короткие запросы.

4. Стратегия Round Robin

(RR — круговая система). В этой стратегии предполагается, что каждому процессу предоставляется процессорное время порциями или, по-другому, квантами времени. В случае если к окончанию кванта времени процесс все еще выполняется, он прерывается, а управление переходит к следующему процессу. Планировщику в этом случае требуется всего лишь поддерживать список процессов в состоянии готовности.

Очень важно в этой стратегии выбрать правильную длину кванта. В случае если выбранное значение кванта больше среднего интервала работы процессора, переключение процессов станет происходить редко. Напротив, основная масса процессов будут исполнять блокирующую операцию прежде, чем истечет продолжительность кванта, что будет вызывать переключение процессов. Предотвращение принудительных переключений процессов улучшает продуктивность системы, поскольку переключения процессов станут происходить

только в том случае, когда это действительно необходимо, то есть когда процесс заблокировался и не имеет возможности продолжать работу.

Итак, очень малый квант приведет к частому переключению процессов и маленькой эффективности, но очень большой квант приведет к медленному реагированию на короткие интерактивные запросы. Обычно разумным решением является значение кванта около 20–50 мс.

Стратегия RR подразумевает, что все процессы имеют одинаковый приоритет, то есть равнозначны.

5. Стратегия приоритетного планирования

В ситуации компьютера с большим числом пользователей не все процессы могут быть равнозначны, а это значит, что предыдущая стратегия RR может не подойти. Для разрешения этой проблемы существует приоритетное планирование. Ключевая идея этой стратегии заключается в том, что каждому процессу присваивается свой приоритет, и управление передается процессу в состоянии готовности с самым высоким приоритетом.

Приоритеты могут быть статическими или динамическими. Статические приоритеты не изменяются, а значит, не откликаются на изменения окружающей среды. Поэтому подобный механизм установки приоритетов считается недостаточно гибким.

Динамические приоритеты, напротив, реагируют на изменение ситуации, что дает возможность увеличить реактивность системы. Система имеет возможность изменить начальное значение приоритета на новое, более подходящее для данной ситуации.

Таким образом, задача диспетчеризации процессора в операционной системе является одной из самых важных.

Многопоточность

Многопоточность — это свойство платформы (допустим, виртуальной машины, ОС или приложения), заключающееся в том, что процессы, созданные в системе, могут состоять из нескольких потоков, что выполняются «параллельно», а значит, что они выполняются без предписанного порядка по времени. Когда выполняются некоторые задачи, подобное разделение может помочь достичь наиболее эффективного использования ресурсов нашей машины.

Данные потоки называют потоками выполнения (с *англ*. thread of execution); иногда их называют «нитями» (с *англ*. thread) или неформально «тредами».

У многопоточности в программировании есть ряд достоинств, а точнее:

– иногда программу можно упростить за счет использования общего пространства;

– за счет распараллеливания вычислений процессором и операций ввода-вывода значительно повышается производительность;

– временные затраты на создание потока меньшие, относительно процесса.

Использование многопоточности в наше время крайне необходимо, так как благодаря ей приложения способны выполнять множество функций одновременно. Например, если мы дали запрос серверу, который будет обрабаты-

ваться продолжительное время, чтобы интерфейс пользователя не заблокировался, ожидая ответа от сервера, используют многопоточность, где один поток дает запрос серверу, а другой ожидает действий от пользователя.

Существуют два вида многозадачности:

1) основывается на процессах;

2) основывается на потоках.

Процесс включает в себя управление ресурсами, виртуальную память, дескрипторы Windows. Процесс обязан содержать хотя бы один поток. Это означает, что в одной программе одновременно могут решаться одна и более задач. Например, при заполнении текста в редакторе он одновременно может форматироваться, если действия происходят в разных потоках.

Многозадачность, которая основывается на процессах, отличается от многозадачности, которая основывается на потоках, тем, что многозадачность на процессах организуется для параллельного выполнения программы, а многозадачность на потоках предназначен для параллельного выполнения отдельных частей одной программы.

Многопоточность является очень сложной темой в программировании, при ее использовании возникает большое количество проблем. Не понимая механизма сложно предсказать, какой результат будет при завершении программы, использующей более одного потока.

Основы многопоточной обработки

Существует два вида многозадачности: первая — на основе процессов, а также вторая — на основе потоков. Процесс отвечает за управление ресурсами машины, к которым относятся дескрипторы Windows и виртуальная память, и содержит минимум один поток. Наличие минимум одного потока является обязательным, при работе любой программы. Значит, что многозадачность на основе процессов — средство, благодаря которому на компьютере существует возможность выполнять две или более программ сразу.

Многозадачность на основе процессов позволяет одновременно выполнять программы, например, работу в электронных таблицах, просмотр содержимого в Интернете, текстового редактора и так далее. При организации многозадачности на основе процессов программа является наименьшей единицей кода, выполнение которой планировщик задач может координировать.

Отличия в многозадачности на основе процессов и потоков могут быть сведены к следующему: многозадачность на основе процессов организуется для параллельного выполнения программ, а многозадачность на основе потоков — для параллельного выполнения отдельных частей одной программы.

Цели многопоточности

Сложность приложения, при использовании нескольких потоков, конечно, значительно увеличивается, так как возникают малозаметные эффекты, которые могут привести к странным ошибкам. Прежде чем разбивать программу на несколько потоков, следует продумать, стоит ли вообще вводить такие дополнительные затраты труда и усложнения кода.

Три основные причины использования нескольких потоков в программе это:

1. Чтобы приложение клиента реагировало незамедлительно на действие пользователя. Ведь если приложение будет выполнять длительную задачу, значит оно будет забирать все ресурсы процессора, а приложение, в свою очередь, перестанет реагировать на действия пользователя. Из чего следует, что у пользователя не будет другого выхода, кроме как ожидать окончания вспомогательной задачи. Если рассматривать это со стороны пользователя, то такое поведение программы неприемлемо. Решение этой проблемы кроется в направлении одной задачи в другой поток, а в результате этого интерфейс, который обслуживается первым потоком, будет без задержки реагировать на действия пользователя. Также нелишним будет предоставить пользователю отменить вспомогательную задачу в любой момент, когда он захочет, пока она еще не завершена.

2. Многопоточность также можно использовать для решения множества задач одновременно. Конечно, если мы будем использовать однопроцессорный компьютер, то многозадачность не сможет повысить производительность. Если смотреть правде в глаза, то она даже немного уменьшится, так как появятся различные расходы на создание, поддержание работы потока. Но существуют и такие случаи, когда работе определенной программы присущи интервалы бездействия процессора. Допустим, когда происходит загрузка данных из внешнего источника или при коммуникации с дистанционным компонентом процессор не работает все время. Значит, когда выполняются задачи такого рода, процессор большую часть времени свободен. Уменьшить время ожидания не получится, так как это зависит от канала передачи информации, а вот занять процессор другими задачами вполне возможно.

3. Серверная часть приложения одновременно может обслуживать произвольное количество клиентов. Эта одновременность обеспечивается серверной технологией (например ASP.NET). У программиста есть возможность создать собственную структуру серверной части.

Виды реализации потоков:

– Поток находится в пространстве пользователя. Каждый процесс имеет таблицу потоков, что аналогична таблице процессов ядра.

– Поток находится в пространстве пользователя. Так же как и с таблицей процессов в пространстве ядра имеется и таблица потоков.

– «Волокна» (с *англ*. fibers). В этом виде несколько потоков режима пользователя исполняются в одном потоке режима ядра. Поток пространства ядра использует очень много ресурсов, и в первую очередь это физическая память и диапазон адресов режима ядра для стека режима ядра. Для этого было введено понятие «волокна» — то есть облеченный поток, который выполняется исключительно в режиме пользователя. У каждого из потоков может быть по несколько «волокон».

Реализация многопоточности в .NET

Далее рассмотрим, что собой представляет программная платформа .NET Framework и как в ней реализуется многопоточность.

.NET Framework — *программная платформа*, которая была выпущена компанией Microsoft в 2002 г. Основой этой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), что подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих данную среду.

Особенность многопоточной программы состоит в том, что она может состоять из нескольких частей, каждая из которых выполняет какую-то свою часть задачи. Таким образом части выполняются параллельно. Каждая из таких частей называется потоком. Среда .NET Framework имеет ряд классов, которые предназначены для реализации многопоточных приложений. При этом, в отличии от других языков программирования, в силу встроенной многопоточности, позволяет свести к минимум или полностью ликвидировать пробелы, которые встречались в других языках.

Многопоточность, как правило, ориентирована на потоки и процессы. Процесс, это отдельно выполняемая программа. А многопоточность основана на том, что выполняются сразу две или более программы.

Поток (с *англ*. thread — нить) — это управляемая единица кода, которая выполняется в адресном пространстве породившего его потока. При использовании многопоточности существует возможность реализовать программу так, чтобы один поток просчитывал графику в сцене, визуализировал ее и обновлял окно, а другой в это же время просчитывал физические законы, что происходят в сцене. Или, допустим, программа должна заниматься просчетом математических алгоритмов. Вычисление одного уравнения занимает секунд 10. А значит, так как программа выполняет строки кода последовательно, до тех пор, пока вычисление не будет завершено, окно приложения не будет отвечать на запросы операционной системы, т. е. оно зависнет на секунд десять, пока не окончатся циклы математических вычислений.

Из всего этого следует, что ни одна современная полноценная программа не может обойтись без многопоточности.

Потоки могут как выполняться, так и ожидать выполнения, могут быть временно приостановлены или даже заблокированы. Также потом поток может просто завершиться. Все это — возможные состояния потока.

Многопоточность в среде .NET Framework осуществляется таким образом: существуют два типа потоков — высоко- и низкоприоритетный.

Высокоприоритетный (с *англ*. foreground) поток, в отличие от низкоприоритетного (или фонового background), назначается как тип потока по умолчанию, а также не будет остановлен в случае, если все высокоприоритетные потоки в его процессе будут остановлены.

Умение писать многопоточные программы сводится к тому, что нужно уметь эффективно разработать объектную модель программы, что будет использовать в ходе решения задачи несколько отдельных потоков, а также коор-

динировать работу их между собой. Такая координация работы называется синхронизацией. По сути, синхронизация — специальное средство, которое оснащено собственной подсистемой функций и являющееся одной из главных составляющих многопоточное программирование.

В .NET потоки являются классами Thread. Эти потоки можно создавать, присваивать им конкретные имена, можно менять приоритет потока, приостанавливать, возобновлять работу потока, ожидать завершения работы.

Существует два вида потоков:

1) background (фоновый);

2) foreground (основной).

Foreground-потоки препятствуют завершению программы. Когда все foreground-потоки будут остановлены, система автоматически прекратит работу всех background-потоков и закончит выполнение программы.

Чтобы узнать вид текущего потока, нужно вызвать свойство этого потока. Для это нужно написать

```
Thread.CurrentThread.IsBackground
```

Когда мы создаем поток, то по умолчанию он будет foreground-потоком. Изменить его на background-поток, т. е. вызвать свойство thread.IsBackground.

Посмотрим на простой пример кода с использованием потоков:

```
using System
using System.Threading;
namespace ConsoleAppl
{
class Program
{
Static void Main(string[ ] args)
{
Thread Thread = new Thread(func);
Thread.Start(); запуск потока
For(int i=0; i<10; i++)
{
Console.WriteLine(«Поток 1 вывод» + i);
Thread.Sleep(0);
}
Console.Read();//приостановка основного потока
}
//Функция запускаемая из другого потока
Static void func()
{
               for (int i=0; i<10; i++)
               {
                    Console.WriteLine(«Поток 2 выводит»
                    +i.ToString());
Thread.Sleep(0);
}
          }
          }
     }
```

Thread Thread=new Thread(func) — создает объект потока. Конструктору этого класса передаем имя функции func, возвращающей void, которая вызывается в параллельном потоке. Основной и фоновой потоки параллельно выполняют одинаковый код, называют свой номер и пишут номер цикла. Метод Thread.Sleep(0) приостанавливает поток. Так как ему передан параметр 0, то поток приостановится, чтобы дать возможность выполниться другому потоку.

Чтобы консоль не закрылась, мы используем Console.Read, он будет ожидать ввода с клавиатуры.

```
Поток 1 выводит 0
Поток 2 выводит 0
Поток 1 выводит 1
Поток 2 выводит 1
Поток 1 выводит 2
Поток 2 выводит 2
Поток 1 выводит 3
Поток 2 выводит 3
Поток 1 выводит 4
Поток 1 выводит 5
Поток 2 выводит 4
Поток 1 выводит 6
Поток 1 выводит 7
Поток 2 выводит 5
Поток 1 выводит 8
Поток 2 выводит 6
Поток 1 выводит 9
Поток 2 выводит 7
Поток 2 выводит 8
Поток 2 выводит 9
```

Рис. 8
Результаты выполнения программы

Результат вывода при каждом выполнении программы будет разным. Результат зависит от многих факторов. По дефолту в приоритетном потоке запускается функция Main, другие потоки будут background. Нам нужно проследить, чтобы основной поток не завершился до полного выполнения дочерних потоков. Для этого мы можем воспользоваться полем IsAlive, если поток активный, то он вернет нам значение true, иначе, вернет нам значение false. Также мы можем использовать метод Join(), он блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод.

Еще один пример программы, которая использует потоки:

```
Using System;
Using System.Threading;
Namespace ConsoleApplication1
{
Class Program
{
     class myThread
     {
          Thread thread;
public myThread (string name, int num)
//конструктор получает имя функции
//и номер, до которого ведется счет
{
```

```
thread = new Thread(this.func);
thread.Name = name;
thread.Start(num);//передача параметра в поток
                }
                void func(object num)// функция потока, передаем
                параметр
                {
                        for (int i = 0; i< (int)num; i++)
                        {
                     Console.WriteLine(Thread.CurrentThread.Name+
                     «выводит» +i);
                     Thread.Sleep(0);
                        }
                     Console.WriteLine(Thread.CurrentThread.Name
                     +«завершился»);
                }
           }
           Static void Main(string[] args)
           {
                myThread t1 = new myThread(«Thread 1», 6);
                myThread t2 = new myThread(«Thread 2», 3);
                myThread t3 = new myThread(«Thread 3», 2);
                Console.Read();
           }
     }
}
```

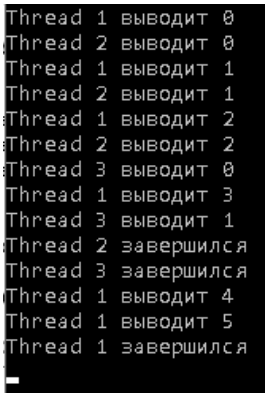


Рис. 9
Вывод программы

В классе myThread есть конструктор, принимающий 2 параметра: строка (имя потока), число, до которого будет вестись счет в цикле. В самом конструкторе создается поток, связанный с func этого объекта. Потоку мы присваиваем имя, для этого используется поле Name созданного потока. Передаем функции Start аргумент и запускаем поток. Функция, которая вызывает поток, принимает один аргумент типа object. Далее проходит цикл в функции func.

Потоки дорого нам обходятся, они занимают много памяти, используют ресурсы системы, да и создаются потоки не мгновенно, на их создание требуется время. Процессы также потребляют большое количество ресурсов, однако

все-таки потоки требуют их больше и на создание, и на уничтожение. Чтобы освободить конкретные ресурсы, которые занимает поток, требуются действия от самого разработчика. Если вам нужно выполнить большое колличество маленьких задач, нежелательно использовать большое количество потоков, иначе затраты на запуск потоков превысят пользу от использования потоков. Поэтому для повторного использования потоков мы можем использовать ThreadPool и избежим затрат на их создание.

ThreadPool является набором потоков. Если нам не хватает количества потоков, пул сам создаст их столько, сколько нужно, и будет использовать по принципу повторного использования. Пул сам определяет, сколько нужно потоков для максимальной эффективности их использования, лишние потоки он может убирать, добавлять дополнительные.

Потоки внутри разделяются на worker и I/O-потоки. Worker потоки связаны с загрузкой CPU, I/O-потоки направлены на работу с устройствами ввода/вывода. Получить количество потоков в пуле можно, написав:

```
ThreadPool.GetAvailableThreads(out workerThreads, out comletir-
tionPortThreads);
```

Для того чтобы определить, что это за поток, нужно написать:

```
Thread.CurrentThread.IsThreadPoollThread
```

- Потокам ThreadPool нельзя дать имя.
- Потоки ThreadPool только фоновые.
- Если заблокировать потоки ThreadPool, то это приведет к созданию дополнительных потоков и производительность сильно снизится.
- Потоку ThreadPool можно изменить приоритет, но когда он вернется в пул, то он вернется к своему дефолтному состоянию.

Синхронизация

Потоки имеют параллельный доступ к разделяемым данным программы, и поэтому их нужно синхронизировать. Синхронизация позволит только одному потоку иметь доступ в определенное время к конкретному блоку кода. Так данные будут гарантированно в сохранности и актуальны.

Синхронизация бывает 4 видов:

– блокировка вызывающего кода;

– конструкции, ограничивающие доступ к кускам кода;

– сигнализирующие конструкции;

– неблокирующая блокировка.

1. Здесь один поток блокируется до того момента пока не завершится другой поток или же блокировка на некоторое время. Для этого используются методы Thread.Sleep() и Thread.Join().

Пример кода:

```
while (!proceed) Thread.Sleep(10);
```

2. Конструкция ограничивает доступ к блокам кода, чтобы только один поток мог пользоваться с конкретным блоком кода. Для этого мы можем использовать lock.

Конструкции, ограничивающие доступ к кускам кода, применяются для того, чтобы удостовериться, что только один поток использует конкретный

участок кода. В .NET существует множество механизмов, позволяющих блокировать доступ к конкретному участку кода. С помощью mutex реализуется межпроцессорная блокировка. SemaphoreSlim позволяют указать, какое количество потоков и процессов может получить доступ к конкретному блоку кода.

3. Сигнализирующие конструкции потоки могут остановиться и ждать, пока поток не получит сигнал от конкретного потока, чтобы поток мог снова продолжить работу.

Наиболее распространенные конструкции:

AutoResetEvent, ManualResetEvent, ManualResetEventSlim, CountdownEvent, Barrier.

4. Неблокирующая блокировка, как видно по названию, не выполняет операции блокировки, остановки и ожидания других потоков, поэтому такой метод будет работать быстрее, чем другие методы. Минус применения данного метода заключается в сложной реализации, где очень легко сделать ошибку.

В следующем подходе компилятор генерирует барьеры памяти при каждом чтении и записи в переменную volatile. Однако его лучше всего использовать, когда у вас один поток, или отдельные потоки читают, а в другие записывают, в ином случае лучше использовать оператор lock.

ПРОЦЕССЫ В UNIX-ПОДОБНЫХ ОПЕРАЦИОННЫХ СИСТЕМАХ

Понятие процесса UNIX. Его контекст. Многозадачность

Процессы в Linux, как и файлы, являются аксиоматическими понятиями. Обычно процесс отождествляют с запущенной программой. Будем считать, что процесс — это рабочая единица системы, которая что-то выполняет. Многозадачность — это возможность одновременного сосуществования нескольких процессов в одной системе.

Linux — многозадачная операционная система. Это означает, что процессы в ней работают одновременно. Естественно, это условная формулировка. Ядро Linux постоянно переключает процессы, то есть время от времени дает каждому из них сколько-нибудь процессорного времени. Переключение происходит довольно быстро, поэтому нам кажется, что процессы работают одновременно.

Одни процессы могут порождать другие процессы, образуя древовидную структуру. Порождающие процессы называются родителями или родительскими процессами, а порожденные — потомками или дочерними процессами. На вершине этого «дерева» находится процесс init, который порождается автоматически ядром в процессе загрузки системы.

Получение информации о процессах в системе

Для получения информации о процессах в системе наиболее часто используются утилиты ps и top. В Linux вся информация о динамике выполнения системы отражается в каталоге /proc, утилиты ps и top собирают данные о запущенных процессах на основании информации, находящейся в этом каталоге.

Контекст процесса складывается из пользовательского контекста и контекста ядра, как изображено на рисунке 10.

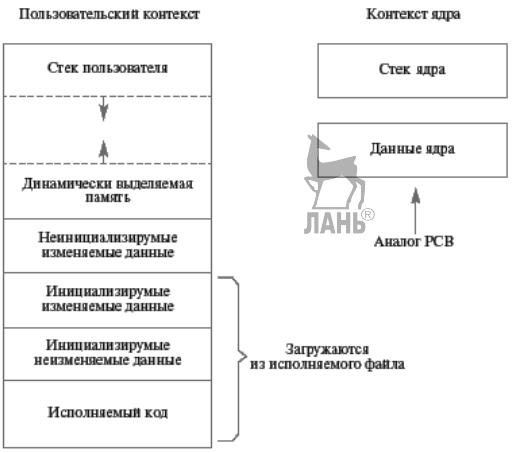


Рис. 10
Контекст процесса в UNIX

Под пользовательским контекстом процесса понимают код и данные, расположенные в адресном пространстве процесса. Все данные подразделяются:

– на инициализируемые неизменяемые данные (например, константы);

– инициализируемые изменяемые данные (все переменные, начальные значения которых присваиваются на этапе компиляции);

– неинициализируемые изменяемые данные (все статические переменные, которым не присвоены начальные значения на этапе компиляции);

– стек пользователя;

– данные, расположенные в динамически выделяемой памяти (например, с помощью стандартных библиотечных С функций malloc(), calloc(), realloc()).

Исполняемый код и инициализируемые данные составляют содержимое файла программы, который исполняется в контексте процесса. Пользовательский стек применяется при работе процесса в пользовательском режиме (user-mode).

Под понятием *«контекст ядра»* объединяются системный контекст и регистровый контекст. Мы будем выделять в контексте ядра стек ядра, который используется при работе процесса в режиме ядра (kernel mode), и данные ядра, хранящиеся в структурах, являющихся аналогом блока управления процессом — PCB. Состав данных ядра будет уточняться на последующих семинарах. На этом занятии нам достаточно знать, что в данные ядра входят: идентификатор пользователя — UID, групповой идентификатор пользователя — GID, идентификатор процесса — PID, идентификатор родительского процесса — PPID.

Идентификация процесса. Иерархия процессов

Каждый процесс в операционной системе получает уникальный идентификационный номер — PID (process identificator). При создании нового процесса операционная система пытается присвоить ему свободный номер больший, чем у процесса, созданного перед ним. Если таких свободных номеров не оказывается (например, мы достигли максимально возможного номера для процесса), то операционная система выбирает минимальный номер из всех свободных номеров. В операционной системе Linux присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс kernel при старте операционной системы. Этот номер впоследствии не может быть присвоен никакому другому процессу. Максимально возможное значение для номера процесса в Linux на базе 32-разрядных процессоров Intel составляет 231-1. Все процессы системы UNIX, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. Процессы с номерами 1 или 0 могут выступать в качестве прародителя всех остальных процессов в системах, подобных UNIX.

Таким образом, все процессы в UNIX связаны отношениями процесс-родитель — процесс-потомок и образуют генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-потомка, идентификатор родительского процесса в данных ядра процесса-

потомка (PPID — parent process identificator) изменяет свое значение на значение 1, соответствующее идентификатору процесса init, время жизни которого определяет время функционирования операционной системы. Тем самым процесс init как бы «усыновляет осиротевшие процессы».

Сотояния процесса. Краткая диаграмма состояний

Модель состояний процессов в операционной системе UNIX представляет собой детализацию модели состояний. Краткая диаграмма состояний процессов в операционной системе UNIX изображена на рисунке 11.

Как мы видим, состояние процесса «исполнение» расщепилось на два состояния: «исполнение в режиме ядра» и «исполнение в режиме пользователя». В состоянии «исполнение в режиме пользователя» процесс выполняет прикладные инструкции пользователя. В состоянии «исполнение в режиме ядра» выполняются инструкции ядра операционной системы в контексте текущего процесса (например, при обработке системного вызова или прерывания). Из состояния «исполнение в режиме пользователя» процесс не может непосредственно перейти в состояния «ожидание», «готовность» и «закончил исполнение». Такие переходы возможны только через промежуточное состояние «исполняется в режиме ядра». Также запрещен прямой переход из состояния «готовность» в состояние «исполнение в режиме пользователя».

Рис. 11
Сокращенная диаграмма состояний процесса в UNIX

Понятие системного вызова

В любой операционной системе поддерживается некоторый механизм, который позволяет пользовательским программам обращаться за услугами ядра ОС UNIX, такие средства называются системными вызовами. Смысл системных вызовов состоит в том, что для обращения к функциям ядра ОС используются «специальные команды» процессора, при выполнении которых возникает

особого рода внутреннее прерывание процессора, переводящее его в режим ядра (в большинстве современных ОС этот вид прерываний называется trap — ловушка). При обработке таких прерываний ядро ОС распознает, что на самом деле прерывание является запросом к ядру со стороны пользовательской программы на выполнение определенных действий, выбирает параметры обращения и обрабатывает его, после чего выполняет «возврат из прерывания», возобновляя нормальное выполнение пользовательской программы. Понятно, что конкретные механизмы возбуждения внутренних прерываний по инициативе пользовательской программы различаются в разных аппаратных архитектурах. Поскольку ОС UNIX стремится обеспечить среду, в которой пользовательские программы могли бы быть полностью мобильны, потребовался дополнительный уровень, скрывающий особенности конкретного механизма возбуждения внутренних прерываний. Этот механизм обеспечивается так называемой библиотекой системных вызовов.

Для пользователя библиотека системных вызовов представляет собой обычную библиотеку заранее реализованных функций системы программирования языка Си. При программировании на языке Си использование любой функции из библиотеки системных вызовов ничем не отличается от использования любой собственной или библиотечной Си-функции. Однако внутри любой функции конкретной библиотеки системных вызовов содержится код, являющийся, вообще говоря, специфичным для данной аппаратной платформы. Поведение всех программ в системе вытекает из поведения системных вызовов, которыми они пользуются. Сам термин «системный вызов» как раз означает «вызов системы для выполнения действия», т. е. вызов функции в ядре системы. Ядро работает в привилегированном режиме — режим ядра, в котором имеет доступ к системным таблицам, регистрам и портам внешних устройств и диспетчера памяти, к которым обычным программам доступ аппаратно запрещен.

Системные вызовы getuid и getgid. Узнать идентификатор пользователя, запустившего программу на исполнение (UID), и идентификатор группы, к которой он относится (GID), можно с помощью системных вызовов getuid() и getgid(), применив их внутри этой программы.

Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid(void);
gid_t getgid(void);
```

Описание системных вызовов

Системный вызов getuid возвращает идентификатор пользователя для текущего процесса.

Системный вызов getgid возвращает идентификатор группы пользователя для текущего процесса.

Типы данных uid_t и gid_t являются синонимами для одного из целочисленных типов языка Си.

Системные вызовы getppid() и getpid(). Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова getpid(), а значение идентификатора родительского процесса для текущего процесса — с помощью системного вызова getppid(). Системные вызовы не имеют параметров и возвращают идентификатор текущего процесса и идентификатор родительского процесса соответственно.

Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Описание системных вызовов

Системный вызов getpid возвращает идентификатор текущего процесса.

Системный вызов getppid возвращает идентификатор процесса-родителя для текущего процесса.

Тип данных pid_t является синонимом для одного из целочисленных типов языка Си.

Программа 1 использования getpid() и getppid() для извлечения идентификаторов процессов

```
#include <iostream>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
#include <cstdlib>
using namespace std;
void mpinfo()
{
     cout<< «id process PID: « <<getpid()<<<\n»;
     cout<< «id parent process PPID: « <<getppid()<<< \n»;
     cout<< «id group process PGID: « <<getpgrp()<<<\n»;
     cout<< «user real id UID: « <<getuid()<<<\n»;
     cout<< «real id group user -GID: « <<getgid()<<< \n»;
     cout<< «effect id user UID: « <<getuid()<<<\n»;
     cout<< «effect id group GID: « <<getgid()<<<\n»;
}
int main()
{
     int i,st;
     signal(SIGCHLD, SIG_IGN);
     for(i=1;i<=3;i++) {
fork();
mpinfo();
cout<<«**********»<<endl;
wait(&st);// ожидание завершения процесса
exit(0);
}
}
```

При запуске данной программы мы видим, что у каждого процесса есть несколько типов идентификаторов, основные типы — это реальные и эффективные. Реальный идентификатор пользователя (или группы) сообщает, кто создал процесс, а эффективный идентификатор пользователя (или группы) сообщает, от чьего лица выполняется процесс, если эта информация изменяется. На примере GID и effectid user обычно не меняются, потому что запускаете программу и выполняете ее только вы (1000). Также стоит обратить внимание на идентификаторы PPID (идентификатор процесса родителя) у процесса-потомка: это значение равно PID родителя. При этом существует еще один тип идентификаторов IDGROUPPROCESS, родители и потомки относятся к одной группе процессов.

Компиляция программ на языке Си в UNIX и запуск их на исполнение

Для компиляции программ в Linux применяются компиляторы gcc и g++. Для нормальной работы компилятора необходимо, чтобы исходные файлы, содержащие текст программы, имели имена, заканчивающиеся на .c.

В простейшем случае откомпилировать программу можно, запуская компилятор командой

```
g++ имя_исходного_файла.
```


Если программа была написана без ошибок, то компилятор создаст исполняемый файл с именем a.out.

Изменить имя создаваемого исполняемого файла можно, задав его с помощью опции -o:

```
g++ имя_исходного_файла -o имя_исполняемого_файла.
```

Компилятор g++ имеет несколько сотен возможных опций. Получить информацию о них возможно в UNIX Manual.

Запустить программу на исполнение можно, набрав имя исполняемого файла и нажав клавишу <Enter>.

Создание процесса в UNIX. Системный вызов fork()

Программное порождение процессов осуществляется с использованием системного вызова ***fork()*** — после выполнения этого вызова в системе появляется процесс, который является почти точной копией процесса, выдавшего данный системный вызов. У порожденного процесса по сравнению с родительским процессом (на уровне уже полученных знаний) изменяются значения следующих параметров:

– идентификатор процесса — PID;

– идентификатор родительского процесса — PPID.

Дополнительно может измениться поведение порожденного процесса по отношению к некоторым сигналам.

Системный вызов для порождения нового процесса

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Описание системного вызова

Процесс, который инициировал системный вызов fork, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

1) идентификатор процесса;

2) идентификатор родительского процесса;

3) время, оставшееся до получения сигнала SIGALRM;

4) сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу;

5) процесс-родитель и процесс-потомок разделяют один и тот же кодовый сегмент. Системный вызов fork() в случае успеха возвращает родительскому процессу идентификатор потомка, а потомку 0. В ситуации, когда процесс не может быть создан, функция fork() возвращает -1. В процессе выполнения системного вызова fork() порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах. Этот системный вызов является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в процессе-родителе и один раз в процессе-ребенке)! После выхода из системного вызова оба процесса продолжают выполнение регулярного пользовательского кода, следующего за системным вызовом.

Для иллюстрации сказанного рассмотрим следующие примеры программ.

Программа 2. Пример создания нового процесса с одинаковой работой процессов ребенка и родителя

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int main()
{
    pid_t pid, ppid;
    int a = 0;
    (void) fork();
    /* При успешном создании нового процесса с этого места
    псевдопараллельно начинают работать два процесса: старый и
    новый */
    /* Перед выполнением следующего выражения значение переменной
a в обоих процессах равно 0 */
    a = a+1;
    /* Узнаем идентификаторы текущего и родительского процесса
    (в каждом из процессов!!!) */
    pid = getpid();
    ppid = getppid();
    /* Печатаем значения PID, PPID и вычисленное значение перемен-
ной a (в каждом из процессов!!!) */
          cout << «My pid = « << (int)pid << endl;
```

```
          cout << «my ppid = « << (int)ppid << endl;
cout << «result = « << a << endl;
return 0;
}
```

Программа 3. Пример создания нового процесса с распечаткой индентификаторов

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <cstdlib>
#include <iostream>
#include <wait.h>
using namespace std;
int main()
{
     pid_t pid;
     int i;
     switch(pid=fork()) {
     case -1:
          perror(«fork»);
          exit(1);
     case 0:
          cout << » CHILD: This is child» << endl;
          cout << » CHILD: PID - « << getpid() << endl;
          cout << » CHILD: PID my parent « << getppid() << endl;
          cout << » CHILD: write code return: «;
          cin >> i;
          cout << « CHILD: exit!» << endl;
          exit(i);
     default:
          cout << «PARENT: This is parent» << endl;
          cout << »PARENT: My Pid - « << getpid() << endl;
          cout << »PARENT: PID my child « << pid << endl;
          cout << «PARENT: I wait my child» << endl;
          wait(0);
          cout << »PARENT: Code return my child « << WEXITSTATUS(i)
          << endl;
          cout << «PARENT: exit!» << endl;
  }
}
```

В данном случае процесс разделяется на родительский процесс и на процесс-потомок, после чего процесс-потомок спрашивает код своего завершения и завершается, все это время процесс-родитель ждет его завершения. После завершения процесса-потомка родитель получает код завершения потомка, выводит его и завершается сам.

Завершение процесса. Функция exit()

Существует два способа корректного завершения процесса в программах, написанных на языке Си. Первый способ мы использовали до сих пор: процесс корректно завершался по достижении конца функции main() или при выполнении оператора return в функции main(). Второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция exit() из стандартной библиотеки функций для языка С. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние «закончил исполнение».

Возврата из функции в текущий процесс не происходит и функция ничего не возвращает.

Значение параметра функции exit() — кода завершения процесса — передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. На самом деле при достижении конца функции main() также неявно вызывается эта функция со значением параметра 0.

Функция для нормального завершения процесса

Прототип функции

```
#include <stdlib.h>
void exit(int status);
```

Параметры функции main() в языке Си. Переменные среды и аргументы командной строки

У функции main() в языке программирования Си существует три параметра, которые могут быть переданы ей операционной системой.

```
void main(int argc, char *argv[], char *envp[]);
```

Если вы наберете команду

```
$ a.out a1 a2 a3,
```

где a.out — имя запускаемой вами программы, то функция main программы из файла a.out вызовется с:

```
argc = 4 /* количество аргументов */
argv[0] = «a.out» argv[1] = «a1»
argv[2] = «a2» argv[3] = «a3»
argv[4] = NULL
```

По соглашению argv[0] содержит имя выполняемого файла.

Первые два параметра при запуске программы на исполнение командной строкой позволяют узнать полное содержание командной строки. Вся командная строка рассматривается как набор слов, разделенных пробелами. Через параметр argc передается количество слов в командной строке, которой была запущена программа. Параметр argv является массивом указателей на отдельные слова. Так, например, если программа была запущена командой

```
$ ./a.out 12 abcd;
```

то значение параметра argc будет равно 3, argv[0] будет указывать на имя программы — первое слово — «a.out», argv[1] — на слово «12», argv[2] — на слово «abcd». Так как имя программы всегда присутствует на первом месте в команд-

ной строке, то argc всегда больше 0, а argv[0] всегда указывает на имя запущенной программы.

Анализируя в программе содержимое командной строки, мы можем предусмотреть ее различное поведение в зависимости от слов, следующих за именем программы. Таким образом, не внося изменений в текст программы, мы можем заставить ее работать по-разному от запуска к запуску. Например, компилятор gcc, вызванный командой gcc 1.c будет генерировать исполняемый файл с именем a.out, а при вызове командой gcc 1.c -o 1.exe — файл с именем 1.exe.

Третий параметр — envp — является массивом указателей на параметры окружающей среды процесса. Начальные параметры окружающей среды процесса задаются в специальных конфигурационных файлах для каждого пользователя и устанавливаются при входе пользователя в систему. В дальнейшем они могут быть изменены с помощью специальных команд операционной системы UNIX. Каждый параметр имеет вид: переменная=строка. Такие переменные используются для изменения долгосрочного поведения процессов, в отличие от аргументов командной строки. Например, задание параметра TERM=vt100 может говорить процессам, осуществляющим вывод на экран дисплея, что работать им придется с терминалом vt100. Меняя значение переменной среды TERM, например на TERM=console, мы сообщаем таким процессам, что они должны изменить свое поведение и осуществлять вывод для системной консоли.

Размер массива аргументов командной строки в функции main() мы получали в качестве ее параметра. Так как для массива ссылок на параметры окружающей среды такого параметра нет, то его размер определяется другим способом. Последний элемент этого массива содержит указатель **NULL**.

Изменение пользовательского контекста процесса. Семейство функций для системного вызова exec()

Для изменения пользовательского контекста процесса применяется системный вызов **exec()**, который пользователь не может вызвать непосредственно. Вызов **exec()** заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора (в том числе устанавливает программный счетчик на начало загружаемой программы). Этот вызов требует для своей работы задания имени исполняемого файла, аргументов командной строки и параметров окружающей среды. Для осуществления вызова программист может воспользоваться одной из шести функций: **execlp()**, **execvp()**, **execl()** и **execv()**, **execle()**, **execve()**, отличающихся друг от друга представлением параметров, необходимых для работы системного вызова **exec()**. Взаимосвязь указанных выше функций изображена на рисунке 12.

Системный вызов execve() объявляется в заголовочном файле unistd.h:

int execve (const char * path, char const * argv[], char * const envp[]);

Все очень просто: системный вызов **execve()** заменяет текущий образ процесса программой из файла с именем **path**, набором аргументов **argv** и окружением **envp**. Здесь следует только учитывать, что **path** — это не просто имя

программы, а путь к ней. Иными словами, чтобы запустить **ls**, нужно в первом аргументе указать **«/bin/ls»**.

Рис. 12

Взаимосвязь различных функций для выполнения системного вызова exec()

Массивы строк **argv** и **envp** обязательно должны заканчиваться элементом **NULL**. Кроме того следует помнить, что первый элемент массива **argv** (**argv[0]**) — это имя программы или что-либо иное. Непосредственные аргументы программы отсчитываются от элемента с номером 1.

В случае успешного завершения **execve()** ничего не возвращает, поскольку новая программа получает полное и безвозвратное управление текущим процессом. Если произошла ошибка, то по традиции возвращается -1.

Прототипы функций

```
#include <unistd.h>
int execlp(const char *file,
    const char *arg0,
... const char *argN,(char *)NULL)
int execvp(const char *file, char *argv[])
int execl(const char *path,
    const char *arg0,
... const char *argN,(char *)NULL)
int execv(const char *path, char *argv[])
int execle(const char *path,
    const char *arg0,
... const char *argN,(char *)NULL,
    char * envp[])
int execve(const char *path, char *argv[],
    char *envp[])
```

Описание функций

Аргумент **file** является указателем на имя файла, который должен быть загружен. Аргумент **path** — это указатель на полный путь к файлу, который должен быть загружен.

Аргументы **arg0, ..., argN** представляют собой указатели на аргументы командной строки. Заметим, что аргумент **arg0** должен указывать на имя загружаемого файла. Аргумент **argv** представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель **NULL**.

Аргумент **envp** является массивом указателей на параметры окружающей среды, заданные в виде строк «переменная=строка». Последний элемент этого массива должен содержать указатель **NULL**.

Поскольку вызов функции не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

1) идентификатор процесса;
2) идентификатор родительского процесса;
3) групповой идентификатор процесса;
4) идентификатор сеанса;
5) время, оставшееся до возникновения сигнала SIGALRM;
6) текущую рабочую директорию;
7) маску создания файлов;
8) идентификатор пользователя;
9) групповой идентификатор пользователя;
10) явное игнорирование сигналов;
11) таблицу открытых файлов (если для файлового дескриптора не устанавливался признак «закрыть файл при выполнении exec()»).

Поскольку системный контекст процесса при вызове **exec()** остается практически неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы, после запуска новой программы также не изменяется.

Важно понимать разницу между системными вызовами **fork()** и **exec()**. Системный вызов fork() создает новый процесс, у которого пользовательский контекст совпадает с пользовательским контекстом процесса-родителя. Системный вызов exec() изменяет пользовательский контекст текущего процесса, не создавая новый процесс.

Для иллюстрации использования системного вызова exec() рассмотрим следующие программы.

Программа 4, изменяющая пользовательский контекст процесса (запускающая другую программу)

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[], char *envp[])
{
/*Мы будем запускать команду cat с аргументом командной строки 3-
02.c без изменения параметров среды, т. е. фактически выполнять
команду «cat 3-02.c», которая должна выдать содержимое данного
файла на экран. Для функции execle в качестве имени программы мы
указываем ее полное имя с путем от корневой директории – /bin/cat.
Первое слово в командной строке у нас должно совпадать с именем
запускаемой программы. Второе слово в командной строке – это имя
файла, содержимое которого мы хотим распечатать. */
(void) execle(«/bin/cat», «/bin/cat»;
```

```
                «3-02.c», NULL, envp);
/* Сюда попадаем только при возникновении ошибки */
cout << «Error on program start» << endl;
exit(-1);
return 0;
}
```

Программа 5, изменяющая пользовательский контекст процесса, пример использования функции execv

```
#include <iostream>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
#include <cstdlib>
using namespace std;
int main()
{
     pid_t pid; // прототип системного вызова
     pid=fork();// тут происходит разделение
     switch (pid) {
     case -1:
          cout<<«error»;// в случае возврата fork`ом -1, это
          считается ошибкой
          break;
     case 0:
          execlp(«/bin/ps»,«ps»,NULL);// процесс-потомок выводит
          запущеные процессы
          exit(0);
     default:
          wait(0);// процесс-родитель ждет завершения
          процесса-потомка
          execlp(«who»,«who»,NULL);//процессу-родителю поручаем
          вывести пользователей
          exit(0);
     }
return 0;
}
```

Организация взаимодействия процессов с помощью каналов

Понятие о потоке ввода-вывода

Канал связи (*англ.* **channel, data line**) — система технических средств и среда распространения сигналов для передачи сообщений (не только данных) от источника к получателю (и наоборот). Канал связи, понимаемый в узком смысле (тракт связи), представляет только физическую среду распространения сигналов, например, физическую линию связи.

Каналы связи в зависимости от способа передачи сигналов классифицируют на несколько видов. Симплексный канал направляет сигналы только в одном направлении. Полудуплексный канал позволяет передать сигналы в двух

направлениях, но поочередно. Такая передача экономически целесообразна также в любых типах каналов при взаимодействии партнеров типа запрос-ответ, когда перед ответом необходимо время для обработки запроса.

Когда процесс создает канал, ядро устанавливает два файловых дескриптора для пользования этим каналом. Первый дескриптор используется, чтобы открыть путь ввода в канал (запись), в то время как второй применяется для получения данных из канала (чтение).

Данные, идущие через канал, проходят через ядро. В операционной системе каналы представлены корректным **inode** — индексным дескриптором, который существует в пределах самого ядра, а не в какой-либо физической файловой системе.

Среди всех категорий средств коммуникации наиболее употребительными являются каналы связи, обеспечивающие достаточно безопасное и достаточно информативное взаимодействие процессов.

Существует две модели передачи данных по каналам связи — поток ввода-вывода и сообщения. Из них более простой является потоковая модель, в которой операции передачи/приема информации вообще не интересуются содержимым того, что передается или принимается. Вся информация в канале связи рассматривается как непрерывный поток байт, не обладающий никакой внутренней структурой.

Понятие о pipe. Системный вызов pipe()

Наиболее простым способом для передачи информации с помощью *потоковой модели* между различными процессами или даже внутри одного процесса в операционной системе UNIX является ***pipe*** (канал, труба, конвейер).

Важное отличие pip'а от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

Pipe можно представить себе в виде трубы ограниченной емкости, расположенной внутри адресного пространства операционной системы, доступ к входному и выходному отверстию которой осуществляется с помощью системных вызовов. В действительности ***pipe*** представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера (хотя существуют и другие виды организации). По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной *указатель* никогда не может обогнать *входной* и наоборот. Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется *системный вызов* ***pipe*()**.

Прототип системного вызова

```
#include <unistd.h>
int pipe(int *fd);
```

Описание системного вызова

Системный вызов **pipe** предназначен для создания **pipe** внутри операционной системы.

Параметр **fd** является указателем на массив из двух целых переменных. При нормальном завершении вызова в первый элемент массива — **fd[0**] — будет занесен файловый дескриптор, соответствующий выходному потоку данных **pipe** и позволяющий выполнять только операцию чтения, а во второй элемент массива — **fd[1]** — будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи.

Возвращаемые значения

Системный вызов возвращает значение **0** при нормальном завершении и значение **-1** при возникновении ошибок.

В процессе работы *системный вызов* организует выделение области памяти под *буфер* и указатели и заносит информацию, соответствующую входному и выходному потокам данных, в два элемента *таблицы открытых файлов*, связывая тем самым с каждым *pip*'ом два *файловых дескриптора*. Для одного из них разрешена только операция чтения из *pip*'а, а для другого — только операция записи в *pipe*. Для выполнения этих операций мы можем использовать те же самые системные вызовы *read()* и *write()*, что и при работе с файлами. Естественно, по окончании использования входного или/и выходного потока данных, нужно закрыть соответствующий *поток* с помощью системного вызова *close()* для освобождения системных ресурсов. Необходимо отметить, что, когда все процессы, использующие *pipe*, закрывают все ассоциированные с ним *файловые дескрипторы*, *операционная система* ликвидирует *pipe*. Таким образом, время существования *pip*'а в системе не может превышать время жизни процессов, работающих с ним.

Программа 6, **иллюстрирующая работу с pip'ом в рамках одного процесса**

Пример создания pip'а, записи в него данных, чтения из него и освобождения выделенных ресурсов. Прогон программы для pipe в одном процессе

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
using namespace std;
int main()
{
     int fd[2];
     size_t size;
     char string[] = «Hello, world!»;
     char resstring[14];
     /* Создание pipe */
     if(pipe(fd) < 0)
     {
     /* Если создать pipe не удалось, вывод сообщения об ошибке
     и прекращение работы */
          cout<<<«Can't create pipe»<< endl;
```

```
    }
    /* Запись в pipe 14 байт из массива, т. е. всю строку «Hello,
    world!»
    вместе с признаком конца строки */
    size = write(fd[1], string, 14);
    if(size != 14)
    {
    /* Если записалось меньшее количество байт, вывод сообщения
    об ошибке */
         cout << «Can't write all string»<<endl;
    }
    /* Чтение из pip'a 14 байт в другой массив, т. е. всю
    записанную строку */
    size = read(fd[0], resstring, 14);
    if(size < 0)
    {
    /* Если прочитать не получилось, вывод сообщения об ошибке */
         cout << «Can't read string»<<endl;
    }
    /* Вывод прочитанной строки */
    cout<< resstring<< endl;
    /* Закрытие входного потока*/
    if(close(fd[0]) < 0)
    {
         cout <<«Can't close input stream»<<endl;
    }
    /* Закрытие выходного потока*/
    if(close(fd[1]) < 0)
    {
         cout << «Can't close output stream»<< endl;
    }
return 0;
}
```

Организация связи через pipe между процессом-родителем и процессом-потомком. Наследование файловых дескрипторов при вызовах fork() и exec()

Таблица открытых файлов наследуется процессом-ребенком при порождении нового процесса системным вызовом **fork()** и входит в состав неизменяемой части системного контекста процесса при системном вызове **exec()** (за исключением тех потоков данных, для файловых дескрипторов которых был специальными средствами выставлен признак, побуждающий операционную систему закрыть их при выполнении **exec()**). Это обстоятельство позволяет организовать передачу информации через **pipe** между родственными процессами, имеющими общего прародителя, создавшего **pipe**.

Программа 7 для организации однонаправленной связи между родственными процессами через pipe

Рассмотрим программу, осуществляющую однонаправленную связь между процессом-родителем и процессом-ребенком. Процесс-родитель создает канал и порождает дочерний процесс, который посылает свой идентифика-

тор (pid) в канал. Родительский процесс считывает данные из канала и выводит их на экран.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <iostream>
using namespace std;
int main()
{
     char s[15] ; //для передачи и получения данных
     int fd[2] ; //для получения дескрипторов канала
     if (pipe(fd)<0) //проверка, открылся ли канал
     {
          cout << «error» << endl;
     return 0; // если не удалось создать канал
     }
     if (fork()==0)
     {
          int r = sprintf(s,»CHILD_Pid=%d», getpid());
          //определение, какие данные записывать
          cout << «I am child, MyPid-->«< getpid()<<endl; //вывод
          на экран
          write(fd[1],&s,r); //запись данных в канал
          exit(0);
     return 1 ;
     }
     wait(0); //ожидание завершения процесса-потомка
     read(fd[0],&s,15); // считывание из канала данных
     процесса-потомка
     cout << «I am Parent, I read your pid - « << s << endl;
     //вывод на экран того,что получилось
     close(fd[0]); //закрытие канала чтения
     close(fd[1]); //закрытие канала записи
return 1 ;
}
```

Программа 8, осуществляющая однонаправленную связь через pipe между процессом-родителем и процессом-ребенком

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int main() {
int fd[2], result;
size_t size;
char resstring[14];
/* Создание pipe */
if(pipe(fd) < 0)
{
/* Если создать pipe не удалось, вывод об этом сообщения
```

```
и прекращение работы */
     cout<<<«Can't create pipe»<<endl;
}
/* Порождение нового процесса */
result = fork();
if(result < 0)
{
/* Если создать процесс не удалось, вывод об этом сообщения
и прекращение работы */
     cout<<<« Can't fork child «<<endl;
}
else if (result > 0)
{
     /* Закрытие выходного потока данных */
     close(fd[0]);
     /*Запись в pipe 14 байт, т. е. всю строку «Hello, world!»
     вместе с признаком конца строки */
     size = write(fd[1], «Hello, world!», 14);
     if(size != 14)
     {
/* Если записалось меньшее количество байт, вывод об этом
сообщения и прекращение работы */
          cout<<<« Can't write all string «<<endl;
     }
     /* Закрытие входного потока данных, на этом родитель
     прекращает работу */
     close(fd[1]);
     cout<<<«Parent exit»<<endl;
}
else
{
/* Порожденный процесс унаследовал от родителя таблицу открытых
файлов и, зная файловые дескрипторы, соответствующие pip, может
его использовать. Закрытие входного потока данных*/
     close(fd[1]);
     /* Чтение из pip'а 14 байт в массив, т. е. всю записанную
     строку */
     size = read(fd[0], resstring, 14);
     if(size < 0)
     {
/* Если прочитать не удалось, вывод об этом сообщения и прекраще-
ние работы */
          cout<<<« Can't read string «<<endl;
     }
     /* Печать прочитанной строки */
     cout<< resstring<<endl;
     /* Закрытие входного потока и завершение работы */
     close(fd[0]);
}
return 0;
}
```

Именованные каналы (FIFO-каналы). Понятие FIFO. Использование системного вызова mknod() для создания FIFO. Функция mkfifo()

Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи, получившее название **FIFO (от First Input First Output)** или именованный **pipe**. **FIFO** во всем подобен pip'у, за одним исключением: данные о расположении **FIFO** в адресном пространстве ядра и его состоянии процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании именованного pip'а на диске заводится файл специального типа, обращаясь к которому, процессы могут получить интересующую их информацию. Для создания **FIFO** используется системный вызов **mknod()** или существующая в некоторых версиях UNIX функция **mkfifo()**.

Следует отметить, что при их работе не происходит действительного выделения области адресного пространства операционной системы под именованный **pipe**, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию **FIFO** в памяти при его открытии с помощью уже известного нам системного вызова **open()**.

После открытия именованный pipe ведет себя точно так же, как и неименованный. Для дальнейшей работы с ним применяются системные вызовы **read()**, **write()** и **close()**. Время существования **FIFO** в адресном пространстве ядра операционной системы, как и в случае с pip'ом, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с **FIFO**, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под **FIFO**. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации **FIFO** в дальнейшем.

Использование системного вызова mknod для создания FIFO

Прототип системного вызова

```
#include <sys/stat.h>
#include <unistd.h>
int mknod(char *path, int mode, int dev);
```

Описание системного вызова

Параметр **dev** является несущественным в нашей ситуации, и мы будем всегда задавать его равным 0.

Параметр **path** является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой **FIFO** на диске. Для успешного создания **FIFO** файла с таким именем перед вызовом существовать не должно.

Параметр **mode** устанавливает атрибуты прав доступа различных категорий пользователей к **FIFO**. Этот параметр задается как результат побитовой операции «или» значения **S_IFIFO**, указывающего, что системный вызов должен создать **FIFO**, и некоторой суммы следующих восьмеричных значений:

0400 — разрешено чтение для пользователя, создавшего FIFO;

0200 — разрешена запись для пользователя, создавшего FIFO;

0040 — разрешено чтение для группы пользователя, создавшего FIFO;
0020 — разрешена запись для группы пользователя, создавшего FIFO;
0004 — разрешено чтение для всех остальных пользователей;
0002 — разрешена запись для всех остальных пользователей.

При создании **FIFO** реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса **umask**, а именно — они равны (**0777 & mode**) **& ~umask**.

Возвращаемые значения

При успешном создании FIFO системный вызов возвращает значение 0, при неуспешном — отрицательное значение.

Функция mkfifo

Прототип функции

```
#include <sys/stat.h>
#include <unistd.h>
int mkfifo(char *path, int mode);
```

Описание функции

Функция **mkfifo** предназначена для создания FIFO в операционной системе.

Параметр **path** является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом функции не должно существовать.

Параметр **mode** устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как некоторая сумма следующих восьмеричных значений:

0400 — разрешено чтение для пользователя, создавшего FIFO;
0200 — разрешена запись для пользователя, создавшего FIFO;
0040 — разрешено чтение для группы пользователя, создавшего FIFO;
0020 — разрешена запись для группы пользователя, создавшего FIFO;
0004 — разрешено чтение для всех остальных пользователей;
0002 — разрешена запись для всех остальных пользователей.

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно — они равны (0777 & mode) & ~umask.

Возвращаемые значения

При успешном создании FIFO функция возвращает значение 0, при неуспешном — отрицательное значение.

Важно понимать, что файл типа FIFO не служит для размещения на диске информации, которая записывается в именованный pipe. Эта информация располагается внутри адресного пространства операционной системы, а файл является только меткой, создающей предпосылки для ее размещения.

Не пытайтесь просмотреть содержимое этого файла с помощью Midnight Commander (mc)!!! Это приведет к его глубокому зависанию!

Для иллюстрации взаимодействия процессов через FIFO рассмотрим следующую программу:

Программа 9. FIFO в родственных процессах

Программа, осуществляющая однонаправленную связь через FIFO между процессом-родителем и процессом-ребенком

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int main()
{
     int fd, result;
     size_t size;
     char resstring[14];
     char name[]=«aaa.fifo»;
     /* Обнуление маски создания файлов текущего процесса для того,
     чтобы права доступа у создаваемого FIFO точно соответствовали
     параметру вызова mknod() */
     (void)umask(0);
     /* Создание FIFO с именем aaa.fifo в текущей директории */
     if(mknod(name, S_IFIFO | 0666, 0) < 0)
     {
/* Если создать FIFO не удалось, вывод об этом сообщения и прекра-
щение работы */
          cout<<«Can't create FIFO»<<endl;
     }
     /* Порождение нового процесса */
     if((result = fork()) < 0)
     {
/* Если создать процесс не удалось, вывод об этом сообщения и пре-
кращение работы */
          cout<<«Can't fork child»<<endl;
     }
     else if (result > 0)
     {
/* В родительском процессе открываем FIFO и передаем информацию
процессу-ребенку*/
          if((fd = open(name, O_WRONLY)) < 0)
          {
/* Если открыть FIFO не удалось, вывод об этом сообщения и прекра-
щение работы */
               cout<<«Can't open FIFO for writing»<<endl;
          }
/* Записать в FIFO 14 байт, т. е. всю строку «Hello, world!» вме-
сте с признаком конца строки */
          size = write(fd, «Hello, world!», 14);
          if(size != 14)
          {
```

```
/* Если записалось меньшее количество байт, вывод об этом сообще-
ния и прекращение работы */
                cout<<«Can't write all string to FIFO»<<endl;
          }
/* Закрытие входного потока данных и на этом родитель прекращает
работу */
          close(fd);
          cout<<«Parent exit»<<endl;
     }
     else
     {
          /* Открытие FIFO на чтение.*/
          if((fd = open(name, O_RDONLY)) < 0)
          {
               /* Если открыть FIFO не удалось, печатаем об этом
               сообщение и прекращаем работу */
               cout<<«Can't open FIFO for reading «<<endl;
          }
          /* Чтение из FIFO 14 байт в массив, т. е. всю записанную
          строку */
          size = read(fd, resstring, 14);
          if(size < 0)
          {
               /* Если прочитать не смогли, вывод об этом сообщения
               и прекращение работы */
               cout<<«Can't read string»<<endl;
          }
          /* Печать прочитанной строки */
          cout<<resstring<< endl;
          /* Закрытие входного потока и завершение работы */
          close(fd);
     }
return 0;
}
```

В этой программе информацией между собой обмениваются процесс-родитель и процесс-ребенок. Повторный запуск этой программы приведет к ошибке при попытке создания FIFO, так как файл с заданным именем уже существует. Здесь нужно либо удалять его перед каждым прогоном программы с диска вручную, либо после первого запуска модифицировать исходный текст, исключив из него все, связанное с системным вызовом mknod().

Открытие файла. Системный вызов open()

Файловый дескриптор используется в качестве параметра, описывающего поток ввода-вывода, для системных вызовов, выполняющих операции над этим потоком. Поэтому, прежде чем совершать операции чтения данных из файла и записи их в файл, необходимо поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом open().

Системный вызов open

Прототип системного вызова

```
#include <fcntl.h>
int open(char *path, int flags);
int open(char *path, int flags, int mode);
```

Описание системного вызова

Системный вызов open предназначен для выполнения операции открытия файла и, в случае ее удачного осуществления, возвращает файловый дескриптор открытого файла (небольшое неотрицательное целое число, которое используется в дальнейшем для других операций с этим файлом).

Параметр path является указателем на строку, содержащую полное или относительное имя файла.

Параметр flags может принимать одно из следующих трех значений:

O_RDONLY — если над файлом в дальнейшем будут совершаться только операции чтения;

O_WRONLY — если над файлом в дальнейшем будут осуществляться только операции записи;

O_RDWR — если над файлом будут осуществляться и операции чтения, и операции записи.

Каждое из этих значений может быть скомбинировано посредством операции «побитовое или (|)» с одним или несколькими флагами:

O_CREAT — если файла с указанным именем не существует, он должен быть создан;

O_EXCL — применяется совместно с флагом O_CREAT. При совместном их использовании и существовании файла с указанным именем, открытие файла не производится и констатируется ошибочная ситуация;

O_NDELAY — запрещает перевод процесса в состояние «ожидание» при выполнении операции открытия и любых последующих операциях над этим файлом;

O_APPEND — при открытии файла и перед выполнением каждой операции записи (если она, конечно, разрешена) указатель текущей позиции в файле устанавливается на конец файла;

O_TRUNC — если файл существует, уменьшить его размер до 0, с сохранением существующих атрибутов файла, кроме, быть может, времен последнего доступа к файлу и его последней модификации.

Кроме того, в некоторых версиях операционной системы UNIX могут применяться дополнительные значения флагов:

O_SYNC — любая операция записи в файл будет блокироваться (т. е. процесс будет переведен в состояние «ожидание») до тех пор, пока записанная информация не будет физически помещена на соответствующий нижележащий уровень hardware;

O_NOCTTY — если имя файла относится к терминальному устройству, оно не становится управляющим терминалом процесса, даже если до этого процесс не имел управляющего терминала.

Параметр **mode** устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Он обязателен, если среди заданных флагов присутствует флаг **O_CREAT**, и может быть опущен в противном случае.

Этот параметр задается как сумма следующих восьмеричных значений:

0400 — разрешено чтение для пользователя, создавшего файл;

0200 — разрешена запись для пользователя, создавшего файл;

0100 — разрешено исполнение для пользователя, создавшего файл;

0040 — разрешено чтение для группы пользователя, создавшего файл;

0020 — разрешена запись для группы пользователя, создавшего файл;

0010 — разрешено исполнение для группы пользователя, создавшего файл;

0004 — разрешено чтение для всех остальных пользователей;

0002 — разрешена запись для всех остальных пользователей;

0001 — разрешено исполнение для всех остальных пользователей.

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра **mode** и маски создания файлов текущего процесса **umask**, а именно — они равны **mode & ~umask**.

При открытии файлов типа **FIFO** системный вызов имеет некоторые особенности поведения по сравнению с открытием файлов других типов. Если **FIFO** открывается только для чтения и не задан флаг **O_NDELAY**, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет **FIFO** на запись. Если флаг **O_NDELAY** задан, то возвращается значение файлового дескриптора, ассоциированного с **FIFO**. Если **FIFO** открывается только для записи и не задан флаг **O_NDELAY**, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет **FIFO** на чтение. Если флаг **O_NDELAY** задан, то констатируется возникновение ошибки и возвращается значение -1.

Возвращаемое значение

Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение -1 при возникновении ошибки.

Системные вызовы read(), write()

Для совершения потоковых операций чтения информации из файла и ее записи в файл применяются системные вызовы read() и write().

Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
size_t read(int fd, void *addr,
            size_t nbytes);
size_t write(int fd, void *addr,
             size_t nbytes);
```

Описание системных вызовов

Системные вызовы **read** и **write** предназначены для осуществления потоковых операций ввода (чтения) и вывода (записи) информации над каналами

связи, описываемыми файловыми дескрипторами, т. е. для файлов, **pipe**, **FIFO** и **socket**.

Параметр **fd** является файловым дескриптором созданного ранее потокового канала связи, через который будет отсылаться или получаться информация, т. е. значением, которое вернул один из системных вызовов **open()**, **pipe()** или **socket()**.

Параметр **addr** представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр **nbytes** для системного вызова **write** определяет количество байт, которое должно быть передано, начиная с адреса памяти **addr**. Параметр **nbytes** для системного вызова **read** определяет количество байт, которое мы хотим получить из канала связи и разместить в памяти, начиная с адреса **addr.**

Возвращаемые значения

В случае успешного завершения системный вызов возвращает количество реально посланных или принятых байт. Это значение (большее или равное 0) может не совпадать с заданным значением параметра **nbytes**, а быть меньше, чем оно, в силу отсутствия места на диске или в линии связи при передаче данных или отсутствия информации при ее приеме. При возникновении какой-либо ошибки возвращается отрицательное значение.

Особенности поведения при работе с файлами

При работе с файлами информация записывается в файл или читается из файла, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов **read** возвращает значение 0, то это означает, что файл прочитан до конца.

Системный вызов close()

Прототип системного вызова

```
#include <unistd.h>
int close(int fd);
```

Описание системного вызова

Системный вызов close предназначен для корректного завершения работы с файлами и другими объектами ввода-вывода, которые описываются в операционной системе через файловые дескрипторы: **pipe, FIFO, socket.**

Параметр **fd** является дескриптором соответствующего объекта, т. е. значением, которое вернул один из системных вызовов **open()**, **pipe()** или **socket()**.

Возвращаемые значения

Системный вызов возвращает значение 0 при нормальном завершении и значение –1 при возникновении ошибки.

Особенности поведения вызова open() при открытии FIFO

Системные вызовы **read()** и **write()** при работе с **FIFO** имеют те же особенности поведения, что и при работе с **pip**'ом. Системный вызов **open()** при

открытии **FIFO** ведет себя несколько иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов. Если **FIFO** открывается только для чтения и флаг **O_NDELAY** не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет **FIFO** на запись. Если флаг **O_NDELAY** задан, то возвращается значение файлового дескриптора, ассоциированного с **FIFO**. Если **FIFO** открывается только для записи и флаг **O_NDELAY** не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет **FIFO** на чтение. Если флаг **O_NDELAY** задан, то констатируется возникновение ошибки и возвращается значение **-1**. Задание флага **O_NDELAY** в параметрах системного вызова **open()** приводит и к тому, что процессу, открывшему **FIFO**, запрещается блокировка при выполнении последующих операций чтения из этого потока данных и записи в него.

Для иллюстрации сказанного давайте рассмотрим следующую программу.

Программа 10 для записи информации в файл

Программа, иллюстрирующая использование системных вызовов open(), write() и close() для записи информации в файл

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main()
{
     int fd;
     size_t size;
     char string[] = «Hello, world!»;
/* Обнуление маски создания файлов текущего процесса для того,
чтобы права доступа у создаваемого файла точно соответствовали па-
раметру вызова open() */
     (void)umask(0);
/* Открытие файла с именем myfile в текущей директории только для
операций вывода.
Если файла не существует, то его создание с правами доступа 0666,
т. е. read-write для всех категорий пользователей */
     if((fd = open(«myfile», O_WRONLY | O_CREAT, 0666)) < 0)
     {
/* Если файл открыть не удалось, вывод об этом сообщения и прекра-
щение работы */
          cout<<«Can't open file»<<endl;
     }
/* Запись в файл 14 байт из нашего массива, т. е. всю строку
«Hello, world!» вместе с признаком конца строки */
     size = write(fd, string, 14);
     if(size == write(fd, string, 14))
     {
          cout<<«File written»<<endl;
     }
```

```
     if(size != 14)
     {
/* Если записалось меньшее количество байт, то вывод сообщения об
ошибке */
          cout<<<«Can't write all string»<<endl;
     }
     /* Закрытие файла */
     if(close(fd) < 0)
     {
          cout<<<«Can't close file»<<endl;
     }
     return 0;
}
```

Особенности поведения вызовов read() и write() для pip'а

Системные вызовы **read()** и **write()** имеют определенные особенности поведения при работе с **pip**'ом, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования обменивающихся информацией процессов.

При написании программ, обменивающихся большими объемами информации через **pipe**, необходимо проверять значения, возвращаемые вызовами. За один раз из **pip**'а может прочитаться меньше информации, чем вы запрашивали, и за один раз в **pipe** может записаться меньше информации, чем вам хотелось бы.

Одна из особенностей поведения блокирующегося системного вызова **read()** связана с попыткой чтения из пустого **pip**'а. Если есть процессы, у которых этот **pipe** открыт для записи, то системный вызов блокируется и ждет появления информации. Если таких процессов нет, он вернет значение 0 без блокировки процесса. Эта особенность приводит к необходимости закрытия файлового дескриптора, ассоциированного с входным концом **pip**'а, в процессе, который будет использовать **pipe** для чтения (**close (fd[1])** в процессе-ребенке в программе из раздела «Организация связи через pipe между процессом-родителем и процессом-потомком»). Аналогичной особенностью поведения при отсутствии процессов, у которых **pipe** открыт для чтения, обладает и системный вызов **write()**, с чем связана необходимость закрытия файлового дескриптора, ассоциированного с выходным концом **pip**'а, в процессе, который будет использовать pipe для записи (**close (fd[0])** в процессе-родителе в той же программе).

Программа 11. Пример работы с именованным каналом между процессами одного приложения

Процесс-родитель создает именованный канал и порождает дочерний процесс, который посылает свой идентификатор (pid) в канал. Родительский процесс считывает данные из канала и выводит их на экран.


```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <iostream>
```

```
#include <sys/stat.h>
using namespace std;
int main()
{
     char s[10]; //для передачи и получения данных
     int fd[2]; //для получения дескрипторов канала
     if (mkfifo(«mypipe», S_IFIFO|0666) < 0) //сообщение, если не
     удалось создать канал
     {
          cout << «Can’t create channel»;
     return 0;
     }
     fd[0] = open(«mypipe»,O_RDONLY|O_NONBLOCK);// //получить
     дескриптор для чтения, если не указать флаг O_NONBLOCK,
     процесс заблокирует сам себя
     fd[1] = open(«mypipe»,O_WRONLY); //получить дескриптор
     для записи
     if (fork()==0)
     {//программный код для дочернего процесса
          int r = sprintf(s,»MyPid=%d»,getpid()) ; //записать свой
          pid в s, r — длина строки
          write(fd[1],&s,r); //послать данные s в канал через
          дескриптор для записи
     return 1;
     }
     wait(NULL); //ожидание дочернего процесса необходимо, так как
     функция чтения из канала стала не блокирующей, т. е. если
     дочерний процесс не успеет записать данные в канал, функция
     чтения не получит данных и завершится
     read(fd[0],&s,10); //прочитать данные из канала через
     дескриптор для чтения
     cout << «Parent read: «<< s<<endl; //вывести полученную строку
     на экран
     close(fd[0]); //закрыть дескрипторы канала
     close(fd[1]); //закрыть дескрипторы канала
     unlink(«mypipe»); //удалить канал
return 1;
}
```

Программа 12. Пример работы с именованным каналом между двумя приложениями: читатель и писатель

Первое приложение создает именованный канал, второе приложение (писатель) получает дескриптор канала на запись и записывает в него свой идентификатор (pid). Первое приложение (читатель) ожидает поступления данных в канал и выводит их на экран. Для совместной работы приложений, исходя из особенностей алгоритма, необходимо сначала запустить первое приложение.

Читатель:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>
#include <cstdlib>
#include <iostream>
#include <fcntl.h>
#include <sys/stat.h>
using namespace std;
int main()
{
     char s[15] ;
     int fd;
     mkfifo(«mypipe»,S_IFIFO|0666);
     fd=open(«mypipe»,O_RDONLY);
     read(fd,&s,15);
     cout << «READ: « << s << endl;
     close(fd);
     unlink(«mypipe»);
return 1 ;
}
```

Писатель:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <cstdlib>
#include <iostream>
#include <fcntl.h>
#include <sys/stat.h>
using namespace std;
int main()
{
     char s[15];
     int fd;
     fd=open(«mypipe»,O_WRONLY);
     sprintf(s,»MyPid=%d»,getpid());
     write(fd,&s,15);
     close(fd);
return 1;
}
```

Работа осуществляется с двух консолей (программы заранее откомпилированы, изменения с методичкой минимальны), первая программа читатель, запускаю ее с 1 терминала, он ждет отклика от писателя. Со второго терминала мы запускаем писателя; после того как писатель передал информацию в канал, писатель закрывается, далее читатель получает информацию из канала, выводит на экран и тоже закрывается.

Таблица 3

Особенности поведения при работе с pipe, FIFO и socket

Системный вызов read	
ситуация	**поведение**
Попытка прочитать меньше байт, чем есть в наличии в канале связи	Читает требуемое количество байт и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи
В канале связи находится меньше байт, чем затребовано, но не нулевое количество	Читает все, что есть в канале связи, и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи
Попытка читать из канала связи, в котором нет информации. Блокировка вызова разрешена	Вызов блокируется до тех пор, пока не появится информация в канале связи и пока существует процесс, который может передать в него информацию. Если информация появилась, то процесс разблокируется, и поведение вызова определяется двумя предыдущими строками таблицы. Если в канал некому передать данные (нет ни одного процесса, у которого этот канал связи открыт для записи), то вызов возвращает значение 0. Если канал связи полностью закрывается для записи во время блокировки читающего процесса, то процесс разблокируется, и системный вызов возвращает значение 0
Попытка читать из канала связи, в котором нет информации. Блокировка вызова не разрешена	Если есть процессы, у которых канал связи открыт для записи, системный вызов возвращает значение -1 и устанавливает переменную errno в значение EAGAIN. Если таких процессов нет, системный вызов возвращает значение 0
Попытка записать в канал связи меньше байт, чем осталось до его заполнения	Требуемое количество байт помещается в канал связи, возвращается записанное количество байт
Попытка записать в канал связи больше байт, чем осталось до его заполнения. Блокировка вызова разрешена	Вызов блокируется до тех пор, пока все данные не будут помещены в канал связи. Если размер буфера канала связи меньше, чем передаваемое количество информации, то вызов будет ждать, пока часть информации не будет считана из канала связи. Возвращается записанное количество байт
Попытка записать в канал связи больше байт, чем осталось до его заполнения, но меньше, чем размер буфера канала связи. Блокировка вызова запрещена	Системный вызов возвращает значение -1 и устанавливает переменную errno в значение EAGAIN
В канале связи есть место. Попытка записать в канал связи больше байт, чем осталось до его заполнения, и больше, чем размер буфера канала связи. Блокировка вызова запрещена	Записывается столько байт, сколько осталось до заполнения канала. Системный вызов возвращает количество записанных байт
Попытка записи в канал связи, в котором нет места. Блокировка вызова не разрешена	Системный вызов возвращает значение -1 и устанавливает переменную errno в значение EAGAIN

Продолжение табл. 3

Системный вызов read	
ситуация	**поведение**
Попытка записи в канал связи, из которого некому больше читать, или полное закрытие канала на чтение во время блокировки системного вызова	Если вызов был заблокирован, то он разблокируется. Процесс получает сигнал SIGPIPE. Если этот сигнал обрабатывается пользователем, то системный вызов вернет значение -1 и установит переменную errno в значение EPIPE
Необходимо отметить дополнительную особенность системного вызова write при работе с pip'ами и FIFO. Запись информации, размер которой не превышает размер буфера, должна осуществляться атомарно — одним подряд лежащим куском. Этим объясняется ряд блокировок и ошибок в предыдущем перечне	

Понятие сигнала. Способы возникновения сигналов и виды их обработки

С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прекращает регулярное исполнение, и управление передается механизму обработки сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Типы сигналов (их принято задавать номерами, как правило, в диапазоне от 1 до 31 включительно или специальными символьными обозначениями) и способы их возникновения в системе жестко регламентированы.

Процесс может получить сигнал от:

– hardware (при возникновении исключительной ситуации);

– другого процесса, выполнившего системный вызов передачи сигнала;

– операционной системы (при наступлении некоторых событий);

– терминала (при нажатии определенной комбинации клавиш);

– системы управления заданиями (при выполнении команды kill).

Передачу сигналов процессу в случаях его генерации источниками 2, 3 и 5, т. е., в конечном счете, каким-либо другим процессом, можно рассматривать как реализацию в UNIX сигнальных средств связи.

Существует три варианта реакции процесса на сигнал:

- Принудительно проигнорировать сигнал.
- Произвести обработку по умолчанию: проигнорировать, остановить процесс (перевести в состояние ожидания до получения другого специального сигнала) либо завершить работу с образованием core файла или без него.
- Выполнить обработку сигнала, специфицированную пользователем.

Изменить реакцию процесса на сигнал можно с помощью специальных системных вызовов. Реакция на некоторые сигналы не допускает изменения, и они могут быть обработаны только по умолчанию. Так, например, сигнал с номером 9 — SIGKILL — обрабатывается только по умолчанию и всегда приводит к завершению процесса.

Важным вопросом при программировании с использованием сигналов является вопрос о сохранении реакции на них при порождении нового процесса

или замене его пользовательского контекста. При системном вызове fork() все установленные реакции на сигналы наследуется порожденным процессом.

При системном вызове exec() сохраняются реакции только для тех сигналов, которые игнорировались или обрабатывались по умолчанию. Получение любого сигнала, который до вызова exec() обрабатывался пользователем, приведет к завершению процесса.

Понятия группы процессов, сеанса, лидера группы, лидера сеанса, управляющего терминала сеанса. Системные вызовы getpgrp(), setpgrp(), getpgid(), setpgid(), getsid(), setsid()

Все процессы в системе связаны родственными отношениями и образуют генеалогическое дерево или лес из таких деревьев, где в качестве узлов деревьев выступают сами процессы, а связями служат отношения родитель-ребенок. Все эти деревья принято разделять на группы процессов или семьи (рис. 13).

Группа процессов включает в себя один или более процессов и существует, пока в группе присутствует хотя бы один процесс. Каждый процесс обязательно включен в какую-нибудь группу. При рождении новый процесс попадает в ту же группу процессов, в которой находится его родитель. Процессы могут мигрировать из группы в группу по своему желанию или по желанию другого процесса (в зависимости от версии UNIX). Многие системные вызовы могут быть применены не к одному конкретному процессу, а ко всем процессам в некоторой группе. Поэтому то, как именно следует объединять процессы в группы, зависит от того, как предполагается их использовать. В свою очередь, группы процессов объединяются в сеансы, образуя, с родственной точки зрения, некие кланы семей. Понятие сеанса изначально было введено в UNIX для логического объединения групп процессов, созданных в результате каждого входа и последующей работы пользователя в системе. Поэтому с каждым сеансом может быть связан в системе терминал, называемый управляющим терминалом сеанса, через который обычно и общаются процессы сеанса с пользователем.

Рис. 13
Иерархия процессов в UNIX

Сеанс не может иметь более одного управляющего терминала, и один терминал не может быть управляющим для нескольких сеансов. В то же время могут существовать сеансы, вообще не имеющие управляющего терминала.

Каждая группа процессов в системе получает собственный уникальный номер. Узнать этот номер можно с помощью системного вызова getpgid(). Используя его, процесс может узнать номер группы для себя самого или для процесса из своего сеанса. Не во всех версиях UNIX присутствует данный системный вызов. Здесь возникает столкновение с тяжелым наследием разделения линий UNIX'ов на линию BSD и линию System V. Вместо вызова getpgid() в таких системах существует системный вызов getpgrp(), который возвращает номер группы только для текущего процесса.

Системный вызов getpgid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgid(pid_t pid);
```

Описание системного вызова

Системный вызов возвращает идентификатор группы процессов для процесса с идентификатором **pid**. Узнать номер группы процесс может только для себя самого или для процесса из своего сеанса. При других значениях **pid** системный вызов возвращает значение –1.

Тип данных **pid_t** является синонимом для одного из целочисленных типов языка Си.

Системный вызов getpgrp()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp(void);
```

Описание системного вызова

Системный вызов **getpgrp** возвращает идентификатор группы процессов для текущего процесса.

Для перевода процесса в другую группу процессов, возможно, с одновременным ее созданием, применяется системный вызов **setpgid()**. Перевести в другую группу процесс может либо сам себя (и то не во всякую и не всегда), либо свой процесс-ребенок, который не выполнял системный вызов **exec()**, т. е. не запускал на выполнение другую программу. При определенных значениях параметров системного вызова создается новая группа процессов с идентификатором, совпадающим с идентификатором переводимого процесса, состоящая первоначально только из одного этого процесса. Новая группа может быть создана только таким способом, поэтому идентификаторы групп в системе уникальны. Переход в другую группу без создания новой группы возможен лишь в пределах одного сеанса. В некоторых разновидностях UNIX системный вызов **setpgid()** отсутствует, а вместо него используется системный вызов **setpgrp()**, способный только создавать новую группу процессов с идентификатором, сов-

падающим с идентификатором текущего процесса, и переводить в нее текущий процесс. (В ряде систем, где сосуществуют вызовы **setpgrp()** и **setpgid()**, например в Solaris, вызов **setpgrp()** ведет себя иначе — он аналогичен рассматриваемому ниже вызову **setsid()**).

Системный вызов setpgid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

Описание системного вызова

Системный вызов **setpgid** служит для перевода процесса из одной группы процессов в другую, а также для создания новой группы процессов.

Параметр **pid** является идентификатором процесса, который нужно перевести в другую группу, а параметр **pgid** — идентификатором группы процессов, в которую предстоит перевести этот процесс. Не все комбинации этих параметров разрешены. Перевести в другую группу процесс может либо сам себя (и то не во всякую, и не всегда), либо свой процесс-ребенок, который не выполнял системный вызов **exec()**, т. е. не запускал на выполнение другую программу.

Если параметр **pid** равен 0, то считается, что процесс переводит в другую группу сам себя.

Если параметр **pgid** равен 0, то в Linux считается, что процесс переводится в группу с идентификатором, совпадающим с идентификатором процесса, определяемого первым параметром.

Если значения, определяемые параметрами **pid** и **pgid**, равны, то создается новая группа с идентификатором, совпадающим с идентификатором переводимого процесса, состоящая первоначально только из этого процесса.

Переход в другую группу без создания новой группы возможен только в пределах одного сеанса. В новую группу не может перейти процесс, являющийся лидером группы, т. е. процесс, идентификатор которого совпадает с идентификатором его группы.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов setpgrp()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int setpgrp(void);
```

Описание системного вызова

Системный вызов **setpgrp** служит для перевода текущего процесса во вновь создаваемую группу процессов, идентификатор которой будет совпадать с идентификатором текущего процесса.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки. Процесс, идентификатор которого совпадает с идентификатором его группы, называется лидером группы. Одно из ограничений на применение вызовов **setpgid()** и **setpgrp()** состоит в том, что лидер группы не может перебраться в другую группу. Каждый сеанс в системе также имеет собственный номер. Для того чтобы узнать его, можно воспользоваться системным вызовом **getsid()**. В разных версиях UNIX на него накладываются различные ограничения. В Linux такие ограничения отсутствуют.

Системный вызов getsid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t getsid(pid_t pid);
```

Описание системного вызова

Системный вызов возвращает идентификатор сеанса для процесса с идентификатором **pid**. Если параметр **pid** равен 0, то возвращается идентификатор сеанса для данного процесса.

Использование системного вызова **setsid()** приводит к созданию новой группы, состоящей только из процесса, который его выполнил (он становится лидером новой группы), и нового сеанса, идентификатор которого совпадает с идентификатором процесса, сделавшего вызов. Такой процесс называется лидером сеанса. Этот системный вызов может применять только процесс, не являющийся лидером группы.

Системный вызов setsid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int setsid(void);
```

Описание системного вызова

Этот системный вызов может применять только процесс, не являющийся лидером группы, т. е. процесс, идентификатор которого не совпадает с идентификатором его группы. Использование системного вызова **setsid** приводит к созданию новой группы, состоящей только из процесса, который его выполнил (он становится лидером новой группы), и нового сеанса, идентификатор которого совпадает с идентификатором процесса, сделавшего вызов.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Если сеанс имеет управляющий терминал, то этот терминал обязательно приписывается к некоторой группе процессов, входящей в сеанс. Такая группа процессов называется текущей группой процессов для данного сеанса. Все процессы, входящие в текущую группу процессов, могут совершать операции ввода-вывода, используя управляющий терминал. Все остальные группы процес-

сов сеанса называются фоновыми группами, а процессы, входящие в них — фоновыми процессами. При попытке ввода-вывода фонового процесса через управляющий терминал этот процесс получит сигналы, которые стандартно приводят к прекращению работы процесса. Передавать управляющий терминал от одной группы процессов к другой может только лидер сеанса. Для сеансов, не имеющих управляющего терминала, все процессы являются фоновыми.

При завершении работы процесса-лидера сеанса все процессы из текущей группы сеанса получают сигнал SIGHUP, который при стандартной обработке приведет к их завершению. Таким образом, после завершения лидера сеанса в нормальной ситуации работу продолжат только фоновые процессы. Процессы, входящие в текущую группу сеанса, могут получать сигналы, инициируемые нажатием определенных клавиш на терминале — SIGINT при нажатии клавиш <**CTRL**> и <**c**>, и SIGQUIT при нажатии клавиш <**CTR**L> и <**4**>. Стандартная реакция на эти сигналы — завершение процесса (с образованием core файла для сигнала SIGQUIT). Необходимо ввести еще одно понятие, связанное с процессом, — эффективный идентификатор пользователя. Каждый пользователь в системе имеет собственный идентификатор — UID. Каждый процесс, запущенный пользователем, задействует этот UID для определения своих полномочий. Однако иногда, если у исполняемого файла были выставлены соответствующие атрибуты, процесс может выдать себя за процесс, запущенный другим пользователем. Идентификатор пользователя, от имени которого процесс пользуется полномочиями, и является эффективным идентификатором пользователя для процесса — EUID. За исключением выше оговоренного случая эффективный идентификатор пользователя совпадает с идентификатором пользователя, создавшего процесс.

Системный вызов kill() и команда kill

Из всех перечисленных ранее в разделе «9.4. Аппаратные прерывания (interrupt), исключения (exception), программные прерывания (trap, software interrupt). Их обработка» источников сигнала пользователю доступны только два — команда kill и посылка сигнала процессу с помощью системного вызова **kill()**.

Команда kill обычно используется в следующей форме:

```
kill [-номер] pid
```

Здесь **pid** — это идентификатор процесса, которому посылается сигнал, а номер — номер сигнала, который посылается процессу. Послать сигнал (если у вас нет полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с идентификатором пользователя, посылающего сигнал. Если параметр-номер отсутствует, то посылается сигнал SIGTERM, обычно имеющий номер 15, и реакция на него по умолчанию — завершить работу процесса, который получил сигнал.

Команда kill

Синтаксис команды

```
kill [-signal] [--] pid
kill -l
```

Описание команды

Команда **kill** предназначена для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя. Параметр pid определяет процесс или процессы, которым будут доставляться сигналы. Он может быть задан одним из следующих четырех способов:

Число **n > 0** — определяет идентификатор процесса, которому будет доставлен сигнал.

Число **0** — сигнал будет доставлен всем процессам текущей группы для данного управляющего терминала.

Число **-1** с предваряющей опцией **--** — сигнал будет доставлен (если позволяют полномочия) всем процессам с идентификаторами, большими 1.

Число **n < 0**, где **n** не равно **-1**, с предваряющей опцией **--** — сигнал будет доставлен всем процессам из группы процессов, идентификатор которой равен -n.

Параметр **-signal** определяет тип сигнала, который должен быть доставлен, и может задаваться в числовой или символьной форме, например **-9** или **-SIGKILL**. Если этот параметр опущен, процессам по умолчанию посылается сигнал SIGTERM. Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с идентификатором пользователя, посылающего сигнал.

Опция **-l** используется для получения списка сигналов, существующих в системе в символьной и числовой формах.

Во многих операционных системах предусмотрены еще и дополнительные опции для команды **kill**. При использовании системного вызова **kill()** послать сигнал (не имея полномочий суперпользователя) можно только процессу или процессам, у которых эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

Системный вызов kill()

Прототип системного вызова

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signal);
```

Описание системного вызова

Системный вызов **kill()** предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя. Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал.

Аргумент **pid** описывает, кому посылается сигнал, а аргумент **sig** — какой сигнал посылается. Этот системный вызов умеет делать много разных вещей в зависимости от значения аргументов:

Если **pid > 0** и **sig > 0**, то сигнал с номером **sig** (если позволяют привилегии) посылается процессу с идентификатором pid.

Если **pid = 0**, а **sig > 0**, то сигнал с номером **sig** посылается всем процессам в группе, к которой принадлежит посылающий процесс.

Если **pid = -1**, **sig > 0** и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

Если **pid = -1**, **sig > 0** и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов с **pid = 0** и **pid = 1**).

Если **pid < 0**, но не **-1**, **sig > 0**, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента **pid** (если позволяют привилегии).

Если значение **sig = 0**, то производится проверка на ошибку, а сигнал не посылается, так как все сигналы имеют номера **> 0**. Это можно использовать для проверки правильности аргумента **pid** (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Возвращаемое значение

Системный вызов возвращает 0 при нормальном завершении и –1 при ошибке.

Программа 13. Процесс порождает ребенка, и они оба зацикливаются

Тривиальная программа для иллюстрации понятий «группа процессов», «сеанс», «фоновая группа» и т. д.

```
#include <unistd.h>
int main(void)
{
    (void)fork();
    while(1);
    return 0;
}
```

Используем команду ps с опциями -e и j, которая позволяет получить информацию обо всех процессах в системе и узнать их идентификаторы, идентификаторы групп процессов и сеансов, управляющий терминал сеанса и к какой группе процессов он приписан. Набрав команду «ps -e j» (обратите внимание на наличие пробела между буквами e и j!!!), получим список всех процессов в системе. Колонка PID содержит идентификаторы процессов, колонка PGID — идентификаторы групп, к которым они принадлежат, колонка SID — идентификаторы сеансов, колонка TTY — номер соответствующего управляющего терминала, колонка TPGID (может присутствовать не во всех версиях UNIX, но в Linux есть) — к какой группе процессов приписан управляющий терминал.

Системный вызов signal(). Установка собственного обработчика сигнала

Одним из способов изменения поведения процесса при получении сигнала в операционной системе UNIX является использование системного вызова signal().

Системный вызов signal()

Прототип системного вызова

```
#include <signal.h>
void (*signal (int sig,
void (*handler) (int)))(int);
```

Описание системного вызова

Системный вызов **signal** служит для изменения реакции процесса на какой-либо сигнал. Хотя прототип системного вызова выглядит довольно пугающе, ничего страшного в нем нет. Приведенное выше описание можно словесно изложить следующим образом: функция **signal**, возвращающая указатель на функцию с одним параметром типа int, которая ничего не возвращает, и имеющая два параметра: параметр **sig** типа int и параметр **handler**, служащий указателем на ничего не возвращающую функцию с одним параметром типа int.

Параметр **sig** — это номер сигнала, обработку которого предстоит изменить.

Параметр **handler** описывает новый способ обработки сигнала, это может быть указатель на пользовательскую функцию — обработчик сигнала, специальное значение **SIG_DFL** или специальное значение **SIG_IGN**. Специальное значение **SIG_IGN** используется для того, чтобы процесс игнорировал поступившие сигналы с номером sig, специальное значение **SIG_DFL** — для восстановления реакции процесса на этот сигнал по умолчанию.

Возвращаемое значение

Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Этот системный вызов имеет два параметра: один из них задает номер сигнала, реакцию процесса, на который требуется изменить, а второй определяет, как именно мы собираемся ее менять.

Для первого варианта реакции процесса на сигнал — его игнорирования — применяется специальное значение этого параметра **SIG_IGN**.

Например, если требуется игнорировать сигнал **SIGINT**, начиная с некоторого места работы программы, в этом месте программы мы должны употребить конструкцию

```
(void) signal(SIGINT, SIG_IGN);
```

Для второго варианта реакции процесса на сигнал — восстановления его обработки по умолчанию — применяется специальное значение этого параметра **SIG_DFL**.

Для третьего варианта реакции процесса на сигнал в качестве значения параметра подставляется указатель на пользовательскую функцию обработки сигнала, которая должна иметь прототип вида

```
void *handler(int);
```

Ниже приведен пример скелета конструкции для пользовательской обработки сигнала **SIGHUP**.

```
void *my_handler(int nsig)
{
    <обработка сигнала>
}
int main()
{
    ...
    (void)signal(SIGHUP, my_handler);
    ...
}
```

В качестве значения параметра в пользовательскую функцию обработки сигнала (в скелете — параметр nsig) передается номер возникшего сигнала, так что одна и та же функция может быть использована для обработки нескольких сигналов.

Приведем пример программы, игнорирующей сигнал SIGINT.

Программа 14. Программа, игнорирующая сигнал SIGINT

```
#include <signal.h>
int main(void)
{
    /* Выставление реакции процесса на сигнал SIGINT на
    игнорирование. */
    (void)signal(SIGINT, SIG_IGN);
    /* Начиная с этого места, процесс будет игнорировать
    возникновение сигнала SIGINT. */
    while(1);
    return 0;
}
```

Эта программа не делает ничего полезного, кроме переустановки реакции на нажатие клавиш <**CTRL**> и <**C**> на игнорирование возникающего сигнала и своего бесконечного зацикливания.

Другой пример программы с пользовательской обработкой сигнала SIGINT.

Программа 15. Программа с пользовательской обработкой сигнала SIGINT

```
#include <signal.h>
#include <stdio.h>
#include <iostream>
using namespace std;
/* Функция my_handler – пользовательский обработчик сигнала. */
void my_handler(int nsig)
{
```

```
    cout<< «Receive signal «<< nsig <<«CTRL-C pressed» << endl;
}
int main(void)
{
    /* Выставление реакции процесса на сигнал SIGINT. */
    (void)signal(SIGINT, my_handler);
    /* Начиная с этого места, процесс будет печатать сообщение о
    возникновении сигнала SIGINT.  */
    while(1);
    return 0;
}
```

Эта программа отличается от программы из раздела «Прогон программы, игнорирующей сигнал SIGINT» тем, что в ней введена обработка сигнала SIGINT пользовательской функцией.

До сих пор в примерах игнорировали значение, возвращаемое системным вызовом signal(). На самом деле этот системный вызов возвращает указатель на предыдущий обработчик сигнала, что позволяет восстанавливать переопределенную реакцию на сигнал.

Программа 16. Программа с пользовательской обработкой сигнала SIGINT, возвращающаяся к первоначальной реакции на этот сигнал после пяти его обработок

```
#include <signal.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int i=0; /* Счетчик числа обработок сигнала */
void (*p)(int); /* Указатель, в который будет занесен адрес преды-
дущего обработчика сигнала */
/* Функция my_handler — пользовательский обработчик сигнала */
void my_handler(int nsig)
{
cout<< «Receive signal «<< nsig <<«CTRL-C pressed» << endl;
    i = i+1;
    /* После 5-й обработки возвращение первоначальной реакции
    на сигнал */
    if(i == 5) (void)signal(SIGINT, p);
}
int main(void)
{
    /* Выставление своей реакции процесса на сигнал SIGINT,
    запоминая адрес предыдущего обработчика */
    p = signal(SIGINT, my_handler);
    /*Начиная с этого места, процесс будет 5 раз печатать
    сообщение о возникновении сигнала SIGINT */
    while(1);
    return 0;
}
```

Сигналы SIGUSR1 и SIGUSR2. Использование сигналов для синхронизации процессов

В операционной системе UNIX существует два сигнала, источниками которых могут служить только системный вызов kill() или команда kill, — это сигналы SIGUSR1 и SIGUSR2. Обычно их применяют для передачи информации о происшедшем событии от одного пользовательского процесса другому в качестве сигнального средства связи. При реализации нитей исполнения в операционной системе Linux сигналы SIGUSR1 и SIGUSR2 используются для организации синхронизации между процессами, представляющими нити исполнения, и процессом-координатором в служебных целях. Поэтому пользовательские программы, применяющие в своей работе нити исполнения, не могут задействовать сигналы SIGUSR1 и SIGUSR2.

Завершение порожденного процесса.
Системный вызов waitpid(). Сигнал SIGCHLD

В материалах семинара 3 (раздел «Завершение процесса. Функция exit()») при изучении завершения процесса говорилось о том, что если процесс-ребенок завершает свою работу прежде процесса-родителя и процесс-родитель явно не указал, что он не заинтересован в получении информации о статусе завершения процесса-ребенка, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии «закончил исполнение» (зомби-процесс) либо до завершения процесса-родителя, либо до того момента, когда родитель соблаговолит получить эту информацию. Для получения такой информации процесс-родитель может воспользоваться системным вызовом waitpid() или его упрощенной формой wait(). Системный вызов waitpid() позволяет процессу-родителю синхронно получить данные о статусе завершившегося процесса-ребенка либо блокируя процесс-родитель до завершения процесса-ребенка, либо без блокировки при его периодическом вызове с опцией WNOHANG. Эти данные занимают 16 бит и в рамках нашего курса могут быть расшифрованы следующим образом.

Если процесс завершился при помощи явного или неявного вызова функции **exit()**, то данные выглядят так (старший бит находится слева):

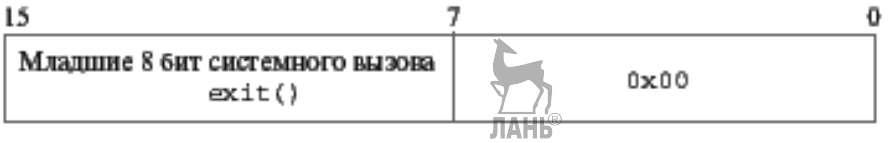


Рис. 14
Результат вызова waitpid() при завершении процесса
с помощью явного или неявного вызова функции exit()

Если процесс был завершен сигналом, то данные выглядят так (старший бит находится слева):

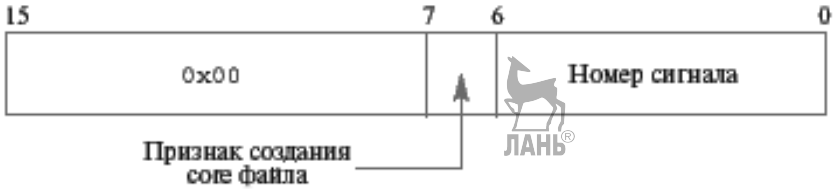


Рис. 15

Результат вызова waitpid() при завершении процесса с помощью сигнала

Каждый процесс-ребенок при завершении работы посылает своему процессу-родителю специальный сигнал SIGCHLD, на который у всех процессов по умолчанию установлена реакция «игнорировать сигнал». Наличие такого сигнала совместно с системным вызовом waitpid() позволяет организовать асинхронный сбор информации о статусе завершившихся порожденных процессов процессом-родителем.

Системные вызовы wait() и waitpid()

Прототипы системных вызовов

```
#include <sys/types.h>
#include <wait.h>
pid_t waitpid(pid_t pid, int *status,
              int options);
pid_t wait(int *status);
```

Описание системных вызовов

Это описание не является полным описанием системных вызовов. Для получения полного описания необходимо обратиться к UNIX Manual. Системный вызов **waitpid()** блокирует выполнение текущего процесса до тех пор, пока либо не завершится порожденный им процесс, определяемый значением параметра **pid**, либо текущий процесс не получит сигнал, для которого установлена реакция по умолчанию «завершить процесс» или реакция обработки пользовательской функцией. Если порожденный процесс, заданный параметром **pid**, к моменту системного вызова находится в состоянии «закончил исполнение», то системный вызов возвращается немедленно без блокирования текущего процесса.

Параметр **pid** определяет порожденный процесс, завершения которого дожидается процесс-родитель, следующим образом:

Если **pid > 0**, ожидаем завершения процесса с идентификатором **pid**.

Если **pid = 0**, ожидаем завершения любого порожденного процесса в группе, к которой принадлежит процесс-родитель.

Если **pid = -1**, ожидаем завершения любого порожденного процесса.

Если **pid < 0**, но не **-1**, ожидаем завершения любого порожденного процесса из группы, идентификатор которой равен абсолютному значению параметра **pid**.

Параметр **option**s может принимать два значения: **0** и **WNOHANG**. Значение **WNOHANG** требует немедленного возврата из вызова без блокировки текущего процесса в любом случае. Если системный вызов обнаружил завер-

шившийся порожденный процесс из числа специфицированных параметром **pid**, то этот процесс удаляется из вычислительной системы, а по адресу, указанному в параметре **status**, сохраняется информация о статусе его завершения. Параметр **status** может быть задан равным **NULL**, если эта информация не имеет для нас значения. При обнаружении завершившегося процесса системный вызов возвращает его идентификатор. Если вызов был сделан с установленной опцией **WNOHANG** и порожденный процесс, специфицированный параметром **pid**, существует, но еще не завершился, системный вызов вернет значение 0. Во всех остальных случаях он возвращает отрицательное значение. Возврат из вызова, связанный с возникновением обработанного пользователем сигнала, может быть в этом случае идентифицирован по значению системной переменной **errno == EINTR**, и вызов может быть сделан снова.

Системный вызов **wait** является синонимом для системного вызова **waitpid** со значениями параметров **pid = -1**, **options = 0**.

Используя системный вызов **signal()**, мы можем явно установить игнорирование этого сигнала (**SIG_IGN**), тем самым проинформировав систему, что нас не интересует, каким образом завершатся порожденные процессы. В этом случае зомби-процессов возникать не будет, но и применение системных вызовов **wait()** и **waitpid()** будет запрещено.

Программа 17 с асинхронным получением информации о статусе завершения порожденного процесса

Программа с асинхронным получением информации о статусе двух завершившихся порожденных процессов

```
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>
#include <stdio.h>
#include <iostream>
using namespace std;
/* Функция my_handler – обработчик сигнала SIGCHLD */
void my_handler(int nsig)
{
    int status;
    pid_t pid;
    /* Опрашиваем статус завершившегося процесса и одновременно
    узнаем его идентификатор */
    if((pid = waitpid(-1, &status, 0)) < 0)
{
        /* Если возникла ошибка – сообщение о ней и продолжение
        работы */
        cout<<<«Some error on waitpid errno «<< errno<< endl;
    }
else
{
        /* Иначе анализирование статуса завершившегося процесса */
        if ((status & 0xff) == 0)
{
```

```
            /* Процесс завершился с явным или неявным вызовом
            функции exit() */
Cout << «Process %d was exited with status %d\n»,pid, status >> 8;
        }
else if ((status & 0xff00) == 0)
{
            /* Процесс был завершен с помощью сигнала */
            Cout << («Process %d killed by signal %d %s\n»,  pid,
            status &0x7f,(status & 0x80) ?  «with core file» :
            «without core file»);
        }
    }
}
int main(void)
{
    pid_t pid;
    /* Установление обработчика для сигнала SIGCHLD */
    (void) signal(SIGCHLD, my_handler);
    /* Порождение Child 1 */
    if((pid = fork()) < 0)
{
        cout<<<«Can't fork child 1»<< endl;
        exit(1);
    }
 else if (pid == 0)
{
        /* Child 1 — завершается с кодом 200 */
        exit(200);
    }
    /* Продолжение процесса-родителя — порождаем Child 2 */
    if((pid = fork()) < 0)
{
        cout<<<«Can't fork child 2»<< endl;
        exit(1);
    }
 else if (pid == 0)
{
        /* Child 2 — циклится, необходимо удалять с помощью сигна-
ла! */
        while(1);
    }
    /* Продолжение процесса-родителя — уходим в цикл */
    while(1);
    return 0;
}
```

В этой программе родитель порождает два процесса. Один из них завершается с кодом 200, а второй зацикливается. Перед порождением процессов родитель устанавливает обработчик прерывания для сигнала SIGCHLD, а после их порождения уходит в бесконечный цикл. В обработчике прерывания вызывается waitpid() для любого порожденного процесса. Так как в обработчик попадаем, когда какой-либо из процессов завершился, системный вызов не блоки-

руется, и можно получить информацию об идентификаторе завершившегося процесса и причине его завершения.

Возникновение сигнала SIGPIPE при попытке записи в pipe или FIFO, который никто не собирается читать

Для pipe и FIFO системные вызовы read() и write() имеют определенные особенности поведения. Одной из таких особенностей является получение сигнала SIGPIPE процессом, который пытается записывать информацию в pipe или в FIFO в том случае, когда читать ее уже некому (нет ни одного процесса, который держит соответствующий pipe или FIFO открытым для чтения). Реакция по умолчанию на этот сигнал — прекратить работу процесса. В процессе изучения курса мы видели ряд системных вызовов, которые могут во время выполнения блокировать процесс. К их числу относятся системный вызов open() при открытии FIFO, системные вызовы read() и write() при работе с pip'ами и FIFO, системные вызовы msgsnd() и msgrcv() при работе с очередями сообщений, системный вызов semop() при работе с семафорами и т. д. Что произойдет с процессом, если он, выполняя один из этих системных вызовов, получит какой-либо сигнал? Дальнейшее поведение процесса зависит от установленной для него реакции на этот сигнал.

Если реакция на полученный сигнал была «игнорировать сигнал» (независимо от того, установлена она по умолчанию или пользователем с помощью системного вызова signal()), то поведение процесса не изменится.

Если реакция на полученный сигнал установлена по умолчанию и заключается в прекращении работы процесса, то процесс перейдет в состояние «закончил исполнение».

Если реакция процесса на сигнал заключается в выполнении пользовательской функции, то процесс выполнит эту функцию (если он находился в состоянии «ожидание», он попадет в состояние «готовность» и затем в состояние «исполнение») и вернется из системного вызова с констатацией ошибочной ситуации (некоторые системные вызовы позволяют операционной системе после выполнения обработки сигнала вновь вернуть процесс в состояние ожидания).

Отличить такой возврат от действительно ошибочной ситуации можно с помощью значения системной переменной errno, которая в этом случае примет значение EINTR (для вызова write и сигнала SIGPIPE соответствующее значение в порядке исключения будет EPIPE).

Чтобы пришедший сигнал SIGPIPE не завершил работу процесса по умолчанию, необходимо его обработать самостоятельно (функция-обработчик при этом может быть и пустой!). Но этого мало. Поскольку нормальный ход выполнения системного вызова был нарушен сигналом, возврат из него будет с отрицательным значением, которое свидетельствует об ошибке. Проанализировав значение системной переменной errno на предмет совпадения со значением EPIPE, можно отличить возникновение сигнала SIGPIPE от других ошибочных ситуаций (неправильные значения параметров и т. д.) и продолжить работу программы.

Понятие о надежности сигналов. POSIX функции для работы с сигналами

Основным недостатком системного вызова signal() является его низкая надежность. Во многих вариантах операционной системы UNIX установленная при его помощи обработка сигнала пользовательской функцией выполняется только один раз, после чего автоматически восстанавливается реакция на сигнал по умолчанию. Для постоянной пользовательской обработки сигнала необходимо каждый раз заново устанавливать реакцию на сигнал прямо внутри функции-обработчика. В системных вызовах и пользовательских программах могут существовать критические участки, на которых процессу недопустимо отвлекаться на обработку сигналов. Можно выставить на этих участках реакцию «игнорировать сигнал» с последующим восстановлением предыдущей реакции, но если сигнал все-таки возникнет на критическом участке, то информация о его возникновении будет безвозвратно потеряна. Наконец, последний недостаток связан с невозможностью определения количества сигналов одного и того же типа, поступивших процессу, пока он находился в состоянии «готовность». Сигналы одного типа в очередь не ставятся! Процесс может узнать о том, что сигнал или сигналы определенного типа были ему переданы, но не может определить их количество.

Этот недостаток можно проиллюстрировать, слегка изменив программу с асинхронным получением информации о статусе завершившихся процессов, рассмотренную ранее в разделе «Изучение особенностей получения терминальных сигналов текущей и фоновой группой процессов».

В новой программе процесс-родитель порождает в цикле пять новых процессов, каждый из которых сразу же завершается со своим собственным кодом, после чего уходит в бесконечный цикл.

Программа 18 для иллюстрации ненадежности сигналов

```
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <cstdlib.h>
#include <iostream>
using namespace std;
/* Функция my_handler – обработчик сигнала SIGCHLD. */
void my_handler(int nsig)
{
    int status;
    pid_t pid;
    /* Опрашиваем статус завершившегося процесса и одновременно
    узнаем его идентификатор. */
    if((pid = waitpid(-1, &status, 0)) < 0)
{
        /* Если возникла ошибка – сообщение о ней и продолжение
        работы. */
        cout<<<«Some error on waitpid errno «<< errno << endl;
```

```
    }
else
 {
        /* Иначе анализируем статус завершившегося процесса. */
        if ((status & 0xff) == 0)
{

            /* Процесс завершился с явным или неявным вызовом
            функции exit(). */
            Cout << «Process %d was exited with status %d\n», pid,
            status >> 8;
        }
 else if ((status & 0xff00) == 0)
{
            /* Процесс был завершен с помощью сигнала. */
            Cout<< («Process %d killed by signal %d %s\n», pid,
            status &0x7f,(status & 0x80) ?»with core file» :
            «without core file»);
        }
    }
}
int main(void)
{
    pid_t pid;
    int i;
    /* Устанавление обработчика для сигнала SIGCHLD. */
    (void) signal(SIGCHLD, my_handler);
    /* В цикле порождение 5 процессов-детей. */
    for (i=0; i < 5; i++)
        if((pid = fork()) < 0)
{
            cout<< «Can't fork child « << i<< endl;
            exit(1);
}
else if (pid == 0)
{
            /* Child i — завершается с кодом 200 + i. */
            exit(200 + i);
        }
        /* Продолжение процесса-родителя — уход на новую
        итерацию. */
    }
    /* Продолжение процесса-родителя — уход в цикл. */
    while(1);
    return 0;
}
```

Последующие версии System V и BSD пытались устранить эти недостатки собственными средствами. Единый способ более надежной обработки сигналов появился с введением POSIX стандарта на системные вызовы UNIX. Набор функций и системных вызовов для работы с сигналами был существенно расширен и построен таким образом, что позволял временно блокировать обработку определенных сигналов, не допуская их потери. Однако проблема, свя-

занная с определением количества пришедших сигналов одного типа, по-прежнему остается актуальной. (Необходимо отметить, что подобная проблема существует на аппаратном уровне и для внешних прерываний. Процессор зачастую не может определить, какое количество внешних прерываний с одним номером возникло, пока он выполнял очередную команду.)

СРЕДСТВА SYSTEM V IPC. ОРГАНИЗАЦИЯ РАБОТЫ С РАЗДЕЛЯЕМОЙ ПАМЯТЬЮ В UNIX. ПОНЯТИЕ НИТЕЙ ИСПОЛНЕНИЯ (THREAD)

Средства межпроцессной коммуникации (IPC)

Средствами межпроцессной коммуникации (IPC — Inter-Process Communication) являются сигналы, каналы, сообщения, семафоры, разделяемая память и сокеты. Процессы выполняются в собственном адресном пространстве, они изолированы друг от друга, поэтому необходимы механизмы для взаимодействия процессов, предоставляемые самой операционной системой и, как правило, расположенные в адресном пространстве системы.

Коммуникация между процессами необходима для решения следующих задач:

1. Передача данных от одного процесса к другому.
2. Совместное использование общих данных несколькими процессами.
3. Синхронизация работы процессов.

Преимущества и недостатки потокового обмена данными

Потоковые механизмы достаточно просты в реализации и удобны для использования, но имеют ряд существенных недостатков.

Операции чтения и записи не анализируют содержимое передаваемых данных. Процесс, прочитавший 20 байт из потока, не может сказать, были ли они записаны одним процессом или несколькими, записывались ли они за один раз или было, например, выполнено 4 операции записи по 5 байт. Данные в потоке никак не интерпретируются системой. Если требуется какая-либо интерпретация данных, то передающий и принимающий процессы должны заранее согласовать свои действия и уметь осуществлять ее самостоятельно.

Для передачи информации от одного процесса другому требуется, как минимум, две операции копирования данных: первый раз — из адресного пространства передающего процесса в системный буфер, второй раз — из системного буфера в адресное пространство принимающего процесса. Процессы, обменивающиеся информацией, должны одновременно существовать в вычислительной системе. Нельзя записать информацию в поток с помощью одного процесса, завершить его, а через некоторое время запустить другой процесс и прочитать записанную информацию.

Понятие о System V IPC

Указанные выше недостатки потоков данных привели к разработке других механизмов передачи информации между процессами. Часть этих механизмов (сообщения, семафоры, разделяемая память), впервые появившихся в UNIX System V и впоследствии перекочевавших оттуда практически во все современные версии операционной системы UNIX, получила общее название System V IPC (IPC — сокращение от interprocess communications). Эти механизмы объе-

диняются в единый пакет, потому что их соответствующие системные вызовы обладают близкими интерфейсами, а в их реализации используются многие общие подпрограммы.

Основные общие свойства всех трех механизмов:

1. Для каждого механизма поддерживается общесистемная таблица, элементы которой описывают все существующие в данный момент части (представители) механизма (конкретные сегменты разделяемой памяти, семафоры или очереди сообщений).

2. Элемент таблицы содержит некоторый числовой ключ, который выбран пользователем именем в качестве представителя соответствующего механизма. Чтобы два или более процесса могли использовать некоторый механизм, они должны заранее договориться об именовании используемого представителя этого механизма.

3. Процесс, желающий начать пользоваться одним из механизмов, обращается к системе с необходимым вызовом, входными параметрами которого является ключ объекта и дополнительные флаги, а ответным параметром является числовой дескриптор, используемый в дальнейших системных вызовах подобно тому, как используется дескриптор файла при работе с файловой системой.

4. Защита доступа к ранее созданным элементам таблицы каждого механизма основывается на тех же принципах, что и защита доступа к файлам.

Пространство имен. Адресация в System V IPC. Функция ftok()

Все средства связи из System V IPC, как и уже рассмотренные pipe и FIFO, являются средствами связи с непрямой адресацией. Для организации взаимодействия неродственных процессов с помощью средства связи с непрямой адресацией необходимо, чтобы это средство связи имело имя. Отсутствие имен у pip'ов позволяет процессам получать информацию о расположении pip'а в системе и его состоянии только через родственные связи. Наличие ассоциированного имени у FIFO — имени специализированного файла в файловой системе — позволяет неродственным процессам получать эту информацию через интерфейс файловой системы. Множество всех возможных имен для объектов какого-либо вида принято называть пространством имен соответствующего вида объектов. Для FIFO пространством имен является множество всех допустимых имен файлов в файловой системе. Для всех объектов из System V IPC таким пространством имен является множество значений некоторого целочисленного типа данных — *key_t* — ключа. Причем программисту не позволено напрямую присваивать значение ключа, это значение задается опосредованно: через комбинацию имени какого-либо файла, уже существующего в файловой системе, и небольшого целого числа — например, номера экземпляра средства связи. Такой хитрый способ получения значения ключа вызван двумя соображениями: если разрешить программистам самим присваивать значение ключа для идентификации средств связи, то не исключено, что два программиста случайно воспользуются одним и тем же значением, не подозревая об этом. Тогда их процессы будут несанкционированно взаимодействовать через одно и то же

средство коммуникации, что может привести к нестандартному поведению этих процессов. Поэтому основным компонентом значения ключа является преобразованное в числовое значение полное имя некоторого файла, доступ к которому на чтение разрешен процессу. Каждый программист имеет возможность использовать для этой цели свой специфический файл, например, исполняемый файл, связанный с одним из взаимодействующих процессов. Преобразование из текстового имени файла в число основывается на расположении указанного файла на жестком диске или ином физическом носителе. Поэтому для образования ключа следует применять файлы, не меняющие своего положения в течение времени организации взаимодействия процессов; второй компонент значения ключа используется для того, чтобы позволить программисту связать с одним и тем же именем файла более одного экземпляра каждого средства связи. В качестве такого компонента можно задавать порядковый номер соответствующего экземпляра. Получение значения ключа из двух компонентов осуществляется функцией **ftok()**.

Функция для генерации ключа System V IPC

Прототип функции

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj);
```

Описание функции

Функция **ftok** служит для преобразования имени существующего файла и небольшого целого числа, например, порядкового номера экземпляра средств связи, в ключ System V IPC.

Параметр pathname должен являться указателем на имя существующего файла, доступного для процесса, вызывающего функцию.

Параметр proj — это небольшое целое число, характеризующее экземпляр средства связи.

В случае невозможности генерации ключа функция возвращает отрицательное значение, в противном случае она возвращает значение сгенерированного ключа. Тип данных *key_t* обычно представляет собой 32-битовое целое.

Дескрипторы System V IPC

Информацию о потоках ввода-вывода, с которыми имеет дело текущий процесс, в частности о pip'ах и FIFO, операционная система хранит в таблице открытых файлов процесса. Системные вызовы, осуществляющие операции над потоком, используют в качестве параметра индекс элемента таблицы открытых файлов, соответствующего потоку, — файловый дескриптор. Использование файловых дескрипторов для идентификации потоков внутри процесса позволяет применять к ним уже существующий интерфейс для работы с файлами, но в то же время приводит к автоматическому закрытию потоков при завершении процесса.

Этим, в частности, объясняется один из перечисленных выше недостатков потоковой передачи информации.

При реализации компонентов System V IPC была принята другая концепция. Ядро операционной системы хранит информацию обо всех средствах System V IPC, используемых в системе, вне контекста пользовательских процессов. При создании нового средства связи или получении доступа к уже существующему процесс получает неотрицательное целое число — дескриптор (идентификатор) этого средства связи, которое однозначно идентифицирует его во всей вычислительной системе. Этот дескриптор должен передаваться в качестве параметра всем системным вызовам, осуществляющим дальнейшие операции над соответствующим средством System V IPC.

Подобная концепция позволяет устранить один из самых существенных недостатков, присущих потоковым средствам связи, — требование одновременного существования взаимодействующих процессов, но в то же время требует повышенной осторожности для того, чтобы процесс, получающий информацию, не принял взамен новых старые данные, случайно оставленные в механизме коммуникации.

Потоки

Потоки предоставляют возможность проведения параллельных или псевдопараллельных, в случае одного процессора, вычислений. Потоки могут порождаться во время работы программы, процесса или другого потока. Основное отличие потоков от процессов заключается в том, что различные потоки имеют различные пути выполнения, но при этом пользуются общей памятью. Таким образом, несколько порожденных в программе потоков могут пользоваться глобальными переменными, и любое изменение данных одним потоком будет доступно и для всех остальных. Путь выполнения потока задается при его создании указанием его стартовой функции, созданный поток начинает выполнять команды этой функции и завершается, когда происходит возврат из функции. Любой поток завершается по окончании работы создавшего его процесса. При создании потока, кроме стартовой функции, ему присуждается буфер для стека, определяемый программистом. Если поток в процессе своей работы превысит размерность стека, выделенного ему программистом, он будет уничтожен системой. Потоки обладают общей памятью, операции с которой также должны защищаться семафорами. Для создания потока используется следующая функция (заголовочный файл — sched.h).

int clone (имя стартовой функции, void *stack, int flags, void *arg).

Функция создает процесс или поток, выполняющий стартовую функцию, стек нового процесса/потока будет храниться в stack, параметр arg определяет входной параметр стартовой функции. Стартовая функция должна иметь такой прототип:

```
int <имя функции>(void *<имя параметра>)
```


Параметр flags может принимать следующие значения:

CLONE_VM — если флаг установлен, создается потомок, обладающий общей памятью с процессом-родителем (поток), если флаг не установлен, создается потомок, которому не доступна память процесса-родителя (процесс).

CLONE_FS — если флаг установлен, потомок обладает той же информацией о файловой системе, что и родитель.

CLONE_FILES — если флаг установлен, потомок обладает теми же файловыми дескрипторами, что и родитель.

CLONE_SIGHAND — если флаг установлен, потомок обладает той же таблицей обработчиков сигналов, что и родитель.

CLONE_VFORK — если флаг установлен, процесс-родитель будет приостановлен до того момента, пока не завершится созданный им потомок.

Если флаги **CLONE_FS**, **CLONE_FILES**, **CLONE_SIGHAND**, **CLONE_VM** не установлены, потомок при создании получает копию соответствующих флагу данных от процесса-родителя. Следует понимать, что копия данных дублирует информацию от родительского процесса в момент создания потомка, но копия и данные, с которых она получена, занимают различные области памяти. Следовательно, изменение данных в процессе работы родителя неизвестно для потомка и наоборот.

Примеры создания потоков

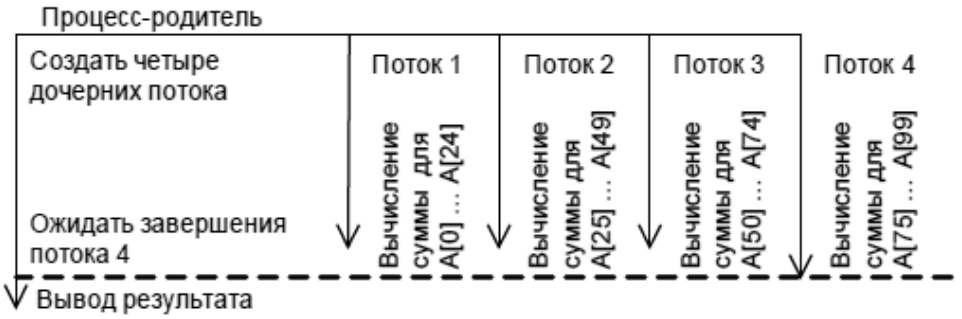


Рис. 16
Схема потоков

Программа 19. Процесс-родитель создает поток, выполняющий функцию func.

```
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <iostream>
using namespace std;
char stack[1000] ; //для хранения стека потока
int func(void *param) //стартовая функция потока
{
     cout<< «Запустился поток»<<endl;
}
int main()
{
     clone(func,(void*)(stack+1000-1), CLONE_VM, NULL) ; //создать
поток
     sleep(1); //блокировка процесса-родителя, чтобы поток успел
выполниться
return 1;
}
```

Программа 20. Процесс-родитель создает четыре потока, вычисляющих сумму элементов определенной части массива.

Созданные потоки выполняют функцию func, получая в качестве параметра индивидуальное целое число от 0 до 3, с помощью которого определяются границы вычисления массива каждым потоком.

```
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <iostream>
#define NUMSTACK 5000 //объем стека для отдельного потока
using namespace std;
int A[100] = {1, 2, 3, 4, 5} ; //массив, сумма элементов которого
вычисляется процессами
int SUM=0 ; //для записи общей суммы
char stack[4][ NUMSTACK] ; //для хранения стека четырех потоков
int func(void *param) //стартовая функция потоков
{
     int i, sum = 0; //для суммирования элементов
     //индекс массива, с которого начинается суммирование
     int p =*(int *)param ;
     p=p * 25;
     for (i=p ; i<p+25 ; i++) sum+=A[i] ; //вычисление суммы части
     элементов массива
          SUM+=sum ; //добавление вычисленного результата в общую
          переменную
return 1 ;
}
int main()
{
     //тут должна быть инициализация элементов массива A
     int param[4] ; //для хранения параметров потоков
     for (int i=0 ; i<3 ; i++) //создание трех потоков
     {
          param[i]=i; //каждому потоку передается уникальное число
          char *tostack=stack[i]; //получить указатель на часть
          массива-стека потоков
          clone(func,(void*)( tostack+ NUMSTACK -1),CLONE_VM, (void
*)(param+i)) ;
     }
//создать поток со стартовой функцией func
//первый поток получает в качестве параметра 0, второй — 1, тре-
тий — 2
     param[3]=3;
     char *tostack=stack[3];
     clone(func,(void*)( tostack+ NUMSTACK -1),
     CLONE_VM|CLONE_VFORK, (void *) (param+3));
//создание четвертого потока, указание процессу дождаться его за-
вершения
     sleep (1);
     cout<<<«Результат = «<< SUM <<endl;
return 1;
}
```

Понятие о нити исполнения (thread) в UNIX. Идентификатор нити исполнения. Функция pthread_self()

Во многих современных операционных системах существует расширенная реализация понятия процесс, когда процесс представляет собой совокупность выделенных ему ресурсов и набора нитей исполнения. Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Поскольку глобальные переменные у нитей исполнения являются общими, они могут использовать их как элементы разделяемой памяти, не прибегая к механизму, описанному выше. В различных версиях операционной системы UNIX существуют различные интерфейсы, обеспечивающие работу с нитями исполнения. Нити исполнения, удовлетворяющие стандарту POSIX, принято называть POSIX thread'ами или кратко — pthread'ами.

Операционная система Linux не полностью поддерживает нити исполнения на уровне ядра системы. При создании нового thread'а запускается новый традиционный процесс, разделяющий с родительским традиционным процессом его ресурсы, программный код и данные, расположенные вне стека, т. е. фактически действительно создается новый thread, но ядро не умеет определять, что эти thread'ы являются составными частями одного целого. Это «знает» только специальный процесс-координатор, работающий на пользовательском уровне и стартующий при первом вызове функций, обеспечивающих POSIX интерфейс для нитей исполнения. Каждая нить исполнения, как и процесс, имеет в системе уникальный номер — идентификатор thread'а. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную нить исполнения, можно узнать идентификатор этой нити и для любого обычного процесса. Для этого используется функция pthread_self(). Нить исполнения, создаваемую при рождении нового процесса, принято называть начальной или главной нитью исполнения этого процесса.

Функция pthread_self()

Прототип функции

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Описание функции

Функция **pthread_self** возвращает идентификатор текущей нити исполнения.

Тип данных **pthread_t** является синонимом для одного из целочисленных типов языка Си.

Создание и завершение thread'а. Функции pthread_create(), pthread_exit(), pthread_join()

Нити исполнения, как и традиционные процессы, могут порождать нити-потомки, правда, только внутри своего процесса. Каждый будущий thread внутри программы должен представлять собой функцию с прототипом:

```
void *thread(void *arg);
```

Параметр arg передается этой функции при создании thread'a и может, до некоторой степени, рассматриваться как аналог параметров функции main(). Возвращаемое функцией значение может интерпретироваться как аналог информации, которую родительский процесс может получить после завершения процесса-ребенка. Для создания новой нити исполнения применяется функция pthread_create().

Функция для создания нити исполнения

Прототип функции

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
pthread_attr_t *attr,
void * (*start_routine)(void *),
void *arg);
```

Описание функции

Функция **pthread_create** служит для создания новой нити исполнения (thread'a) внутри текущего процесса.

Новый thread будет выполнять функцию start_routine с прототипом:

```
void *start_routine(void *)
```

передавая ей в качестве аргумента параметр arg. Если требуется передать более одного параметра, они собираются в структуру; передается адрес этой структуры. Значение, возвращаемое функцией start_routine, не должно указывать на динамический объект данного thread'a. Параметр attr служит для задания различных атрибутов создаваемого thread'a.

Возвращаемые значения

При удачном завершении функция возвращает значение 0 и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр thread. В случае ошибки возвращается положительное значение (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле <**errno.h**>. Значение системной переменной errno при этом не устанавливается. Результатом выполнения этой функции является появление в системе новой нити исполнения, которая будет выполнять функцию, ассоциированную со thread'ом, передав ей специфицированный параметр, параллельно с уже существовавшими нитями исполнения процесса.

Созданный thread может завершить свою деятельность тремя способами.

1. С помощью выполнения функции **pthread_exit()**. Функция никогда не возвращается в вызвавшую ее нить исполнения. Объект, на который указывает параметр этой функции, может быть изучен в другой нити исполнения, например, в породившей завершившийся thread. Этот параметр, следовательно, должен указывать на объект, не являющийся локальным для завершившегося thread'a, например, на статическую переменную.

2. С помощью возврата из функции, ассоциированной с нитью исполнения. Объект, на который указывает адрес, возвращаемый функцией, как и в предыдущем случае, может быть изучен в другой нити исполнения, например, в

породившей завершившийся thread, и должен указывать на объект, не являющийся локальным для завершившегося thread'a.

3. Если в процессе выполняется возврат из функции **main()** или где-либо в процессе (в любой нити исполнения) осуществляется вызов функции **exit()**, это приводит к завершению всех thread'ов процесса.

Функция для завершения нити исполнения

Прототип функции

```
#include <pthread.h>
void pthread_exit(void *status);
```

Описание функции

Функция **pthread_exit** служит для завершения нити исполнения (thread) текущего процесса.

Функция никогда не возвращается в вызвавший ее thread. Объект, на который указывает параметр status, может быть впоследствии изучен в другой нити исполнения, например, в нити, породившей завершившуюся нить. Поэтому он не должен указывать на динамический объект завершившегося thread'a.

Одним из вариантов получения адреса, возвращаемого завершившимся thread'ом, с одновременным ожиданием его завершения является использование функции **pthread_join()**. Нить исполнения, вызвавшая эту функцию, переходит в состояние «ожидание» до завершения заданного thread'a. Функция позволяет также получить указатель, который вернул завершившийся thread в операционную систему.

Функция pthread_join()

Прототип функции

```
#include <pthread.h>
int pthread_join (pthread_t thread,
void **status_addr);
```

Описание функции

Функция **pthread_join** блокирует работу вызвавшей ее нити исполнения до завершения thread'a с идентификатором thread. После разблокирования в указатель, расположенный по адресу **status_addr**, заносится адрес, который вернул завершившийся thread либо при выходе из ассоциированной с ним функции, либо при выполнении функции **pthread_exit()**. Если не интересует, что вернула нить исполнения, в качестве этого параметра можно использовать значение **NULL**.

Возвращаемые значения

Функция возвращает значение 0 при успешном завершении. В случае ошибки возвращается положительное значение (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле <errno.h>. Значение системной переменной errno при этом не устанавливается.

Для иллюстрации вышесказанного рассмотрим программу, в которой работают две нити исполнения.

Программа 21 для иллюстрации работы двух нитей исполнения

Каждая нить исполнения просто увеличивает на 1 разделяемую переменную a.

```
#include <pthread.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int a = 0;
/* Переменная a является глобальной статической для всей программы, поэтому она будет разделяться обеими нитями исполнения.*/
/* Ниже следует текст функции, которая будет ассоциирована со 2-м thread'ом */
void *mythread(void *dummy)
/* Параметр dummy в функции не используется и присутствует только для совместимости типов данных. По той же причине функция возвращает значение void, хотя это никак не используется в программе.*/
{
     pthread_t mythid; /*Для идентификатора нити исполнения */
     /* Переменная mythid является динамической локальной
     переменной функции mythread(), т. е. помещается в стеке и,
     следовательно, не разделяется нитями исполнения. */
     /* Запрос идентификатор thread'a */
     mythid = pthread_self();
     a = a+1;
     cout << «Thread « << mythid << endl;
     cout << «Calculation result = « << a << endl;
return NULL;
}
/* Функция main() —
она же ассоциированная функция главного thread'a*/
int main()
{
     pthread_t thid, mythid;
     int result;
     /* Создание новой нити исполнения, ассоциированной с функцией
     mythread(). Передача ей в качестве параметра значение NULL.
     В случае удачи в переменную thid занесется идентификатор
     нового thread'a. Если возникнет ошибка, завершение работы. */
     result = pthread_create( &thid, (pthread_attr_t *)NULL,
     mythread, NULL);
     if(result != 0)
     {
     cout<<<«Error on thread create, return value = « << result <<
     endl;
     }
     cout<<<«Thread created, thid = « << thid <<endl;
     /* Запрос идентификатора главного thread'a */
     mythid = pthread_self();
     a = a+1;
     cout<<<«Thread « << mythid<< endl;
     cout<< «Calculation result = «<< a << endl;
     /* Ожидание завершения порожденного thread'a, не интересуясь,
```

```
        какое значение он вернет. Если не выполнить вызов этой
        функции, то возможна ситуация, когда завершится функция main()
        до того, как выполнится порожденный thread, что автоматически
        повлечет за собой его завершение, исказив результаты. */
        pthread_join(thid, (void **)NULL);
return 0;
}
```

Для сборки исполняемого файла при работе редактора связей необходимо явно подключить библиотеку функций для работы с pthread'ами, которая не подключается автоматически. Это делается с помощью добавления к команде компиляции и редактирования связей параметра -lpthread — подключить библиотеку pthread.

Необходимость синхронизации процессов и нитей исполнения, использующих общую память.

Все рассмотренные примеры являются не совсем корректными. В большинстве случаев они работают правильно, однако возможны ситуации, когда совместная деятельность этих процессов или нитей исполнения приводит к неверным и неожиданным результатам. Это связано с тем, что любые неатомарные операции, связанные с изменением содержимого разделяемой памяти, представляют собой критическую секцию процесса или нити исполнения.

При одновременном существовании двух процессов в операционной системе может возникнуть следующая последовательность выполнения операций во времени:

...

Процесс 1: **array[0] += 1;**

Процесс 2: **array[1] += 1;**

Процесс 1: **array[2] += 1;**

Процесс 1: **cout<<«Program 1 was spawn» << array[0] << «times, program 2 –» <<array[0] << «times, total –» << array[0]<< «times»<< endl;**

Тогда печать будет давать неправильные результаты. Естественно, что воспроизвести подобную последовательность действий практически нереально. Невозможно подобрать необходимое время старта процессов и степень загруженности вычислительной системы. Но можно смоделировать эту ситуацию, добавив в обе программы достаточно длительные пустые циклы перед оператором **array[2] += 1;** Это проделано в последующих программах.

Сравнение возможностей, достоинств и недостатков различных средств синхронизации процессов

На сегодняшний день вопросы межпроцессорного взаимодействия и синхронизации актуальны для многих программистов, так как от этого зависит эффективность выполнения задач и затраченное на это время.

Программа в момент выполнения называется процессом. У каждого процесса есть свое адресное пространство и свои ресурсы. Выполнение процессов в мультипрограммной среде в общем случае происходит независимо друг от друга, другими словами процессы асинхронны. Но им часто приходится взаи-

модействовать между собой, используя аппаратные и информационные ресурсы вычислительной системы. Например: два процесса обрабатывают данные из одного и того же файла, или же один процесс передает данные другому. Поэтому очень важно правильно организовывать это взаимодействие, что и делает синхронизация.

Программист может использовать как свои средства и приемы синхронизации, так и средства операционной системы. Обычно разработчики операционных систем предоставляют широкий спектр средств синхронизации. Часто функциональные возможности разных системных вызовов синхронизации перекрываются, так что для решения одной задачи программист может воспользоваться несколькими вызовами в зависимости от своих личных предпочтений.

Рассмотрим наиболее частые проблемы, возникающие при неправильной синхронизации.

Ситуация состязания или гонки возникает, когда два или более процессов обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей их исполнения. Если два процесса используют один и тот же ресурс, не синхронизируя свои действия, тогда может получиться следующая не самая приятная ситуация. Процессы A и B одновременно отправили по одному файлу на печать. Для этого каждый процесс считывает индекс свободной ячейки в очереди заданий на печать, помещает туда имя файла и увеличивает индекс. Затем демон печати обрабатывает эту очередь. И может так случиться, что в момент чтения n-го индекса время процесса A заканчивается и управление переходит к процессу B. Тогда процесс B помещает имя файла по n-му индексу и увеличивает его. Затем управление возвращается к процессу A. Но он уже считал номер n, поэтому не будет делать этого повторно. Процесс A просто занесет имя файла по тому же n-му индексу, что и процесс B, и увеличит индекс. Из-за чего файл процесса B сотрется, а демон печати ничего не заподозрит.

Взаимоблокировки, называемые также дедлоками (deadlocks), клинчами (clinch) или тупиками. Иногда бывает, что две или большее число программ зависают, ожидая действий друг друга. Например, процессу A может понадобиться ресурс, но он занят процессом B. Из-за чего A переходит в состояние ожидания. Но если процесс B тоже находится в состоянии ожидания, то ни один из них не может перейти в другое состояние. Также взаимоблокировки могут возникать при некорректном использовании спин-блокировок, мьютексов и семафоров и практически любых синхронизационных примитивов. Даже объекты потоков могут стать причиной появления тупика. Распознавание тупика является нетривиальной задачей, особенно когда он образован многими процессами, использующими много ресурсов. Разумеется, каждый случай такой «неразберихи» уникален.

Разобраться с подобными проблемами может помочь распределение ресурсов и запрет одновременной записи и чтения общих ресурсов более чем одним процессом. Поэтому важным понятием является «критическая секция». Критическая секция — это часть программы, в которой есть обращение к общим ресурсам.

Для правильной совместной работы процессов необходимо выполнение четырех условий:

– два процесса не должны одновременно находиться в критической области (взаимоисключение);

– в программе не должно быть предположений о скорости или количестве процессов;

– процесс, находящийся вне критической области, не может блокировать другие процессы;

– невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

Простым решением задачи может служить запрет прерываний. В этом случае никто не может вмешаться в работу процесса, когда он находится внутри критической секции. Но у такого подхода есть большой минус. Такое решение позволяет процессу пользователя разрешать или запрещать прерывания во всей вычислительной системе. То есть пользователь случайно (или же нарочно) может запретить прерывания, тем самым зациклить процесс или завершить его. Тогда придется перезагружать систему. Также эффективность может заметно снизиться из-за ограничения возможности чередования программ. Кроме того, в многопроцессорной архитектуре очень сложно реализовать такое решение. При включении нескольких процессоров вполне возможно (и обычно так и бывает), что одновременно выполняются несколько процессов. В этом случае запрет прерываний не гарантирует выполнения взаимоисключений.

Блокирующая переменная или алгоритм активного ожидания. Данный метод предполагает существование глобальной переменной, которая меняет свое значение в зависимости от того, находится ли какой-либо процесс в критической области или нет. Но здесь может образоваться гонка процессов, похожая на случай, где процессы А и В отправляли файлы на печать. В этом случае желательно дополнительно предусматривать средства, запрещающие прерывания процесса, находящегося в критической области. Также использование блокирующих переменных имеет один недостаток: пока один процесс находится в критической секции, другой процесс только и делает, что опрашивает блокирующую переменную, бесполезно тратя время.

Для устранения таких ситуаций может быть использован так называемый аппарат событий. Так могут решаться не только проблемы взаимного исключения, но и более общие задачи синхронизации процессов. В разных операционных системах аппарат событий реализуется по-своему, но в любом случае используются системные функции аналогичного назначения. Суть этого аппарата заключается в том, что процесс не запрашивает каждый раз блокирующую переменную, а ждет, когда освободится нужный ресурс. То есть процесс перестает быть активным и переходит в режим ожидания. После освобождения ресурса процесс переходит в состояние готовности.

Очевидный способ синхронизации заключается в том, что все процессы используют один и тот же ресурс строго по очереди. Здесь используются общая переменная turn (показывающая, чья очередь входить в критическую область) и запрет прерываний. Но, допустим, есть два быстродействующих процесса А и

B. Процесс А попадает в критическую область, быстро выполняет свою задачу и предает эстафету процессу В, который быстро выполняет некритическую секцию и переходит в режим активного ожидания. И в этот момент, когда оба процесса находятся вне критической секции, В блокирует А. Что нарушает условие 3 и является большой проблемой. Она возникает из-за того, что процессы не знают о состоянии друг друга. Чтобы исправить это, можно сделать флаги, оповещающие о готовности процесса. В качестве такого флага создаем массив ready [i]. При готовности процесса А массиву ready [i] присваивается значение 1, по выходу из критической области обнуляем его. Если же есть процесс В, готовый войти в критическую секцию, или он уже находится там, то процесс А не входит в критическую секцию. Очевидно, что данный алгоритм обеспечивает взаимоисключение, но, к сожалению, нарушает 4-е условие. Если два процесса одновременно приготовятся войти в критическую область, они будут бесконечно долго ждать друг друга.

Предотвратить подобное состояние можно, присвоив приоритеты процессам. Приоритеты бывают динамическими (изменяющимися во времени) и абсолютными (фиксированными). В любом случае выполнять будет процесс с наивысшим приоритетом. И у этого решения есть проблемы. Во-первых, иногда сложно решить, какой же процесс главнее. Во-вторых, процессы с меньшим приоритетом могут не дождаться своей очереди, и тогда у них наступит голодание.

Для решения проблемы «взаимной вежливости», лучше будет упорядочить доступ к одной ячейке памяти. Для этого можно применить алгоритмы Деккера и Петерсона. Оба способа имеют одинаковую суть и, несомненно, оба правильны. Единственное различие заключается в том, что Деккер решил задачу достаточно сложным путем, из-за чего корректность его алгоритма доказать не так легко. Поэтому заострим свое внимание на алгоритме Петерсона. Здесь также используются массив ready [i] и переменная turn.

```
shared int ready[2] = {0, 0};
shared int turn;
while (some condition)
{
ready[i] = 1;
turn =1-i;
while(ready[1-i] && turn == 1-i);
critical section
ready[i] = 0;
remainder section
}
```

Когда процесс А заявляет о своей готовности войти в критическую секцию, он одновременно вежливо предлагает процессу В войти первым. Если они оба заявили о своей готовности почти одновременно, то они предложат выполняться друг другу. И выполняться будет процесс, которому предложили выполняться последним. То есть, А готов и предлагает В войти, и в этот момент В тоже готов и предлагает войти А. Последнее приглашение остается за А, поэтому он входит в критическую секцию первым.

Присвоим нашим процессам индексы и получится A0 и B1. Тогда процесс B1 может войти, только если ready[i – 1] = 0 и turn = 1, что означает: A0 не находится в состоянии готовности и наступила очередь B. Оба процесса не могут оказаться в критической секции одновременно, потому что тогда переменная turn имела бы одновременно значения 0 и 1, что невозможно. Но вернемся к началу, из-за чего мы решили применить этот алгоритм. В предыдущем примере со строгими очередями не выполнялось 4-е условие. Из сказанного выше следует, что A0 не может выполниться при ready[1] = 1 и turn = 1. Но если B1 не готов (ready[1] = 0), то A0 может входить. Иначе, если B1 готов, то выполняется тот процесс, который пригласили последним. После выполнения он сбрасывает флаг ready[i], давая возможность войти следующему процессу. Таким образом, 4-е условие выполняется.

Описанные выше способы являются абсолютно верными, но они громоздкие и занимают много процессорного времени. Поэтому были разработаны средства синхронизации более высокого уровня. Одним из первых предложенных механизмов является семафор. Фундаментальный принцип заключается в том, что процессы могут сотрудничать посредством простых сигналов, так что в определенном месте процесс может приостановить работу до тех пор, пока не дождется соответствующего сигнала. Требования к операции любой степени сложности могут быть удовлетворены соответствующей структурой сигналов. Главное преимущество заключается в следующем:

1. Значение семафора расположено не в адресном пространстве некоторого процесса, а в адресном пространстве ядра.

2. Операция проверки и изменения значения семафора выполняется в режиме ядра и поэтому не прерывается другими процессами.

Это помогает организовывать процессы, не находящиеся в системе одновременно.

Семафор представляет собой особый вид положительной целой переменной. Классический набор содержит лишь две операции работы с семафорами P и V, уменьшение значения семафора на единицу и увеличение соответственно. Операция P не может выполняться, если значение семафора равно 0. При выполнении процесса V один процесс, из числа задержанных на выполнении P, может выйти из состояния ожидания и выполнить свою P-операцию.

Набор System V IPC для семафоров содержит 3 операции.

A(S, n) — увеличить значение семафора S на величину n, что соответствует операции V;

D(S, n) — пока значение семафора S < n, процесс блокируется. Далее S = S – n соответствует операции P;

Z(S) — процесс блокируется до тех пор, пока значение семафора S не станет равным 0.

Для создания ключа семафор используются специальные функции, что обеспечивает дополнительную безопасность. Если пользователь будет зада-

вать ключи самостоятельно, то он может случайно выбрать ранее использованный ключ, что может привести к неправильной работе программы.

Семафоры можно и нужно использовать для уменьшения нагрузки на куске кода, исполняемого параллельно. Как правило, таким образом можно вылечить множество тупиков, которые легко найти профилированием кода — когда при увеличении количества потоков время исполнения отдельных функций заметно растет, при том, что другие функции отрабатывают с той же или сравнимой скоростью. Семафоры являются гибким и удобным средством для синхронизации и взаимного исключения процессов, учета ресурсов, скрытые семафоры также используются в операционных системах как основа для других средств взаимодействия процессов.

Иногда используется упрощенная версия семафоров, называемая мьютексами. Его отличие в том, что он не использует счетчики. Мьютекс ведет себя почти как критическая секция с единственным отличием, что последняя является объектом пользовательского режима. Его реализация проста и эффективна для процессов, действующих только в пространстве пользователя. Но в этом случае нельзя будет узнать, что сделал с защищенными данными бывший владелец объекта мьютекса.

Спин-блокировка является, по сути, объектом типа мьютекс, с более широкими полномочиями. Когда фрагмент кода, работающего на уровне режима ядра, собирается обратиться к одной из «охраняемых» структур данных, он должен сначала выполнить запрос на владение спин-блокировкой. Разделение доступа к охраняемым данным между потоками, работающими на разных процессорах, обеспечивается тем, что только один из процессоров в каждый момент времени имеет право собственности на объект спин-блокировки. Не рекомендуется удерживать объект спин-блокировки более 25 микросекунд. Категорически не рекомендуется обращаться к страничной памяти из кода, получившего спин-блокировку: рано или поздно это приведет к краху системы. Чревато опасностями и использование конструкций, в которые заложена зависимость одновременно от нескольких спин-блокировок. По крайней мере, следует избегать получения новых спин-блокировок, когда не освобождены ранее полученные: другой поток, владея запрашиваемыми объектами, в это же время может ожидать доступа к спин-блокировкам, которые отданы первому. И в этом случае возникает тупиковая ситуация.

Но программирование с использованием семафоров требует осторожности. Это связано с правильным порядком обращения к семафорам. Если случайно перепутать P и V операции, то это может привести к тупиковой ситуации. И в сложных программах можно, конечно, вручную проверить все семафоры, но сложно и долго. И для облегчения работы был предложен еще один механизм, называемый монитор. Это особые конструкции языка программирования, у которого есть свои собственные переменные, определяющие его состояние. Изменять эти переменные могут только функции метода монитора, которые могут использовать только данные внутри монитора и свои параметры. Особенностью этого метода является то, что внутри

монитора только один процесс может быть активен. Компилятор может отличить монитор от других функций и обработать его специальным образом, добавив пролог и эпилог для обеспечения взаимоисключения, или же используя семафоры. Из-за чего работа значительно упрощается и уменьшается количество возможных ошибок. Но помимо взаимоисключений необходимы средства организации очередности процесса. Здесь используются операции wait и signal, похожие на семафорные операции P и V соответственно. Но в отличие от семафоров мониторы не запоминают историю, то есть signal всегда выполняется после wait. Иначе информация о произошедшем событии будет утеряна. Несмотря на все достоинства, у мониторов есть существенный недостаток. Мониторы встречаются в таких языках как параллельный Паскаль, Java и так далее. Для обычных широко применяемых языков эмуляция мониторов с помощью системных вызовов гораздо сложнее расстановки семафоров.

Есть очень распространенная проблема в программировании, называемая проблемой читателей-писателей, где несколько читателей и один писатель. Использование мьютекса для ее решения считается медленным, так как нет смысла блокировать ресурс, когда его только читают. Для этого используют примитив RW-lock. В отличие от мьютекса, RW-lock имеет два набора функций блокировки-освобождения ресурса — один для читателей, другой для писателя. Блокировки чтения/записи применяются точно в соответствии с их названием: несколько читателей могут использовать ресурс в отсутствие писателей, или один писатель может использовать ресурс в отсутствие читателей и других писателей.

Очевидно, что в любой момент времени только один писатель может записывать данные в эту структуру. Если бы несколько писателей одновременно попытались писать, то получилась бы неразбериха, так как одни данные могли записаться поверх других. Для предотвращения таких ситуаций писатель должен эксклюзивно получить блокировку чтения/записи, обозначив этим, что он и только он имеет доступ к структуре данных. Заметьте, что это исключительное право доступа зависит только от вас.

С читателями ситуация противоположная. Несколько читателей могут читать файл одновременно без вреда для работы. Единственное условие: файл в это время не должен изменяться, потому что это может ввести читателей в заблуждение. Что может привести к нарушениям целостности данных.

Отметим, что мы не смогли бы реализовать такую форму синхронизации только с помощью мьютекса. Мьютекс был бы хорош в случае записи (чтобы только один поток мог использовать ресурс в определенный момент времени), но оплошал бы в случае считывания, потому что не допустил бы к ресурсу более чем одного читателя. Семафор также был бы бесполезен, потому что он не смог бы отличить читателей от писателей, что вызвало бы некрасивую ситуацию с множеством читателей и множеством же писателей!

Необходимость синхронизации неоспорима. Она нужна для предотвращения гонок и тупиков. Пренебрежение этим может привести к непра-

вильной работе системы или даже к ее краху. Но и злоупотреблять синхронизацией не следует: если при небольшом количестве потоков программа работает быстро, то при увеличении количества потоков скорость, наоборот, может заметно снизиться. Так что средствами межпроцессорного взаимодействия следует пользоваться с умом.

СЕМАФОРЫ В UNIX

Отличие операций над UNIX-семафорами от классических операций

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 г. При разработке средств System V IPC семафоры вошли в их состав как неотъемлемая часть. Общими ресурсами процессов являются файлы, сегменты разделяемой памяти. Возможность одновременного изменения несколькими процессами общих данных называют критической секцией, так как такая совместная работа процессов может привести к возникновению ошибок. Например, если несколько процессов осуществляют запись данных в один и тот же файл, эти данные могут оказаться перемешанными. Наиболее простой механизм защиты критической секции состоит в расстановке «замков», пропускающих только один процесс для выполнения критической секции и останавливающий все остальные процессы, пытающиеся выполнить критическую секцию, до тех пор, пока эту критическую секцию не выполнит пропущенный процесс. Семафоры позволяют выполнять такую операцию, как и многие другие.

Использование общих данных несколькими процессами может привести к ошибкам и конфликтам. Но при этом семафоры и сами являются общими данными. Такое положение не является противоречивым в силу того, что:

1) значение семафора расположено не в адресном пространстве некоторого процесса, а в адресном пространстве ядра;

2) операция проверки и изменения значения семафора, вызываемая процессом, является атомарной, т. е. не прерываемой другими процессами. Эта операция выполняется в режиме ядра.

Общими данными процессов также являются каналы и сообщения. Но операции с каналами и сообщениями защищаются системой, и, как правило, программист не должен использовать семафор для защиты записи сообщения в очередь сообщений либо записи данных в канал. Однако это не всегда правильно. Например, система обеспечивает атомарную запись в канал только для данных не больше определенного объема.

Набор операций над семафорами System V IPC отличается от классического набора операций {P, V}, предложенного Дейкстрой. Он включает три операции:

A(S, n) — увеличить значение семафора S на величину n;

D(S, n) — пока значение семафора S < n, процесс блокируется. Далее S = S – n;

Z(S) — процесс блокируется до тех пор, пока значение семафора S не станет равным 0.

Изначально все IPC-семафоры инициируются нулевым значением.

Классической операции P(S) соответствует операция D(S,1), а классической операции V(S) соответствует операция A(S,1). Аналогом ненулевой ини-

циализации семафоров Дейкстры значением n может служить выполнение операции A(S,n) сразу после создания семафора S, с обеспечением атомарности создания семафора и ее выполнения посредством другого семафора. Классические семафоры реализуются через семафоры System V IPC. Обратное не является верным. Используя операции P(S) и V(S), не получится реализовать операцию Z(S).

Поскольку IPC-семафоры являются составной частью средств System V IPC, то для них верно все, что говорилось об этих средствах в материалах предыдущего семинара. IPC-семафоры являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по его окончании. Пространством имен IPC-семафоров является множество значений ключа, генерируемых с помощью функции ftok(). Для совершения операций над семафорами системным вызовам в качестве параметра передаются IPC-дескрипторы семафоров, однозначно идентифицирующих их во всей вычислительной системе, а вся информация о семафорах располагается в адресном пространстве ядра операционной системы. Это позволяет организовывать через семафоры взаимодействие процессов, даже не находящихся в системе одновременно.

Создание массива семафоров или доступ к уже существующему. Системный вызов semget()

В целях экономии системных ресурсов операционная система UNIX позволяет создавать не по одному семафору для каждого конкретного значения ключа, а связывать с ключом целый массив семафоров (в Linux — до 500 семафоров в массиве, хотя это количество может быть уменьшено системным администратором). Для создания массива семафоров, ассоциированного с определенным ключом, или для доступа по ключу к уже существующему массиву используется системный вызов semget(), являющийся аналогом системного вызова shmget() для разделяемой памяти, который возвращает значение IPC-дескриптора для этого массива. При этом применяются те же способы создания доступа, что и для разделяемой памяти. Вновь созданные семафоры инициируются нулевым значением.

Системный вызов semget()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems,
           int semflg);
```

Описание системного вызова

Системный вызов **semget** предназначен для выполнения операции доступа к массиву IPC-семафоров и, в случае ее успешного завершения, возвращает дескриптор System V IPC для этого массива (целое неотрицательное число, однозначно характеризующее массив семафоров внутри вычислительной системы и использующееся в дальнейшем для других операций с ним).

Параметр **key** является ключом System V IPC для массива семафоров, т. е. фактически его именем из пространства имен System V IPC. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции **ftok()**, или специальное значение **IPC_PRIVATE**. Использование значения **IPC_PRIVATE** всегда приводит к попытке создания нового массива семафоров с ключом, который не совпадает со значением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции **ftok()** ни при одной комбинации ее параметров.

Параметр **nsems** определяет количество семафоров в создаваемом или уже существующем массиве. В случае если массив с указанным ключом уже имеется, но его размер не совпадает с указанным в параметре nsems, констатируется возникновение ошибки.

Параметр **semflg** — флаги — играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву, а также необходимость создания нового массива и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или — « | ») следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT — если массива для указанного ключа не существует, он должен быть создан;

IPC_EXCL — применяется совместно с флагом **IPC_CREAT**. При совместном их использовании и существовании массива с указанным ключом доступ к массиву не производится и констатируется ошибка, при этом переменная errno, описанная в файле <**errno.h**>, примет значение **EEXIST**;

0400 — разрешено чтение для пользователя, создавшего массив;

0200 — разрешена запись для пользователя, создавшего массив;

0040 — разрешено чтение для группы пользователя, создавшего массив;

0020 — разрешена запись для группы пользователя, создавшего массив;

0004 — разрешено чтение для всех остальных пользователей;

0002 — разрешена запись для всех остальных пользователей.

Вновь созданные семафоры инициируются нулевым значением.

Возвращаемое значение

Системный вызов возвращает значение дескриптора System V IPC для массива семафоров при нормальном завершении и значение -1 при возникновении ошибки.

Семафоры для синхронизации процессов

Семафор обладает внутренним значением — целым числом, принадлежащим типу unsigned short. Семафоры могут быть объединены в единую группу.

```
int semop(int semid, sembuf *semop, size_t nops)
```

Функция выполняет над группой семафоров с дескриптором semid набор операций semop, nops — количество операций, выполняемых из набора semop.

Для задания операции над группой семафоров используется структура sembuf.

Первый параметр структуры sembuf определяет порядковый номер семафора в группе. Семафоры в группе индексируются с нуля.

Второй параметр структуры sembuf представляет собой целое число = S и определяет действие, которое необходимо произвести над семафором, с индексом, записанным в первом параметре.

Если S > 0, к внутреннему значению семафора добавляется число S. Эта операция не блокирует процесс.

Если S = 0, процесс приостанавливается, пока внутреннее значение семафора не станет равно нулю.

Если S < 0, процесс должен отнять от внутреннего значения семафора модуль S.

Если значение семафора — |S| ≥ 0, производится вычитание и процесс продолжает свою работу. Если значение семафора — |S| < 0, процесс останавливается до тех пор, пока другой процесс не увеличит значение семафора на достаточную величину, чтобы операция вычитания выдала неотрицательный результат. Тогда производится операция вычитания и процесс продолжает свою работу. Например, если значение семафора равно трем, а процесс пытается выполнить над ним операцию –4, этот процесс будет заблокированным, пока значение семафора не увеличится хотя бы на единицу.

Третий параметр структуры sembuf может быть равен 0, тогда операция S ≥ 0 будет предполагать блокировку процесса, т. е. выполняться так, как описано выше. Также sembuf может быть равен IPC_NOWAIT, в этом случае работа процесса не будет останавливаться. Если процесс будет пытаться выполнить вычитание от значения семафора, дающее отрицательный результат, эта операция просто игнорируется и процесс продолжает выполнение.

Последний параметр в функции semop определяет количество операций, берущихся для выполнения из второго параметра функции.

Т. е. следующий вызов

sembuf Minus4 = {0, -4, 0} ;
semop(semid, &Minus4, 1) ; нельзя заменить таким вызовом.
sembuf Minus1 = {0, -1, 0} ;
semop(semid, &Minus1, 4) ; корректной заменой может являться.
sembuf Minus1[4] = {0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0} ;
semop(semid, Minus1, 4) ;

Для иллюстрации сказанного рассмотрим простейшие программы, синхронизирующие свои действия с помощью семафоров.

Программа 22а для иллюстрации работы с семафорами

Эта программа получает доступ к одному системному семафору, ждет, пока его значение не станет больше или равным 1 после запусков программы 05-1b, затем уменьшает его на 1.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <iostream>
using namespace std;
```

```
int main()
{
     int semid; /* IPC дескриптор для массива IPC семафоров. */
     char pathname[] = «Имя файла»; /* Имя файла, использующееся
     для генерации ключа.     Файл с таким именем должен
     существовать в текущей директории. */
     key_t key; /* IPC ключ */
     struct sembuf mybuf; /* Структура для задания операции
     над семафором. */
     /* Генерирование IPC-ключа из файла в текущей директории
     и номера экземпляра массива семафоров 0. */
     if((key = ftok(pathname,0)) < 0)
     {
          cout<<«Can't generate key»<<endl;
     }
     /* Попытка получить доступ по ключу к массиву семафоров, если
     он существует, или создать его из одного семафора, если его
     еще не существует, с правами доступа read & write для всех
     пользователей. */
     if((semid = semget(key, 1, 0666 | IPC_CREAT)) < 0)
     {
          cout<<«Can't get semid»<<endl;
     } /* Выполнение операции D(semid1,1) для массива семафоров.
     Для этого сначала необходимо заполнить структуру. Флаг,
     равный 0. Массив семафоров состоит из одного семафора
     с номером 0. Код операции -1.*/
     mybuf.sem_op = -1;
     mybuf.sem_flg = 0;
     mybuf.sem_num = 0;
     if(semop(semid, &mybuf, 1) < 0)
     {
          cout<<«Can't wait for condition»<<endl;
     }
     cout<<«Condition is present»<<endl;
return 0;
```

Программа 22b для иллюстрации работы с семафорами

```
/* Эта программа получает доступ к одному системному семафору и
увеличивает его на 1*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int main()
{
     int semid; /* IPC дескриптор для массива IPC семафоров. */
     char pathname[] = «Имя файла»; /* Имя файла, использующееся
     для генерации ключа.     Файл с таким именем должен
     существовать в текущей директории. */
     key_t key; /* IPC ключ */
     struct sembuf mybuf; /* Структура для задания операции над
     семафором. */
```

```
     /* Генерирование IPC ключа из имени файла 08-1a.c в текущей
     директории и номера экземпляра массива семафоров 0. */
     if((key = ftok(pathname,0)) < 0)
     {
          cout<<<«Can't generate key»<<endl;
     } /* Попытка получить доступ по ключу к массиву семафоров,
     если он существует, или создать его из одного семафора, если
     его еще не существует, с правами доступа read & write для всех
     пользователей. */
     if((semid = semget(key, 1, 0666 | IPC_CREAT)) < 0)
     {
          cout<<<«Can't get semid»<<endl;
     } /* Выполнение операции A(semid1,1) для массива семафоров.
     Для этого сначала необходимо заполнить структуру. Флаг,
     равный 0. Массив семафоров состоит из одного семафора
     с номером 0. Код операции 1.*/
     mybuf.sem_op = 1;
     mybuf.sem_flg = 0;
     mybuf.sem_num = 0;
     if(semop(semid, &mybuf, 1) < 0)
     {
          cout<<<«Can't wait for condition»<<endl;
     }
     cout<<<«Condition is set «<<endl;
return 0;
}
```

Первая программа выполняет над семафором S операцию D(S,1), вторая программа выполняет над тем же семафором операцию A(S,1). Если семафора в системе не существует, любая программа создает его перед выполнением операции. Поскольку при создании семафор всегда инициируется 0, то программа 1 может работать без блокировки только после запуска программы 2.

Программа 23. Пример использования семафора для синхронизации процессов

Процесс-родитель создает четыре процесса-потомка и ожидает их завершения, используя для этого семафор.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <iostream>
using namespace std;
int main()
{
     int semid ;
     /* Для хранения дескриптора группы семафоров. */
     /* Создать группу семафоров, состоящую из одного семафора. */
     semid = semget (IPC_PRIVATE, 1, IPC_CREAT|0666) ;
     if ( semid < 0 )
     /* Если не удалось создать группу семафоров, завершить
     выполнение. */
     {
```

```
        cout<< ««Ошибка»«<<endl;
    return 0 ;
    }
    sembuf Plus1 = {0, 1, 0} ;
    /* Операция прибавляет единицу к семафору с индексом 0. */
    sembuf Minus4 = {0, -4, 0} ;
    /* Операция вычитает 4 от семафора с индексом 0. */
    /* Создать четыре процесса-потомка. */
    for (int i=0 ; i<4 ; i++)
    {
        if (fork() == 0)
    /* Истинно для дочернего процесса. */
    {
        cout<<<hi « << i << «!»<<endl;
        /* Здесь должен быть код, выполняемый
        процессом-потомком. */
        /* Добавить к семафору единицу, по окончании работы. */
        semop( semid, &Plus1, 1) ;
    return 1 ;
    }}
    semop( semid, &Minus4, 1) ;
return 1 ;
}
```

В описанном примере новый семафор при создании обладает нулевым значением. Каждый из четырех порожденных процессов после выполнения своих вычислений увеличивает значение семафора на единицу. Родительский процесс пытается уменьшить значение семафора на четыре. Таким образом, процесс-родитель останется заблокированным до тех пор, пока не отработают все его потомки.

Использование семафора для защиты критической секции

Использование семафора для синхронизации доступа нескольких процессов к общему ресурсу, т. е. для защиты критической секции. Общим ресурсом будет являться разделяемая память. Процесс-родитель создает сегмент разделяемой памяти и порождает три дочерних процесса, процесс-родитель и его потомки вычисляют сумму элементов определенной части массива и записывают вычисленную сумму в разделяемую память. Родительский процесс дожидается окончания работы потомков и выводит окончательный результат — сумму всех элементов массива.

Так как четыре процесса могут изменять данные разделяемой памяти, необходимо сделать это изменение атомарным для каждого процесса. Для этого необходимо создать семафор, который будет принимать два значения — ноль и единицу.

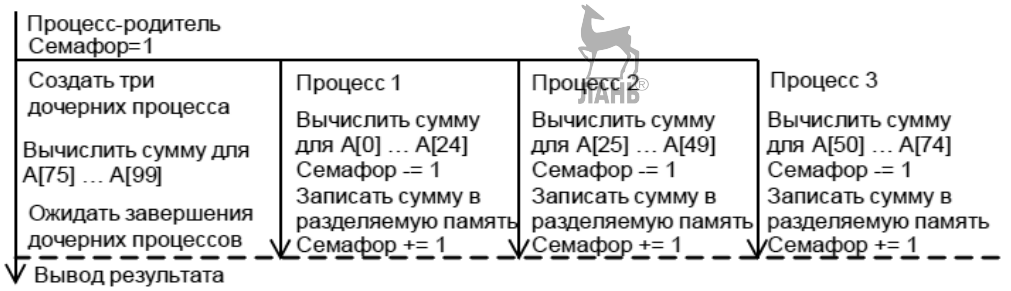


Рис. 17
Схема процессов

Программа 24. Использование семафора для синхронизации доступа нескольких процессов к общему ресурсу (разделяемой памяти)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;
int shmid;
int semid;
sembuf Plus1 = {0, 1, 0};
sembuf Minus1 = {0, -1, 0};
int A[100];
struct mymem
{
     int sum;
}
*mem_sum;
void summa (int p)
{
     int i, sum = 0;
     int begin =25*p;
     int end = begin+25;
     for(i=begin ; i<end ; i++)
          sum+=A[i];
     semop( semid, &Minus1, 1);
     mem_sum->sum+=sum;
     semop( semid, &Plus1, 1);
}
int main()
{
     shmid = shmget(IPC_PRIVATE, 2, IPC_CREAT|0666);
     if (shmid< 0 )
     {
          cout<<<«Ошибка»<< endl;
     return 0;
     }
     semid = semget (IPC_PRIVATE, 1, IPC_CREAT|0666);
```

```
     if ( semid< 0 )
     {
          cout<<<«Ошибка»<< endl;
     return 0;
     }
     semop( semid, &Plus1, 1);
     mem_sum = (mymem *)shmat(shmid,NULL,0);
     mem_sum->sum = 0;
     A[10]=5;
     A[30]=5;
     A[60]=5;
     A[90]=5;
     for (int i=0 ; i<3 ; i++)
     {
          if (fork() == 0     )
          {
               summa(i);
          return 1;
     }}
     summa(3);
     for (int i=0 ; i<3 ; i++)
          wait(NULL);
     cout<<<«Calculate= «<<<mem_sum->sum<< endl;
return 1;
}
```

Программа 25. Иллюстрация использования семафора для защиты критической секции

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;
int shmid ; /* Для хранения дескриптора разделяемой памяти. */
int semid ; /* Для хранения дескриптора группы семафоров. */
sembuf Plus1 = {0, 1, 0} ; /* Операция прибавляет единицу к сема-
фору с индексом 0. */
sembuf Minus1 = {0, -1, 0} ; /* Операция вычитает единицу от сема-
фора с индексом 0. */
int A[100] =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};
/* Массив, сумма элементов которого вычисляется процессами. */
struct mymem /* Структура, под которую будет выделена разделяемая
память. */
{
     int sum ; /* Для записи суммы.*/
}
*mem_sum;
void summa (int p) /* Для вычисления суммы части элементов масси-
ва. */
{
```

```
     int i, sum = 0 ; /* Для суммирования элементов. */
     int begin =25*p ;
/* Индекс массива, с которого начинается суммирование. */
     int end = begin+25 ;
/* Индекс массива, на котором завершается суммирование. */
     for(i=begin; i<end; i++)
          sum+=A[i]; /* Вычисление суммы части элементов
          массива. */
     semop( semid, &Minus1, 1) ; /* Отнять единицу от семафора. */
     mem_sum->sum+=sum; /* Добавление вычисленного результата
     в общую переменную. */
     semop( semid, &Plus1, 1); /* Добавить единицу к семафору. */
}
int main()
{
     /* Запрос на создание разделяемой памяти объемом 2 байта. */
     shmid = shmget(IPC_PRIVATE, 2, IPC_CREAT|0666);
     /* Если запрос оказался неудачным, завершить выполнение. */
     if (shmid < 0 )
     {
          cout<< «Ошибка»<<endl;
     return 0;
     }
     /* Создать группу семафоров, состоящую из одного семафора. */
     semid = semget (IPC_PRIVATE, 1, IPC_CREAT|0666);
     /* Если не удалось создать группу семафоров, завершить
     выполнение. */
     if ( semid < 0 )
     {
          cout<< «Ошибка»<<endl;
     return 0;
     }
     semop( semid, &Plus1, 1) ; /* Теперь семафор равен единице. */
     /* Теперь mem_sum указывает на выделенную разделяемую
     память. */
     mem_sum = (mymem *)shmat(shmid,NULL,0) ;
     mem_sum->sum = 0;
     /* Тут должна быть инициализация элементов массива A. */
     /* Создать три процесса-потомка. */
     for (int i=0; i<3; i++)
     {
          if (fork() == 0     )
     /* Истинно для дочернего процесса. */
          {
               summa(i) ;
          return 1 ;
          }
     }
     summa(3); /* Родительский процесс вычисляет последнюю четверть
     массива. */
     for (int i=0; i<3; i++)
          wait(NULL) ; /* Дождаться завершения процессов-потомков.
*/
```

```
    /* Вывести на экран сумму всех элементов массива. */
    cout<< «Результат = « << mem_sum->sum << endl;
return 1;
}
```

Семафор получает при создании значение, равное нулю, которое сразу же устанавливается в единицу. Первый процесс, вызвавший функцию semop(semid, &Minus1, 1), уменьшает значение семафора до нуля, переходит к записи в разделяемую память и по завершении операции записи устанавливает значение семафора в единицу, вызвав semop(semid, &Plus1, 1). Если управление перейдет к другому процессу во время записи в разделяемую память первым процессом, вызов функции «отнять от семафора единицу» остановит работу другого процесса до того момента, когда значение семафора не станет положительным. Что может произойти только тогда, когда первый процесс завершит запись в общую память и выполнит операцию «добавить к семафору единицу».

Если процессы обладают несколькими общими ресурсами, то необходимо для каждого общего ресурса создавать свой семафор.

Семафоры для синхронизации потоков

Для многопоточного приложения, как и для многопроцессного, критической секцией является изменение несколькими потоками общего ресурса, например, файла или глобальной переменной. Для синхронизации работы потоков и для синхронизации доступа нескольких потоков к общим ресурсам используются семафоры. Но при этом семафоры, используемые для синхронизации потоков, принадлежат к другому стандарту, чем семафоры, используемые для синхронизации процессов.

Каждый семафор содержит неотрицательное целое значение. Любой поток может изменять значение семафора. Когда поток пытается уменьшить значение семафора, происходит следующее: если значение больше нуля, то оно уменьшается, если же значение равно нулю, поток приостанавливается до того момента, когда значение семафора станет положительным, тогда значение уменьшается и поток продолжает работу. Увеличение значения семафора возможно всегда, эта операция не предполагает блокировки. Однако значение семафора не должно выходить за границы типа unsigned int.

Функции для работы с семафорами (заголовочный файл — semaphore.h):

```
int sem_init (sem_t *sem, int pshared, unsigned int value)
```

Функция инициализирует семафор sem и присваивает ему значение value. Если параметр pshared больше нуля, семафор может быть доступен нескольким процессам, если pshared равен нулю, семафор создается для использования внутри одного процесса.

Функция возвращает ноль в случае успеха.

```
Int sem_destroy (sem_t *sem)
```

Функция разрушает семафор sem и возвращает ноль в случае успеха.

```
int sem_getvalue (sem_t *sem, int *sval)
```

Функция записывает значение семафора sem в *sval. Если несколько потоков используют семафор, полученное значение семафора может быть устаревшим.

```
int sem_post (sem_t *sem)
```

Функция увеличивает значение семафора sem на единицу.

```
int sem_wait (sem_t *sem)
```

Если текущее значение семафора больше нуля, функция уменьшает значение семафора sem на единицу. Если текущее значение семафора равно нулю, выполнение потока, вызвавшего функцию sem_wait, приостанавливается до тех пор, когда значение семафора станет положительным, тогда значение семафора уменьшается на единицу и поток продолжает работу.

```
int sem_trywait (sem_t *sem)
```

Если текущее значение семафора больше нуля, функция уменьшает значение семафора sem на единицу и возвращает ноль. Если текущее значение семафора равно нулю, функция возвращает не нулевое значение. Функция sem_trywait не останавливает работу потока.

Обратите внимание!

Для корректной работы описанных семафоров необходимо при компиляции программы использовать команду:

```
g++ <исходный файл> -o <исполняемый файл> -lpthread.
```

Программа 26. Пример использования семафора для синхронизации потоков

Перепишем пример использования потоков из предыдущей главы, введя семафор для защиты глобальной переменной SUM и семафор, блокирующий процесс-родитель, пока не выполнятся все его дочерние потоки. Изменения выделены жирным шрифтом.

```
#include <sched.h>
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <iostream>
using namespace std;
#define NUMSTACK 5000 /* Объем стека для отдельного потока. */
sem_t sem ; /* Семафор для защиты критической секции. */
sem_t sem4 ; /* Семафор для синхронизации родителя и потомков. */
int A[100] =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24} ;
/* Массив, сумма элементов которого вычисляется потоками. */
int SUM=0; /* Для записи общей суммы. */
char stack[4][ NUMSTACK]; /* Для хранения стека четырех потоков.
*/
int func(void *param) /* Стартовая функция потоков. */
{
     int i, sum = 0; /* Для суммирования элементов. */
     /* Индекс массива, с которого начинается суммирование. */
     int p =*(int *)param;
     p = p * 25;
     for(i=p; i<p+25; i++)
          sum+=A[i] ; /* Вычисление суммы части элементов
```

```
          массива. */
     sem_wait(&sem); /* Отнять единицу от значения семафора sem. */
     SUM+=sum; /* Добавление вычисленного результата в общую
     переменную. */
     sem_post(&sem); /* Добавить единицу к значения
     семафора sem. */
     sem_post(&sem4); /* Добавить единицу к значения
     семафора sem4. */
return 1;
}
int main()
{
     /* Тут должна быть инициализация элементов массива A. */
     sem_init (&sem, 1, 1); /* Инициализация семафора sem
     со значением 1. */
     sem_init (&sem4, 1, 0); /* Инициализация семафора sem4
     со значением 0. */
     int param[4]; /* Для хранения параметров потоков. */
     for (int i=0 ; i<4 ; i++) /* Создание четырех потоков. */
     {
          param[i]=i;
          char *tostack=stack[i];
          clone(func,(void*)( tostack+ NUMSTACK -1), CLONE_VM,
          (void *) (param+i));
     } /* Отнять четыре единицы от значения семафора sem4. */
     for (int i =0; i<4; i++)
          sem_wait(&sem4);
     cout<< «Результат = « <<SUM <<endl ;
return 1;
}
```

Принцип работы семафоров, использованных в данном примере, полностью аналогичен семафорам из двух предыдущих примеров — пример использования семафора для синхронизации процессов и пример использования семафора для защиты критической секции.

Удаление набора семафоров из системы с помощью команды ipcrm или системного вызова semctl()

Массив семафоров может продолжать существовать в системе и после завершения использовавших его процессов, а семафоры будут сохранять свое значение. Это может привести к некорректному поведению программ, предполагающих, что семафоры были только что созданы и, следовательно, имеют нулевое значение. Необходимо удалять семафоры из системы перед запуском таких программ или перед их завершением. Для удаления семафоров можно воспользоваться командами ipcs и ipcrm, рассмотренными в материалах предыдущего семинара. Команда ipcrm в этом случае должна иметь вид

```
ipcrm sem <IPC идентификатор>
```

Для этой же цели можно применять системный вызов semctl().

Системный вызов semctl()

Прототип системного вызова

```
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, union semun arg);
```

Описание системного вызова

Системный вызов **semctl** предназначен для получения информации о массиве IPC-семафоров, изменения его атрибутов и удаления его из системы. Данное описание не является полным описанием системного вызова. Для изучения полного описания обращайтесь к UNIX Manual.

В данном курсе применяется системный вызов **semctl** только для удаления массива семафоров из системы. Параметр **semid** является дескриптором System V IPC для массива семафоров, т. е. значением, которое вернул системный вызов **semget()** при создании массива или при его поиске по ключу.

В качестве параметра **cmd** в рамках нашего курса мы всегда будем передавать значение **IPC_RMID** — команду для удаления сегмента разделяемой памяти с заданным идентификатором. Параметры semnum и arg для этой команды не используются, поэтому мы всегда будем подставлять вместо них значение 0.

Если какие-либо процессы находились в состоянии «ожидание» для семафоров из удаляемого массива при выполнении системного вызова **semop()**, то они будут разблокированы и вернутся из вызова **semop()** с индикацией ошибки.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Понятие о POSIX-семафорах

В стандарте POSIX вводятся другие семафоры, полностью аналогичные семафорам Дейкстры. Для инициализации значения таких семафоров применяется функция sem_init(), аналогом операции P служит функция sem_wait(), а аналогом операции V — функция sem_post(). К сожалению, в Linux такие семафоры реализованы только для нитей исполнения одного процесса.

Контрольные вопросы

1. Семафоры в UNIX. Отличие операций над UNIX-семафорами от классических операций.
2. Создание массива семафоров или доступ к уже существующему. Системный вызов semget().
3. Семафоры для синхронизации процессов. Использование семафора для защиты критической секции.
4. Удаление набора семафоров из системы с помощью команды ipcrm или системного вызова semctl().
5. Понятие о POSIX-семафорах.

Очереди сообщений в UNIX как составная часть System V IPC

Очередь сообщений представляет собой однонаправленный связанный список, расположенный в адресном пространстве ядра. Процессы могут записывать сообщения в очередь и изымать их из очереди. Само сообщение включает в себя тип сообщения — целое положительное число и непосредственно данные.

Так как очереди сообщений входят в состав средств System V IPC, для них верно все, что говорилось ранее об этих средствах в целом. Очереди сообщений, как и семафоры, и разделяемая память являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по окончании взаимодействия. Пространством имен очередей сообщений является то же самое множество значений ключа, генерируемых с помощью функции ftok(). Для выполнения примитивов send и receive соответствующим системным вызовам в качестве параметра передаются IPC-дескрипторы очередей сообщений, однозначно идентифицирующих их во всей вычислительной системе.

Очереди сообщений располагаются в адресном пространстве ядра операционной системы в виде однонаправленных списков и имеют ограничение по объему информации, хранящейся в каждой очереди. Каждый элемент списка представляет собой отдельное сообщение. Сообщения имеют атрибут, называемый типом сообщения.

Выборка сообщений из очереди (выполнение примитива receive) может осуществляться тремя способами:

– в порядке FIFO, независимо от типа сообщения;

– в порядке FIFO для сообщений конкретного типа;

– первым выбирается сообщение с минимальным типом, не превышающим некоторого заданного значения, пришедшее раньше других сообщений с тем же типом.

Реализация примитивов send и receive обеспечивает скрытое от пользователя взаимоисключение во время помещения сообщения в очередь или его получения из очереди. Также она обеспечивает блокировку процесса при попытке выполнить примитив receive над пустой очередью или очередью, в которой отсутствуют сообщения запрошенного типа, или при попытке выполнить примитив send для очереди, в которой нет свободного места.

Очереди сообщений, как и другие средства System V IPC, позволяют организовать взаимодействие процессов, не находящихся одновременно в вычислительной системе.

Создание очереди сообщений или доступ к уже существующей. Системный вызов msgget()

Для создания очереди сообщений, ассоциированной с определенным ключом, или для доступа по ключу к уже существующей очереди используется

системный вызов msgget(), являющийся аналогом системных вызовов shmget() для разделяемой памяти и semget() для массива семафоров, который возвращает значение IPC-дескриптора для этой очереди. При этом существуют те же способы создания и доступа, что и для разделяемой памяти или семафоров.

Системный вызов msgget()

Прототип системного вызова

```
#include <types.h>
#include <ipc.h>
#include <msg.h>
int msgget(key_t key, int msgflg);
```

Описание системного вызова

Системный вызов **msgget** предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор System V IPC для этой очереди (целое неотрицательное число, однозначно характеризующее очередь сообщений внутри вычислительной системы и использующееся в дальнейшем для других операций с ней).

Параметр **key** является ключом System V IPC для очереди сообщений, т. е. фактически ее именем из пространства имен System V IPC. В качестве значения этого параметра может быть использовано значение ключа, полученное с помощью функции **ftok()**, или специальное значение **IPC_PRIVATE**. Использование значения **IPC_PRIVATE** всегда приводит к попытке создания новой очереди сообщений с ключом, который не совпадает со значением ключа ни одной из уже существующих очередей и не может быть получен с помощью функции **ftok()** ни при одной комбинации ее параметров.

Параметр **msgflg** — флаги — играет роль только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди, а также необходимость создания новой очереди и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или — « | ») следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT — если очереди для указанного ключа не существует, она должна быть создана;

IPC_EXCL — применяется совместно с флагом **IPC_CREAT**. При совместном их использовании и существовании массива с указанным ключом доступ к очереди не производится и констатируется ошибочная ситуация, при этом переменная errno, описанная в файле <**errno.h**>, примет значение **EEXIST**;

0400 — разрешено чтение для пользователя, создавшего очередь;

0200 — разрешена запись для пользователя, создавшего очередь;

0040 — разрешено чтение для группы пользователя, создавшего очередь;

0020 — разрешена запись для группы пользователя, создавшего очередь;

0004 — разрешено чтение для всех остальных пользователей;

0002 — разрешена запись для всех остальных пользователей.

Очередь сообщений имеет ограничение по общему количеству хранимой информации, которое может быть изменено администратором системы. Текущее значение ограничения можно узнать с помощью команды **ipcs -l**

Возвращаемое значение

Системный вызов возвращает значение дескриптора System V IPC для очереди сообщений при нормальном завершении и значение –1 при возникновении ошибки.

Реализация примитивов send и receive. Системные вызовы msgsnd() и msgrcv()

Для выполнения примитива send используется системный вызов msgsnd(), копирующий пользовательское сообщение в очередь сообщений, заданную IPC-дескриптором. При изучении описания этого вызова нужно обратить особое внимание на следующие моменты.

Тип данных struct msgbuf не является типом данных для пользовательских сообщений, а представляет собой лишь шаблон для создания таких типов. Пользователь сам должен создать структуру для своих сообщений, в которой первым полем должна быть переменная типа long, содержащая положительное значение типа сообщения.

В качестве третьего параметра — длины сообщения — указывается не вся длина структуры данных, соответствующей сообщению, а только длина полезной информации, т. е. информации, располагающейся в структуре данных после типа сообщения. Это значение может быть и равным 0 в случае, когда вся полезная информация заключается в самом факте прихода сообщения (сообщение используется как сигнальное средство связи).

В материалах семинаров мы, как правило, будем использовать нулевое значение флага системного вызова, которое приводит к блокировке процесса при отсутствии свободного места в очереди сообщений.

Системный вызов msgsnd()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *ptr,
int length, int flag);
```

Описание системного вызова

Системный вызов **msgsnd** предназначен для помещения сообщения в очередь сообщений, т. е. является реализацией примитива send.

Параметр **msqid** является дескриптором System V IPC для очереди, в которую отправляется сообщение, т. е. значением, которое вернул системный вызов **msgget()** при создании очереди или при ее поиске по ключу.

Структура **struct msgbuf** описана в файле <**sys/msg.h**> как

```
struct msgbuf {
long mtype;
char mtext[1];};
```

Она представляет собой некоторый шаблон структуры сообщения пользователя. Сообщение пользователя — это структура, первый элемент которой обязательно имеет тип long и содержит тип сообщения, а далее следует информативная часть теоретически произвольной длины (практически в Linux она ограничена размером 4080 байт и может быть еще уменьшена системным администратором), содержащая собственно суть сообщения.

Например:

```
struct mymsgbuf {
long mtype;
char mtext[1024];
} mybuf;
```

При этом информация вовсе не обязана быть текстовой, *например*:

```
struct mymsgbuf {
long mtype;
struct {
int iinfo;
float finfo;
} info;
} mybuf;
```

Тип сообщения должен быть строго положительным числом. Действительная длина полезной части информации (т. е. информации, расположенной в структуре после типа сообщения) должна быть передана системному вызову в качестве параметра **length**. Этот параметр может быть равен и 0, если вся полезная информация заключается в самом факте наличия сообщения. Системный вызов копирует сообщение, расположенное по адресу, на который указывает параметр ptr, в очередь сообщений, заданную дескриптором **msqid**.

Параметр **flag** может принимать два значения: 0 и **IPC_NOWAIT**. Если значение флага равно 0, и в очереди не хватает места для того, чтобы поместить сообщение, то системный вызов блокируется до тех пор, пока не освободится место. При значении флага **IPC_NOWAIT** системный вызов в этой ситуации не блокируется, а констатирует возникновение ошибки с установлением значения переменной errno, описанной в файле <**errno.h**>, равным **EAGAIN**.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Примитив **receive** реализуется системным вызовом **msgrcv()**. При изучении описания этого вызова нужно обратить особое внимание на следующие моменты.

Тип данных **struct msgbuf**, как и для вызова **msgsnd()**, является лишь шаблоном для пользовательского типа данных.

Способ выбора сообщения задается нулевым, положительным или отрицательным значением параметра **type**. Точное значение типа выбранного сообщения можно определить из соответствующего поля структуры, в которую системный вызов скопирует сообщение.

Системный вызов возвращает длину только полезной части скопированной информации, т. е. информации, расположенной в структуре после поля типа сообщения.

Выбранное сообщение удаляется из очереди сообщений.

В качестве параметра **length** указывается максимальная длина полезной части информации, которая может быть размещена в структуре, адресованной параметром ptr.

В материалах семинаров мы будем, как правило, пользоваться нулевым значением флагов для системного вызова, которое приводит к блокировке процесса в случае отсутствия в очереди сообщений с запрошенным типом и к ошибочной ситуации в случае, когда длина информативной части выбранного сообщения превышает длину, специфицированную в параметре **length**.

Системный вызов msgrcv()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msqid, struct msgbuf *ptr,
int length, long type, int flag);
```

Описание системного вызова

Системный вызов **msgrcv** предназначен для получения сообщения из очереди сообщений, т. е. является реализацией примитива **receive**.

Параметр **msqid** является дескриптором System V IPC для очереди, из которой должно быть получено сообщение, т. е. значением, которое вернул системный вызов **msgget()** при создании очереди или при ее поиске по ключу.

Параметр **type** определяет способ выборки сообщения из очереди следующим образом.

Таблица 4

Способ выборки сообщения из очереди

Способ выборки	Значение параметра type
В порядке FIFO, независимо от типа сообщения	0
В порядке FIFO для сообщений с типом n	n
Первым выбирается сообщение с минимальным типом, не превышающим значения n, пришедшее ранее всех других сообщений с тем же типом	-n

Максимально возможная длина информативной части сообщения в операционной системе Linux составляет 4080 байт и может быть уменьшена при генерации системы. Текущее значение максимальной длины можно определить с помощью команды

```
ipcs -l
```

Удаление очереди сообщений из системы с помощью команды ipcrm или системного вызова msgctl()

После завершения процессов, использовавших очередь сообщений, она не удаляется из системы автоматически, а продолжает сохраняться в системе вме-

сте со всеми невостребованными сообщениями до тех пор, пока не будет выполнена специальная команда или специальный системный вызов. Для удаления очереди сообщений можно воспользоваться командой ipcrm, которая в этом случае примет вид

```
ipcrm msg <IPC идентификатор>
```

Если какой-либо процесс находился в состоянии «ожидание» при выполнении системного вызова msgrcv() или msgsnd() для удаляемой очереди, то он будет разблокирован, и системный вызов констатирует наличие ошибочной ситуации.

Можно удалить очередь сообщений и с помощью системного вызова msgctl().

Системный вызов msgctl()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd,
           struct msqid_ds *buf);
```

Описание системного вызова

Системный вызов **msgctl** предназначен для получения информации об очереди сообщений, изменения ее атрибутов и удаления из системы. Данное описание не является полным описанием системного вызова. Для изучения полного описания обращайтесь к UNIX Manual.

В нашем курсе мы будем пользоваться системным вызовом **msgctl** только для удаления очереди сообщений из системы. Параметр **msqid** является дескриптором System V IPC для очереди сообщений, т. е. значением, которое вернул системный вызов **msgget()** при создании очереди или при ее поиске по ключу.

В качестве параметра **cmd** в рамках нашего курса мы всегда будем передавать значение **IPC_RMID** — команду для удаления очереди сообщений с заданным идентификатором. Параметр **buf** для этой команды не используется, поэтому мы всегда будем подставлять туда значение **NULL**.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение –1 при возникновении ошибки.

Пример использования очереди сообщений

Процесс-родитель создает очередь сообщений и порождает три дочерних процесса, процесс-родитель и его потомки вычисляют сумму элементов определенной части массива, процессы-потомки записывают вычисленную сумму в очередь сообщений. Родительский процесс дожидается поступления трех сообщений и выводит на экран окончательный результат — сумму всех элементов массива.

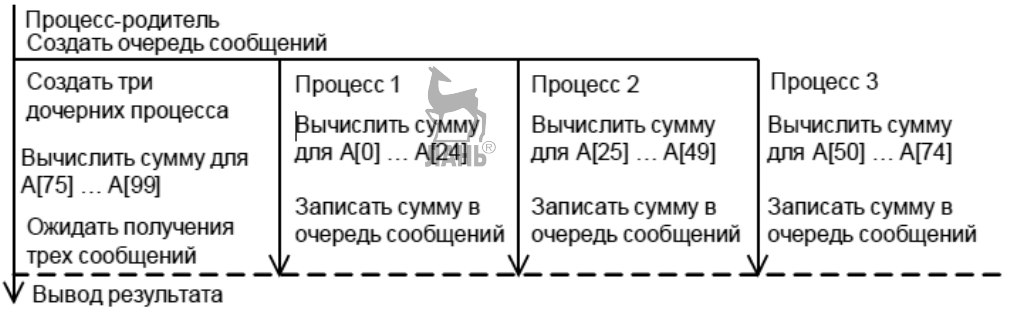


Рис. 18
Схема процессов

Программа 27. Пример использования очереди сообщений

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <iostream>
using namespace std;
int msgid; /* Для хранения дескриптора очереди сообщений. */
int A[100] =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24} ;
/* Массив, сумма элементов которого вычисляется процессами. */
struct mymsg /* Структура для сообщений. */
{
     int mtype; /* Тип сообщения. */
     int mdata; /* Данные сообщения. */
}
m;
int summa (int p) /* Для вычисления суммы части элементов
массива. */
{
     int i, sum = 0 ; /* Для суммирования элементов. */
     int begin =25*p; /* Индекс массива, с которого начинается
     суммирование. */
     int end = begin+25; /* Индекс массива, на котором завершается
     суммирование. */
     for(i=begin; i<end; i++) sum+=A[i]; /* Вычисление суммы части
     элементов массива. */
     m.mtype = 1; /* Установить тип сообщения в 1. */
     m.mdata=sum; /* Записать вычисленную сумму в сообщение. */
     msgsnd(msgid, &m, 2, 0); /* Послать сообщение в очередь,
     объем 2 байта. */
return sum ; /* Возвратить вычисленную сумму. */
}
int main()
{
     /* Тут должна быть инициализация элементов массива A. */
     /* Создать очередь сообщений. */
     msgid = msgget(IPC_PRIVATE, IPC_CREAT|0666);
     /* Если не удалось создать очередь сообщений, завершить
```

```
     выполнение. */
     if (msgid < 0 )
     {
          cout<<<«Ошибка»<<endl;
     return 0;
     }
     for (int i=0 ; i<3 ; i++) /* Создать три процесса-потомка. */
     {
          if (fork() == 0)
          /* Истинно для дочернего процесса. */
          {
               summa(i);
          return 1;
          }
     } /* Родительский процесс вычисляет последнюю четверть
     массива. */
     int rez = summa(3);
     for (int i=0 ; i<3 ; i++) /* Дождаться получения трех
     сообщений. */
     {
          msgrcv(msgid, &m, 2, 0, 0);
/* Тип получаемого сообщения не важен. */
          rez += m.mdata; /* Добавить данные сообщения
          к результату. */
     } /* Вывести на экран сумму всех элементов массива. */
     cout<<<«Сумма = «<< rez<<endl;
return 1;
}
```

Программа 28a. Пример с однонаправленной передачей текстовой информации

Эта программа получает доступ к очереди сообщений, отправляет в нее 5 текстовых сообщений с типом 1 и одно пустое сообщение с типом 255, которое будет служить для программы 7-02b сигналом прекращения работы.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#include <iostream>
using namespace std;
#define LAST_MESSAGE 255 /* Тип сообщения для прекращения работы
программы 2b. */
int main()
{
     int msqid; /* IPC дескриптор для очереди сообщений. */
     char pathname[] = «Имя файла»; /* Имя файла, использующееся
     для генерации ключа. Файл с таким именем должен существовать
     в текущей директории. */
     key_t key; /* IPC ключ. */
     int i, len; /* Счетчик цикла и длина информативной части
     сообщения.*/
     /* Ниже следует пользовательская структура для сообщения. */
```

```
    struct mymsgbuf
    {
        long mtype;
        char mtext[81];
    }
    mybuf;
    /* Генерирование IPC ключа из имени файла 2a в текущей
    директории и номера экземпляра очереди сообщений 0. */
    if((key = ftok(pathname,0)) < 0)
    {
        cout<<«Can't generate key»<<endl;
        exit(-1);
    } /* Попытка получить доступ по ключу к
    очереди сообщений, если она существует, или
    создать ее с правами доступа read & write для всех
    пользователей. */
    if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0)
    {
        cout<<«Can't get msqid «<< endl;
        exit(-1);
    }
    /* Посылка в цикле 5 сообщений с типом 1 в очередь сообщений,
    идентифицируемую msqid.*/
    for (i = 1; i <= 5; i++)
    { /* Сначала заполнение структуры для сообщения и
    определение длины информативной части.
    */
        mybuf.mtype = 1;
        strcpy(mybuf.mtext, «This is text message»);
        len = strlen(mybuf.mtext)+1;
        /* Отправка сообщения. В случае ошибки сообщение об этом
        и удаление очереди сообщений из системы. */
        if (msgsnd(msqid, (struct msgbuf *) &mybuf, len, 0) < 0)
        {
            cout<<«Can't send message to queue»<< endl;
            msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
            exit(-1);
        }
    }
    /* Отправка сообщения, которое заставит получающий процесс
    прекратить работу, с типом LAST_MESSAGE и длиной 0. */
    mybuf.mtype = LAST_MESSAGE;
    len = 0;
    if (msgsnd(msqid, (struct msgbuf *) &mybuf, len, 0) < 0)
    {
        cout<<«Can't send message to queue «<< endl;
        msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
        exit(-1);
    }
return 0;
}
```

Программа 28b. Пример использования очереди сообщений

Эта программа получает доступ к очереди сообщений и читает из нее сообщения с любым типом в порядке FIFO до тех пор, пока не получит сообщение с типом 255, которое будет служить сигналом прекращения работы.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#include <iostream>
using namespace std;
#define LAST_MESSAGE 255 /* Тип сообщения для прекращения работы.
*/
int main()
{
     int msqid; /* IPC дескриптор для очереди сообщений. */
     char pathname[] = «Имя файла»; /* Имя файла, использующееся
     для генерации ключа. Файл с таким именем должен существовать
     в текущей директории. */
     key_t key; /* IPC ключ. */
     int len, maxlen; /* Реальная длина и максимальная длина
     информативной части сообщения. */
     /* Ниже следует пользовательская структура для сообщения. */
     struct mymsgbuf
     {
          long mtype;
          char mtext[81];
     }
     mybuf;
     /* Генерирование IPC ключа из имени файла 2a в текущей
     директории и номера экземпляра очереди сообщений 0. */
     if((key = ftok(pathname,0)) < 0)
     {
          cout<<<«Can't generate key»<<endl;
          exit(-1);
     }
     /* Попытка получить доступ по ключу к очереди сообщений, если
     она существует, или создать ее с правами доступа read & write
     для всех пользователей. */
     if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0)
     {
          cout<<<«Can't get msqid»<<endl;
          exit(-1);
     }
     while(1)
          { /* В бесконечном цикле принятие сообщения любого типа в
          порядке FIFO с максимальной длиной информативной части 81
          символ до тех пор, пока не поступит сообщение
          с типом LAST_MESSAGE.*/
          maxlen = 81;
          if(len = msgrcv(msqid, (struct msgbuf *) &mybuf,
          maxlen, 0, 0) < 0)
```

```
            {
                 cout<<«Can't receive message from queue»<<endl;
                 exit(-1);
            }
            /*Если принятое сообщение имеет тип LAST_MESSAGE,
            прекращение работы и удаление очереди сообщений из
            системы. В противном случае вывод принятого сообщения. */
            if (mybuf.mtype == LAST_MESSAGE)
            {
                 msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
                 exit(0);
            }
            cout<<«message type = « << mybuf.mtype<< «info = « <<
            mybuf.mtext<< endl;
      }
return 0; /* Исключительно для отсутствия warning'ов при компиля-
ции. */
}
```

Первая из этих программ посылает пять текстовых сообщений с типом 1 и одно сообщение нулевой длины с типом 255 второй программе. Вторая программа в цикле принимает сообщения любого типа в порядке FIFO и печатает их содержимое до тех пор, пока не получит сообщение с типом 255. Сообщение с типом 255 служит для нее сигналом к завершению работы и ликвидации очереди сообщений. Если перед запуском любой из программ очередь сообщений еще отсутствовала в системе, то программа создаст ее. Необходимо обратить внимание на использование сообщения с типом 255 в качестве сигнала прекращения работы второго процесса. Это сообщение имеет нулевую длину, так как его информативность исчерпывается самим фактом наличия сообщения.

Модификация предыдущего примера для передачи числовой информации. В описании системных вызовов msgsnd() и msgrcv() говорится о том, что передаваемая информация не обязательно должна представлять собой текст. Можно воспользоваться очередями сообщений для передачи данных любого вида. При передаче разнородной информации целесообразно информативную часть объединять внутри сообщения в отдельную структуру для правильного вычисления длины информативной части.

```
struct mymsgbuf {
    long mtype;
    struct {
        short sinfo;
        float finfo;
    } info;
} mybuf;
```

В некоторых вычислительных системах числовые данные размещаются в памяти с выравниванием на определенные адреса (например, на адреса, кратные 4).

Поэтому реальный размер памяти, необходимой для размещения нескольких числовых данных, может оказаться больше суммы длин этих данных, т. е. в нашем случае:

```
sizeof(info)>=sizeof(short)+sizeof(float)
```

Для полной передачи информативной части сообщения в качестве длины нужно указывать не сумму длин полей, а полную длину структуры.

ОБЗОР ИЕРАРХИИ И ВИДОВ ВНЕШНЕЙ ПАМЯТИ

Обычно, когда говорят о внешней памяти, представляют устройства, которые используются для долговременного хранения информации. А под внутренней памятью подразумевают оперативное запоминающее устройство (ОЗУ). В отличие от ОЗУ, внешняя память не может на прямую обратиться к процессору. Также внешняя память является, в отличие от ОЗУ, более объемной.

Основным значением внешней памяти является возможность долговременно хранить большое количество информации.

Устройства, обеспечивающие запись и считывание информации, называются накопителями, а обеспечивающие хранение информации — носителями.

Существует множество видов внешней памяти, такие как:

Магнитная лента. В свое время это был самый многофункциональный способ хранения информации.

Магнитная лента была разработана в 1930 г. в Германии при сотрудничестве двух крупных компаний: химического концерна BASF и электронной компании AEG при содействии немецкой компании телерадиовещательной компании RRG.

Принцип магнитной ленты до боли прост: в 1927 г. немецкий инженер Фриц Пфлеймер, после ряда экспериментов с различными материалами, сделал напыление порошком оксида железа на тонкую бумагу с помощью клея — это и есть весь секрет магнитной ленты.

Первой компанией, кто взял на вооружение эту идею, была компания AEG, занимавшаяся производством аудиозаписей.

Позже, в 1956 г., компания Ampex представила первый видеомагнитофон на основе работы магнитной ленты.

В 1967 г. Алан Шугарт возглавлял команду, разрабатывающую дисководы в лаборатории под названием IBM. Они создали накопители на гибких дисках. И Дэвид Нобл предложил идею гибкого диска и защитного кожуха с тканевой прокладкой. И уже в 1971 г. фирма IBM представила дискету диаметром 8 дюймов с дисководом.

Дискета представляет собой сменный носитель, который используется для многократной записи и хранения информации. Гибкий магнитный диск имеет функцию защиты от записи, с помощью которой можно предоставить доступ к данным только в режиме чтения.

В устройстве чтения/записи информации на флоппи-диске (гибкий диск, или просто дискета) имеются два двигателя: один позволяет сохранять постоянную скорость вращения вставленной в устройство дискеты, а второй двигает головки записи/чтения. Скорость вращения первого двигателя зависит только от типа дискеты и составляет от 300 до 360 оборотов в минуту. Двигатель для перестановки головок в этих приводах всегда шаговый. С его помощью головки двигаются по радиусу от края диска к его центру интервалами. В отличие от привода винчестера, головки в данном устройстве не летают над поверхностью дискеты, а соприкасаются с ней. Работой всех узлов привода управляет соответствующий контроллер.

В зависимости от модели менялся объем от 80 до 2880 кб.

В составе первого серийного компьютера накопитель занимал ящик огромного размера и имел вес 971 кг, а общий объем памяти составлял всего лишь 3,5 мб.

Первый носитель в виде *жесткого диска*, который использовался в качестве постоянного накопителя информации, имел незначительную емкость. Для нашего времени эти числа не велики, так как современные жесткие диски имеют емкость до 8 тб.

Жесткий диск имеет неплохую скорость передачи данных при последовательном доступе (во внутренней зоне — 44,2–74,5 мб/с, в зависимости от модели, и во внешней зоне диска — 60–111,4 мб/с, и также в зависимости от модели).

Жесткий диск представляет собой коробку, в которой находятся металлические диски, которые покрыты магнитным материалом (плоттером). Они соединены при помощи шпинделя.

При записи информации используется от 4 до 9 пластин (дисков). Шпиндель крутится с очень высокой и постоянной скоростью (от 4200 до 7200 оборотов в минуту).

Информация записывается/считывается с помощью специальных головок записи или чтения по одной на все поверхности пластины. Количество головок, в любом случае, должно быть равно рабочим поверхностям всех пластин. Запись на пластины производится по специальным дорожкам, они делятся на определенные сектора. В каждом из них содержится 512 байт информации.

К списку преимуществ перед другими можно отнести большой объем хранимой информации, высокую скорость работы, дешевизна хранения данных.

Недостатки: большие габариты, чувствительность к вибрации, тепловыделение.

Самый первый *компакт-диск* был разработан в 1979 г. и использовался только для хранения аудиозаписей. В 1970 г. инженеры двух компаний Philips и Sony начали работу над ALP-дисками, которые должны были сместить с рынка виниловые пластинки. Диаметр этих дисков был примерно 30 см. Позже диск был уменьшен в размерах.

CD-ROM устройства представляют собой диски с записанной на них информацией, доступной только лишь для чтения.

CD-ROM является, фактически, более доработанной версией CD-DA (CD-DA представляет собой диски для хранения аудиозаписей), но в отличие от них на CD-ROM можно хранить прочие данные.

Позже разработаны версии CD-ROM, позволяющие вести одноразовую и многоразовую записи (CD-R и CD-RW соответственно).

CD-ROM имеет не самую большую емкость, всего лишь 879 мб (и это максимальное значение данного вида внешней памяти).

Скорость чтения у CD-ROM довольно-таки низкая — всего лишь 150 кб/с.

Принцип работы CD-ROM напоминает принцип работы обычных дисководов для гибких дисков. Считывание информации с компакт-диска так же, как

и запись, происходит при помощи лазерного луча, но, конечно же, менее мощного, как в случае с дисководом для гибких дисков. Покрытие оптического диска (CD-ROM) двигается относительно лазерной головки с постоянной линейной скоростью, а угловая скорость меняется в зависимости от угла наклона головки. Так что, чтение внутренних полос осуществляется с увеличенным, а внешних — с уменьшенным числом оборотов. Благодаря этому характеризуется очень низкая скорость доступа к данным для компакт-дисков по сравнению, например, с винчестерами (жесткими дисками) или более современной модели компакт дисков DVD-ROM. Сервомотор по команде от внутреннего микропроцессора привода двигает отражающее зеркало. Это помогает точно поставить лазерный луч на определенную дорожку. Луч проходит через защитный слой пластика и попадает на поверхность диска. При попадании его на горку, он будет отражен на детектор и после чего проходит через призму, отклоняющую его на светочувствительный диод. Если луч попадает в яму, он рассеивается, и только малая часть излучения попадает обратно и проходит до светочувствительного диода. На диоде световые импульсы преобразуются в электрические; яркое излучение преобразуется в единицы, слабое — в нули.

Компакт-диск, он же CD-ROM, представляет собой поликарбонатный диск толщиной 1,2 мм, покрытый слоем металла и лаком для защиты, на котором, обычно, изображено графическое представление содержащейся на диске информации.

Преимущества компакт-диска состоят в следующем: удобство транспортировки, возможность тиражирования, дешевизна хранения информации.

Недостатки состоят из следующих факторов: небольшой объем, нужно считывающее устройство, ограничения при операциях, невысокая скорость работы, чувствительность к вибрации.

DVD-накопитель по форме напоминает CD-накопитель, но сравнивая их по структуре, можно заметить, что DVD имеет более плотную структуру рабочей поверхности, это позволяет хранить/считывать больший объем информации за счет использования лазера с меньшей длиной волны и линзы с большей числовой апертурой.

DVD имеет множество форм в зависимости от того, какое количество сторон и слоев. Варьируются размеры емкости от 1,4 гб до 17,08 гб.

Скорость чтения у DVD-накопителя имеет неплохой результат — 10,5 мб.

Flash-память является энергонезависимым типом памяти, который позволяет записывать/хранить данные в микросхемах. Карты flash-памяти не имеют в своем составе движущихся частей, и это обеспечивает высокую сохранность данных при их использовании в мобильных устройствах (портативных компьютерах, цифровых камерах и др.).

Flash-память представляет собой микросхему, помещенную в пластмассовый плоский корпус. Для считывания или записи информации карта памяти вставляется в специальные накопители, имеющие различные формы и модификации, встроенные в мобильные устройства или подключаемые к компьютеру через USB-порт. Информационная емкость карт памяти может достигать до 64 гбайт.

К недостаткам flash-памяти следует отнести то, что не имеется одного общепринятого стандарта, и практически все производители производят несовместимые друг с другом виды по техническим характеристикам.

Все эти виды внешней памяти можно привести в одну иерархическую систему (рис. 19), в которой можно заметить, что все эти устройства не имеют прямого доступа к процессору, а сначала проходят через главную память (оперативную), проходя через быстродействующую, память поступает в кэш процессора и ждет там свою очередь.

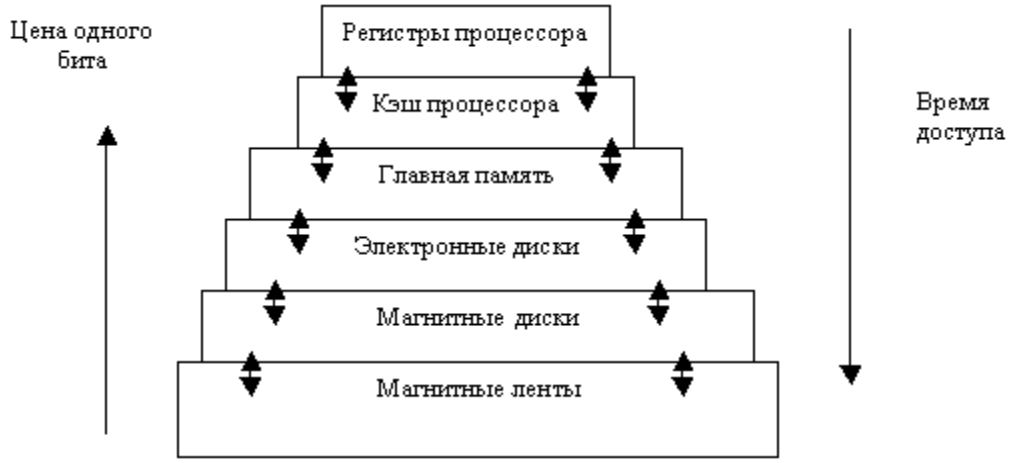


Рис. 19
Иерархическая система памяти

Идея иерархической организации памяти сводится к использованию на едином компьютере нескольких уровней памяти, характеризующихся различным временем доступа к памяти и ее объемом. Основным принципом организации памяти по способу иерархии служит принцип локальности адресов во времени и в пространстве. Локальность во времени представляет собой то, что процессор многократно работает с одними и теми же командами и данными. Локальность в пространстве представляет собой то, что если программе обязателен доступ к слову с адресом B, то, вероятнее всего, следующие ссылки будут к адресам, расположенным в максимальной близости с адресом B. Из свойства локальности ссылок выходит, что в типичном вычислении обращения к памяти сосредотачиваются вокруг определенной области адресного множества и, более того, выборка происходит последовательным путем адреса. Время доступа к иерархически организованной памяти уменьшается из-за следующего сокращения количества обращений к оперативной памяти, совмещению обработки текущего фрагмента программы и пересылки данных из основной памяти в буферную память.

Обзор файловой системы NTFS и сравнение с другими системами

Чтобы во всем разобраться, пойдем издалека. Что же такое *файловая система*? Кто-то может и не знать что такое файл. **Файл (file — папка)** — это именованная совокупность любых данных, размещенная на внешнем запоминающем устройстве, хранимая, обрабатываемая, пересылаемая, как единое целое, которая также может содержать числовые данные, изображение, текст, программу. А файловая система — это средство для организации хранения файлов на каком-либо носителе. Файлы реализуются как участки памяти на внешних носителях — CD-ROM, магнитных дисках, flash-карты и др. Каждый файл занимает определенное количество блоков памяти, обычно длина блока равна 512 байт.

С определением файла более-менее разобрались, теперь перейдем к файловой системе. Обслуживает файлы специальный модуль операционной системы — *драйвер файловой системы*. Каждый файл имеет имя, зарегистрированное в каталоге — оглавлении файлов. Каталог доступен пользователю, его можно просматривать, переносить находимые в нем файлы. Каталог может иметь собственное имя и храниться в другом каталоге, и так образуются иерархические файловые структуры.

Ранее нам встретился такой термин, как «драйвер файловой системы». Что же он значит и как работает этот драйвер? **Драйвер файловой системы** обеспечивает доступ к информации на каком-либо физическом носителе или магнитном диске, распределяет пространство на этом носителе между файлами. Например, что происходит, когда пользователь подал команду «открыть файл», в котором указано имя файла и каталога, где размещен этот файл. Драйвер файловой системы обращается к своему справочнику, ищет, какие блоки диска соответствуют указанному файлу, и подает запрос на считывание этих блоков драйверу диска. Поэтому для выполнения разных функций драйвер файловой системы хранит на диске не только пользовательскую информацию, но и свою служебную информацию. В этих областях диска хранится список всех каталогов и файлов и различные дополнительные справочные таблицы для повышения скорости работы этого драйвера. Структура хранения данных и структура файловой системы определяет удобство работы пользователя, скорость доступа к файлам.

Файловые системы в настоящее время разнообразны, каждая операционная система имела свою файловую систему, например, Btrfs на базе операционной системы Linux от разработчика Oracle, JFS на базе той же операционной системы от другого разработчика — IBM и др. Если провести поиск различных файловых систем, можно обнаружить не менее 70.

Первоначально остановимся на файловых системах операционной системы Windows, так как именно эта операционная система в наше время является наиболее популярной, поэтому разберемся, какие же файловые системы поддерживает операционная система от Microsoft. Затем более подробно рассмотрим организацию файловой системы Unix.

Наиболее широко используемая и популярная файловая система является FAT, которая делится на FAT16 и FAT32. Файловая система FAT16 появилась еще до появления MS-DOS и поддерживается всеми операционными системами Microsoft. ***File Allocation Table (Таблица расположения файлов)***, так расшифровывается эта аббревиатура, максимальный размер жесткого диска или раздела не превышает 4096 Мб. В те времена 4-гигабайтные жесткие диски казались мечтой, так как роскошью были диски объемом 40 Мбайт, поэтому такого запаса хватало вполне. Раздел, отформатированный под файловую систему FAT16, разделялся на кластеры. Его размер зависит от размера тома и может быть от 512 байт до 64 Кбайт. Не рекомендуется использовать такую файловую систему на томах больше 511 Мбайт, потому что дисковое пространство будет использоваться очень неэффективно. В этой файловой системе кластеры могут иметь разные значения, это может быть кластер, занятый файлом, свободный кластер, последний кластер файла или дефектный кластер. Различием между корневым и другими каталогами является то, что корневой располагается в определенном месте и имеет фиксированное число вхождений. Если число фиксированных вхождений для корневого каталога равно 1024 и создано 200 подкаталогов, в корневом каталоге можно создать не более 824 файлов. В структуре каталогов файлу отводится первый незанятый кластер. Номер начального кластера позволяет определить местонахождение файла: каждый кластер содержит указатель на следующий кластер или значение FFFF, указывающее на то, что это последний кластер в цепочке кластеров, занимаемых файлом. Так как все вхождения имеют одинаковый размер информационного блока, они различаются по байту атрибутов. Один из битов в данном байте может указывать, что это каталог, другой — что это метка тома. Для пользователей доступны четыре бита, позволяющих управлять атрибутами файла — архивный (archive), системный (system), скрытый (hidden) и доступный только для чтения (read-only).

Преимущества:

- Поддерживается старенькими операционными системами Windows 95, Windows 98 и некоторыми операционными системами UNIX.
- Разработано много программ для исправления ошибок в файловой системе и восстановления данных.
- Эффективна для томов объемом менее 256 Мбайт.

Недостатки:

- Корневой каталог не может содержать более 512 элементов.
- Поддерживает не более 65 535 кластеров.
- Нет поддержки резервной копии загрузочного сектора.
- Нет поддержки защиты файлов и сжатия.

С файловой системой FAT16 мы познакомились, теперь перейдем к следующей файловой системе, познакомимся с ней и узнаем, чем же она отличается от предшествующей. Как уже видно из названия файловой системы, появилась поддержка 32-битной FAT. Впервые эта файловая система стала поддерживаться в Windows 2000. Основным отличием является то, что объем тома увеличился в несколько раз, а именно с 4 Гбайт до 2 Тбайт, что считают это огромным плюсом. Также размер кластера может изменяться от 512 байт до

32 Кбайт. Теперь для хранения значений надо 4 байт или 32 бит, а в FAT16 — 16 бит, что значит, что некоторые файлы, рассчитанные на FAT16, не могут работать с FAT32. Ну и самое главное отличие этих двух файловых систем это то, что изменился размер логического диска. Более поздняя файловая система поддерживает тома объемом до 127 Гбайт. В FAT32 кластер размером в 4 Кбайт подходит для носителей объемом от 512 Мбайт до 8 Гбайт, а вот в FAT16, если использовать носитель с объемом в 2 гигабайта, требовался кластер размером 32 Кбайт. Из этого можно сделать вывод о том, что дисковое пространство можно использовать наиболее эффективно. Поэтому чем меньше кластер, тем меньше места требуется для хранения. Далее расскажем вам об одном недостатке данной файловой системы, с которой пришлось столкнуться. При использовании этой файловой системы максимальный размер файла может достигать 4 Гбайт. Сейчас, когда жесткие диски, внешние физические носители имеют объем 256 Гб–1 Тб, размер файла может достигать от 1 Гб и более. На внешнем физическом носителе была именно эта файловая система и было невозможно записать файл, объемом более 4 Гбайт. Чтобы решить эту проблему, пришлось отформатировать этот физический носитель до другой файловой системы. В сравнении с FAT16 увеличилось максимальное число вхождений в корневой каталог с 512 до 65 535, что, на наш взгляд, является огромным плюсом. Ведь теперь пользователь мог создать для себя такую файловую структуру, чтобы ему было удобно, комфортно работать, и мог создавать огромное количество каталогов и файлов. FAT32 накладывает ограничения на минимальный размер тома, не менее 65 527 кластеров.

Преимущества:

- Более эффективное выделение дискового пространства.
- Есть возможность перемещения корневого каталога, использование резервной копии.
- На 10–15% меньше занято дисковое пространство, чем в FAT16.

Недостатки:

- 32 Гб — максимальный размер тома под Windows 2000.
- Недоступность томов из других операционных систем.
- Нет поддержки резервной копии загрузочного сектора.
- Нет поддержки защиты файлов и сжатия.

Файловоя система NTFS, которая наиболее часто используется в настоящее время. Итак, файловая система NTFS (New Technology File System) разрабатывалась компанией Microsoft с 1990-х гг. как основная файловая система для серверных версий операционных систем Windows. Выпущена в 1993 г. в операционной системе Windows NT 3.1. Но если взглянуть на эту файловую систему в наше время, то можно заметить, что теперь она рассматривается и для клиентских версий Windows. В NTFS используются 64-разрядные идентификаторы кластеров. Для томов, больших 4 Гбайт, Windows устанавливает размер кластера 4 Кбайт. При решении проблемы, которая возникла при использовании файловой системы FAT32, физический носитель был отформатирован именно под файловую систему NTFS. Раздел NTFS может быть какого угодно размера, его максимальный размер может быть ограничен лишь размером самого физиче-

ского носителя. Как и все остальные файловые системы, NTFS делит все пространство на кластеры и поддерживает размеры от 512 байт до 64 Кбайт. Когда файловая система отформатирована в NTFS, создается файл Master File Table (MTF) и области для хранения метаданных. Они используются для реализации файловой структуры. Файл использует одну запись в MFT, но если у файла много атрибутов, то для хранения информации могут потребоваться дополнительные записи. С появлением файловой системы NTFS у нее появились **новые возможности**, которых нет у FAT:

- Возможность восстановления. Она встроена в NTFS. Данная файловая система гарантирует, что все данные будут сохранены за счет того, что использует протокол и алгоритмы восстановления данных. Если происходит сбой, целостность файловой системы автоматически восстанавливается.
- Сжатие данных. Это происходит так, что при чтении файл сразу используется и его не нужно предварительно распаковывать. При закрытии он также автоматически упаковывается.
- Исходя из операционной системы, некоторые ее функции требуют наличия именно этой файловой системы.
- Скорость доступа к файлам намного выше, так как NTFS старается использовать как можно меньшее число обращений к диску.
- Защита каталогов и файлов, то есть можно задать атрибуты доступа к файлам и папкам, для того, чтобы другие пользователи не могли изменять данные в файлах, если задан атрибут только для чтения.
- NTFS поддерживает резервную копию загрузочного сектора.
- Файловая система поддерживает систему шифрования EFS, что обеспечивает защиту от неавторизованного доступа к содержимому файлов.
- Можно ограничить объем дискового пространства, занимаемого пользователями.

Но и у этой файловой системы есть свои **недостатки**, например, ее тома недоступны в операционных системах Windows 95, Windows 98, MS-DOS, а также то, что если тома небольшого объема и содержат много файлов, то имеет место снижение производительности в сравнении с файловыми системами FAT.

Разделы носителя информации (partitions) в UNIX

Физические носители информации — магнитные или оптические диски, ленты и т. д., использующиеся как физическая основа для хранения файлов, в операционных системах принято логически делить на разделы (partitions) или логические диски. Причем слово «делить» не следует понимать буквально, в некоторых системах несколько физических дисков могут быть объединены в один раздел.

В операционной системе UNIX физический носитель информации обычно представляет собой один или несколько разделов. В большинстве случаев разбиение на разделы производится линейно, хотя некоторые варианты UNIX могут допускать некое подобие древовидного разбиения (Solaris). Количество разделов и их размеры определяются при форматировании диска.

Наличие нескольких разделов на диске может определяться требованиями операционной системы или пожеланиями пользователя. Если пользователь хочет разместить на одном жестком диске несколько операционных систем с возможностью попеременной работы в них, тогда он размещает каждую операционную систему в своем разделе. Или другая ситуация: необходимость работы с несколькими видами файловых систем. Под каждый тип файловой системы выделяется отдельный логический диск. Третий вариант — это разбиение диска на разделы для размещения в разных разделах различных категорий файлов.

Примером операционной системы, внутренние требования которой приводят к появлению нескольких разделов на диске, могут служить ранние версии MS-DOS, для которых максимальный размер логического диска не превышал 32 Мбайт.

Логическая структура файловой системы и типы файлов в UNIX

Файл — именованный абстрактный объект, обладающий определенными свойствами. При этом в пространстве имен файлов одному файлу могут соответствовать несколько имен.

В операционной системе UNIX существуют файлы нескольких типов, а именно:

– обычные или регулярные файлы;
– директории или каталоги;
– файлы типа FIFO или именованные pip'ы;
– специальные файлы устройств;
– сокеты (sockets);
– специальные файлы связи (link).

Файлы всех перечисленных типов логически объединены в ациклический граф с однонаправленными ребрами, получающийся из дерева в результате сращивания нескольких терминальных узлов дерева или нескольких его нетерминальных узлов таким образом, чтобы полученный граф не содержал циклов. В нетерминальных узлах такого ациклического графа (т. е. в узлах, из которых выходят ребра) могут располагаться только файлы типов «директория» и «связь». Причем из узла, в котором располагается файл типа «связь», может выходить только ровно одно ребро. В терминальных узлах этого ациклического графа (т. е. в узлах, из которых не выходит ребер) могут располагаться файлы любых типов (рис. 20), хотя присутствие в терминальном узле файла типа «связь» обычно говорит о некотором нарушении целостности файловой системы.

В отличие от древовидной структуры набора файлов, где имена файлов связывались с узлами дерева, в таком ациклическом графе имя файла связывается не с узлом, соответствующим файлу, а с входящим в него ребром. Ребра, выходящие из узлов, соответствующих файлам типа «связь», являются неименованными.

Надо отметить, что практически во всех существующих реализациях UNIX-подобных систем, в узел графа, соответствующий файлу типа «директо-

рия», не может входить более одного именованного ребра, хотя стандарт на операционную систему UNIX и не запрещает этого. В качестве полного имени файла может использоваться любое имя, получающееся при прохождении по ребрам от корневого узла графа (т. е. узла, в который не входит ни одно ребро) до узла, соответствующего этому файлу, по любому пути с помощью следующего алгоритма:

Если интересующему нас файлу соответствует корневой узел, то файл имеет имя «/».

Берем первое именованное ребро в пути и записываем его имя, которому предваряем символ «/».

Для каждого очередного именованного ребра в пути приписываем к уже получившейся строке справа символ «/» и имя соответствующего ребра.

Полное имя является уникальным для всей файловой системы и однозначно определяет соответствующий ему файл.

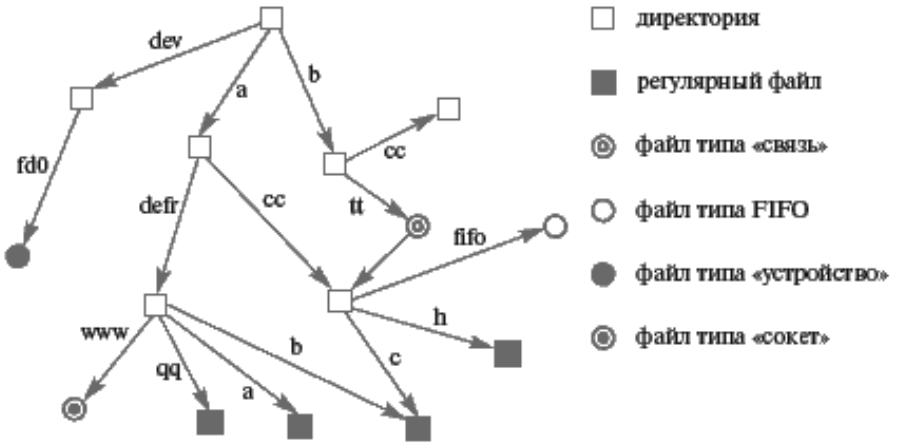


Рис. 20
Пример графа файловой системы

Организация файла на диске в UNIX на примере файловой системы s5fs. Понятие индексного узла (inode)

Все дисковое пространство раздела в файловой системе s5fs (System V file system) логически разделяется на две части: заголовок раздела и логические блоки данных. Заголовок раздела содержит служебную информацию, необходимую для работы файловой системы, и обычно располагается в самом начале раздела. Логические блоки хранят собственно содержательную информацию файлов и часть информации о размещении файлов на диске (т. е. какие логические блоки и в каком порядке содержат информацию, записанную в файл). Для размещения любого файла на диске используется метод индексных узлов (inode — от index node). Индексный узел содержит атрибуты файла и оставшуюся часть информации о его размещении на диске. Такие типы файлов, как «связь», «сокет», «устройство», «FIFO» не занимают на диске никакого иного

места, кроме индексного узла (им не выделяется логических блоков). Все необходимое для работы с этими типами файлов содержится в их атрибутах. Часть атрибутов файлов, хранящихся в индексном узле и свойственных большинству типов файлов:

- Тип файла и права различных категорий пользователей для доступа к нему.
- Идентификаторы владельца-пользователя и владельца-группы.
- Размер файла в байтах (только для регулярных файлов, директорий и файлов типа «связь»).
- Время последнего доступа к файлу.
- Время последней модификации файла.
- Время последней модификации самого индексного узла.

Количество индексных узлов в разделе является постоянной величиной, определяемой на этапе генерации файловой системы. Все индексные узлы системы организованы в виде массива, хранящегося в заголовке раздела. Каждому файлу соответствует только один элемент этого массива и, наоборот, каждому непустому элементу этого массива соответствует только один файл. Таким образом, каждый файл на диске может быть однозначно идентифицирован номером своего индексного узла (его индексом в массиве). На языке представления логической организации файловой системы в виде графа это означает, что каждому узлу графа соответствует только один номер индексного узла, и никакие два узла графа не могут иметь одинаковые номера. Надо отметить, что свойством уникальности номеров индексных узлов, идентифицирующих файлы, мы уже неявно пользовались при работе с именованными pip'ами и средствами System V IPC. Для именованного pip'а именно номер индексного узла, соответствующего файлу с типом FIFO, является той самой точкой привязки, пользуясь которой, неродственные процессы могут получить данные о расположении pip'а в адресном пространстве ядра и его состоянии и связаться друг с другом. Для средств System V IPC при генерации IPC-ключа с помощью функции **ftok()** в действительности используется не имя заданного файла, а номер соответствующего ему индексного дескриптора, который по определенному алгоритму объединяется с номером экземпляра средства связи.

Организация директорий (каталогов) в UNIX

Содержимое регулярных файлов (информация, находящаяся в них, и способ ее организации) всецело определяется программистом, создающим файл. В отличие от регулярных, остальные типы файлов, содержащих данные, т. е. директории и связи, имеют жестко заданную структуру и содержание, определяемые типом используемой файловой системы. Основным содержимым файлов типа «директория» являются имена файлов, лежащих непосредственно в этих директориях, и соответствующие им номера индексных узлов. В терминах представления в виде графа содержимое директорий представляет собой имена ребер, выходящих из узлов, соответствующих директориям, вместе с индексными номерами узлов, к которым они ведут.

В файловой системе s5fs пространство имен файлов (ребер) содержит имена длиной не более 14 символов, а максимальное количество inode в одном разделе файловой системы не может превышать значения 65 535. Эти ограничения не позволяют давать файлам осмысленные имена и приводят к необходимости разбиения больших жестких дисков на несколько разделов. Зато они помогают упростить структуру хранения информации в директории. Все содержимое директории представляет собой таблицу, в которой каждый элемент имеет фиксированный размер в 16 байт. Из них 14 байт отводится под имя соответствующего файла (ребра), а 2 байта — под номер его индексного узла. При этом первый элемент таблицы дополнительно содержит ссылку на саму данную директорию под именем « .», а второй элемент таблицы — ссылку на родительский каталог (если он существует), т. е. на узел графа, из которого выходит единственное именованное ребро, ведущее к текущему узлу, под именем « ..». В более современной файловой системе FFS (Fast File System) размерность пространства имен файлов (ребер) увеличена до 255 символов. Это позволило использовать практически любые мыслимые имена для файлов (вряд ли найдется программист, которому будет не лень набирать для имени более 255 символов), но пришлось изменить структуру каталога (чтобы уменьшить его размеры и не хранить пустые байты). В системе FFS каталог представляет собой таблицу из записей переменной длины. В структуру каждой записи входят: номер индексного узла, длина этой записи, длина имени файла и собственно его имя. Две первых записи в каталоге, как и в s5fs, по-прежнему адресуют саму данную директорию и ее родительский каталог.

Понятие суперблока

В предыдущих разделах уже была рассмотрена часть заголовка раздела. Оставшуюся часть заголовка в s5fs принято называть суперблоком. Суперблок хранит информацию, необходимую для правильного функционирования файловой системы в целом. В нем содержатся, в частности, следующие данные.

- Тип файловой системы.
- Флаги состояния файловой системы.
- Размер логического блока в байтах (обычно кратен 512 байтам).
- Размер файловой системы в логических блоках (включая сам суперблок и массив inode).
- Размер массива индексных узлов (т. е. сколько файлов может быть размещено в файловой системе).
- Число свободных индексных узлов (сколько файлов еще можно создать).
- Число свободных блоков для размещения данных.
- Часть списка свободных индексных узлов.
- Часть списка свободных блоков для размещения данных.

В некоторых модификациях файловой системы s5fs последние два списка выносятся за пределы суперблока, но остаются в заголовке раздела. При первом же обращении к файловой системе суперблок обычно целиком считывается в адресное пространство ядра для ускорения последующих обращений. Поскольку количество логических блоков и индексных узлов в файловой системе может

быть весьма большим, нецелесообразно хранить списки свободных блоков и узлов в суперблоке полностью. При работе с индексными узлами часть списка свободных узлов, находящаяся в суперблоке, постепенно убывает. Когда список почти исчерпан, операционная система сканирует массив индексных узлов и заново заполняет список. Часть списка свободных логических блоков, лежащая в суперблоке, содержит ссылку на продолжение списка, расположенное где-либо в блоках данных. Когда эта часть оказывается использованной, операционная система загружает на освободившееся место продолжение списка, а блок, применявшийся для его хранения, переводится в разряд свободных.

Операции над файлами и директориями

Хотя с точки зрения пользователя рассмотрение операций над файлами и директориями представляется достаточно простым и сводится к перечислению ряда системных вызовов и команд операционной системы, попытка систематического подхода к набору операций вызывает определенные затруднения. Далее речь пойдет в основном о регулярных файлах и файлах типа «директория».

Существует два основных вида файлов, различающихся по методу доступа: файлы последовательного доступа и файлы прямого доступа. Если рассматривать файлы прямого и последовательного доступа как абстрактные типы данных, то они представляются как нечто, содержащее информацию, над которой можно совершать следующие операции.

Для последовательного доступа: чтение очередной порции данных (read), запись очередной порции данных (write) и позиционирование на начале файла (rewind).

Для прямого доступа: чтение очередной порции данных (read), запись очередной порции данных (write) и позиционирование на требуемой части данных (seek). Работа с объектами этих абстрактных типов подразумевает наличие еще двух необходимых операций: создание нового объекта (new) и уничтожение существующего объекта (free).

Расширение математической модели файла за счет добавления к хранимой информации атрибутов, присущих файлу (права доступа, учетные данные), влечет за собой появление еще двух операций: прочитать атрибуты (get attribute) и установить их значения (set attribute). Наделение файлов какой-либо внутренней структурой (как у файла типа «директория») или наложение на набор файлов внешней логической структуры (объединение в ациклический направленный граф) приводит к появлению других наборов операций, составляющих интерфейс работы с файлами, которые, тем не менее, будут являться комбинациями перечисленных выше базовых операций.

Для директории, например, такой набор операций, определяемый ее внутренним строением, может выглядеть так: операции new, free, set attribute и get attribute остаются без изменений, а операции read, write и rewind (seek) заменяются более высокоуровневыми:

– прочитать запись, соответствующую имени файла, — get record;

– добавить новую запись — add record;

– удалить запись, соответствующую имени файла, — delete record.

Неполный набор операций над файлами, связанный с их логическим объединением в структуру директорий, будет выглядеть следующим образом.

- Операции для работы с атрибутами файлов — get attribute, set attribute.
- Операции для работы с содержимым файлов — read, write, rewind(seek) для регулярных файлов и get record, add record, delete record для директорий.
- Операция создания регулярного файла в некоторой директории (создание нового узла графа и добавление в граф нового именованного ребра, ведущего в этот узел из некоторого узла, соответствующего директории) — create. Эту операцию можно рассматривать как суперпозицию двух операций: базовой операции new для регулярного файла и add record для соответствующей директории.
- Операция создания поддиректории в некоторой директории — make directory. Эта операция отличается от предыдущей операции create занесением в файл новой директории информации о файлах с именами « .» и « ..», т. е. по сути дела она есть суперпозиция операции create и двух операций add record.
- Операция создания файла типа «связь» — symbolic link.
- Операция создания файла типа «FIFO» — make FIFO.
- Операция добавления к графу нового именованного ребра, ведущего от узла, соответствующего директории, к узлу, соответствующему любому другому типу файла, — link. Это просто add record с некоторыми ограничениями.
- Операция удаления файла, не являющегося директорией или «связью» (удаление именованного ребра из графа, ведущего к терминальной вершине с одновременным удалением этой вершины, если к ней не ведут другие именованные ребра) — unlink.
- Операция удаления файла типа «связь» (удаление именованного ребра, ведущего к узлу, соответствующему файлу типа «связь», с одновременным удалением этого узла и выходящего из него неименованного ребра, если к этому узлу не ведут другие именованные ребра), — unlink link.
- Операция рекурсивного удаления директории со всеми входящими в нее файлами и поддиректориями — remove directory.
- Операция переименования файла (ребра графа) — rename.
- Операция перемещения файла из одной директории в другую (перемещается точка выхода именованного ребра, которое ведет к узлу, соответствующему данному файлу) — move.
- Возможны и другие подобные операции.

Способ реализации файловой системы в реальной операционной системе также может добавлять новые операции. Если часть информации файловой системы или отдельного файла кэшируется в адресном пространстве ядра, то появляются операции синхронизации данных в кэше и на диске для всей системы в целом (sync) и для отдельного файла (sync file).

Все перечисленные операции могут быть выполнены процессом только при наличии у него определенных полномочий (прав доступа и т. д.). Для выполнения операций над файлами и директориями операционная система предоставляет процессам интерфейс в виде системных вызовов, библиотечных функций и команд операционной системы.

Системные вызовы и команды для выполнения операций над файлами и директориями

Далее в этом разделе, если не будет оговорено особо, под словом «файл» будет подразумеваться регулярный файл. Вся информация об атрибутах файла и его расположении на физическом носителе содержится в соответствующем файлу индексном узле и, возможно, в нескольких связанных с ним логических блоках. Чтобы при каждой операции над файлом не считывать эту информацию с физического носителя заново, представляется логичным, считав информацию один раз при первом обращении к файлу, хранить ее в адресном пространстве процесса или в части адресного пространства ядра, характеризующей данный процесс.

С точки зрения пользовательского процесса каждый файл представляет собой линейный набор байт, снабженный указателем текущей позиции процесса в этом наборе. Все операции чтения из файла и записи в файл производятся в этом наборе с того места, на которое показывает указатель текущей позиции. По завершении операции чтения или записи указатель текущей позиции помещается после конца прочитанного или записанного участка файла. Значение этого указателя является динамической характеристикой файла для использующего его процесса и также должно храниться в PCB.

На самом деле организация информации, описывающей открытые файлы в адресном пространстве ядра операционной системы UNIX, является более сложной. Некоторые файлы могут использоваться одновременно несколькими процессами независимо друг от друга или совместно. Для того чтобы не хранить дублирующуюся информацию об атрибутах файлов и их расположении на внешнем носителе для каждого процесса отдельно, такие данные обычно размещаются в адресном пространстве ядра операционной системы в единственном экземпляре, а доступ к ним процессы получают только при выполнении соответствующих системных вызовов для операций над файлами.

Независимое использование одного и того же файла несколькими процессами в операционной системе UNIX предполагает возможность для каждого процесса совершать операции чтения и записи в файл по своему усмотрению. При этом для корректной работы с информацией необходимо организовывать взаимоисключения для операций ввода-вывода. Совместное использование одного и того же файла в операционной системе UNIX возможно для близко родственных процессов, т. е. процессов, один из которых является потомком другого или которые имеют общего родителя. При совместном использовании файла процессы разделяют некоторые данные, необходимые для работы с файлом, в частности, указатель текущей позиции. Операции чтения или записи, выполненные в одном процессе, изменяют значение указателя текущей позиции во всех близкородственных процессах, одновременно использующих этот файл.

Вся информация о файле, необходимая процессу для работы с ним, может быть разбита на три части:

– данные, специфичные для этого процесса;

– данные, общие для близкородственных процессов, совместно использующих файл, например, указатель текущей позиции;

– данные, являющиеся общими для всех процессов, использующих файл, — атрибуты и расположение файла.

Рис. 21

Взаимосвязи между таблицами, содержащими данные об открытых файлах в системе

Для хранения этой информации применяются три различные связанные структуры данных, лежащие, как правило, в адресном пространстве ядра операционной системы, — таблица открытых файлов процесса, системная таблица открытых файлов и таблица индексных узлов открытых файлов. Для доступа к этой информации в управляющем блоке процесса заводится таблица открытых файлов, каждый непустой элемент которой содержит ссылку на соответствующий элемент системной таблицы открытых файлов, содержащей данные, необходимые для совместного использования файла близко родственными процессами. Из системной таблицы открытых файлов можно по ссылке добраться до общих данных о файле, содержащихся в таблице индексных узлов открытых файлов (рис. 21). Только таблица открытых файлов процесса входит в состав его PCB и, соответственно, наследуется при рождении нового процесса. Индекс элемента в этой таблице (небольшое целое неотрицательное число) или файловый дескриптор является той величиной, характеризующей файл, который мо-

жет оперировать процесс при работе на уровне пользователя. В эту же таблицу открытых файлов помещаются и ссылки на данные, описывающие другие потоки ввода-вывода, такие как pipe и FIFO.

Системный вызов open(). Для выполнения большинства операций над файлами через системные вызовы пользовательский процесс обычно должен указать в качестве одного из параметров системного вызова дескриптор файла, над которым нужно совершить операцию. Поэтому, прежде чем совершать операции, необходимо поместить информацию о файле в таблицы файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом open(). При открытии файла операционная система проверяет, соответствуют ли права, которые запросил процесс для операций над файлом, правам доступа, установленным для этого файла. В случае соответствия она помещает необходимую информацию в системную таблицу файлов и, если этот файл не был ранее открыт другим процессом, в таблицу индексных дескрипторов открытых файлов. Далее операционная система находит пустой элемент в таблице открытых файлов процесса, устанавливает необходимую связь между всеми тремя таблицами и возвращает на пользовательский уровень дескриптор этого файла.

С помощью операции открытия файла операционная система осуществляет отображение из пространства имен файлов в дисковое пространство файловой системы, подготавливая почву для выполнения других операций.

Системный вызов close(). Обратным системным вызовом по отношению к системному вызову open() является системный вызов close(). После завершения работы с файлом процесс освобождает выделенные ресурсы операционной системы и, возможно, синхронизирует информацию о файле, содержащуюся в таблице индексных узлов открытых файлов, с информацией на диске, используя этот системный вызов. Надо отметить, что место в таблице индексных узлов открытых файлов не освобождается по системному вызову close() до тех пор, пока в системе существует хотя бы один процесс, использующий этот файл. Для обеспечения такого поведения в ней для каждого индексного узла заводится счетчик числа открытий, увеличивающийся на 1 при каждом системном вызове open() для данного файла и уменьшающийся на 1 при каждом его закрытии. Очищение элемента таблицы индексных узлов открытых файлов с окончательной синхронизацией данных в памяти и на диске происходит только в том случае, если при очередном закрытии файла этот счетчик становится равным 0.

Операция создания файла. Системный вызов creat()

Прототип системного вызова

```
#include <fcntl.h>
int creat(char *path, int mode);
```

Описание системного вызова

Системный вызов **creat** эквивалентен системному вызову **open()** с параметром **flags**, установленным в значение **O_CREAT | O_WRONLY | O_TRUNC**.

Параметр **path** является указателем на строку, содержащую полное или относительное имя файла.

Если файла с указанным именем не существовало к моменту системного вызова, он будет создан и открыт только для выполнения операций записи. Если файл уже существовал, то он открывается также только для операции записи, при этом его длина уменьшается до 0 с одновременным сохранением всех других атрибутов файла.

Параметр **mode** устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Этот параметр задается как сумма следующих восьмеричных значений:

0400 — разрешено чтение для пользователя, создавшего файл.

0200 — разрешена запись для пользователя, создавшего файл.

0100 — разрешено исполнение для пользователя, создавшего файл.

0040 — разрешено чтение для группы пользователя, создавшего файл.

0020 — разрешена запись для группы пользователя, создавшего файл.

0010 — разрешено исполнение для группы пользователя, создавшего файл.

0004 — разрешено чтение для всех остальных пользователей.

0002 — разрешена запись для всех остальных пользователей.

0001 — разрешено исполнение для всех остальных пользователей.

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса **umask**, а именно — они равны **mode & ~umask**.

Возвращаемое значение

Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение –1 при возникновении ошибки.

Системные вызовы для чтения атрибутов файла

Прототипы системных вызовов

```
#include <sys/stat.h>
#include <unistd.h>
int stat(char *filename,
        struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(char *filename,
         struct stat *buf);
```

Описание системных вызовов

Системные вызовы **stat, fstat** и **lstat** служат для получения информации об атрибутах файла.

Системный вызов **stat** читает информацию об атрибутах файла, на имя которого указывает параметр filename, и заполняет ими структуру, расположенную по адресу **buf**. Заметим, что имя файла должно быть полным либо должно строиться относительно той директории, которая является текущей для процесса, совершившего вызов. Если имя файла относится к файлу типа «связь», то читается информация (рекурсивно!) об атрибутах файла, на который указывает символическая связь.

Системный вызов **lstat** идентичен системному вызову **stat** за одним исключением: если имя файла относится к файлу типа «связь», то читается информация о самом файле типа «связь».

Системный вызов **fstat** идентичен системному вызову **stat**, только файл задается не именем, а своим файловым дескриптором (естественно, файл к этому моменту должен быть открыт).

Для системных вызовов **stat** и **lstat** процессу не нужны никакие права доступа к указанному файлу, но могут понадобиться права для поиска во всех директориях, входящих в специфицированное имя файла.

Структура **stat** в различных версиях UNIX может быть описана поразному. В Linux она содержит следующие поля:

```
struct stat {
dev_t st_dev; /* Устройство, на котором расположен файл. */
ino_t st_ino; /* Номер индексного узла для файла. */
mode_t st_mode; /* Тип файла и права доступа к нему. */
nlink_t st_nlink; /* Счетчик числа жестких связей. */
uid_t st_uid; /* Идентификатор пользователя владельца. */
gid_t st_gid; /* Идентификатор группы владельца. */
dev_t st_rdev; /* Тип устройства для специальных файлов устройств.
*/
off_t st_size; /* Размер файла в байтах (если определен для данно-
го типа файлов). */
unsigned long st_blksize; /* Размер блока для файловой системы. */
unsigned long st_blocks; /* Число выделенных блоков. */
time_t st_atime; /* Время последнего доступа к файлу. */
time_t st_mtime; /* Время последней модификации файла. */
time_t st_ctime; /* Время создания файла. */
}
```

Для определения типа файла можно использовать следующие логические макросы, применяя их к значению поля st_mode:

S_ISLNK(m) — файл типа «связь».

S_ISREG(m) — регулярный файл.

S_ISDIR(m) — директория.

S_ISCHR(m) — специальный файл символьного устройства.

S_ISBLK(m) — специальный файл блочного устройства.

S_ISFIFO(m) — файл типа FIFO.

S_ISSOCK(m) — файл типа «socket».

Младшие 9 бит поля **st_mode** определяют права доступа к файлу подобно тому, как это делается в маске создания файлов текущего процесса.

Возвращаемое значение

Системные вызовы возвращают значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Операции изменения атрибутов файла. Большинство операций изменения атрибутов файла обычно выполняется пользователем в интерактивном режиме с помощью команд операционной системы. Отметим операцию изменения размеров файла, а точнее операцию его обрезания без изменения всех других атрибутов. Для того чтобы уменьшить размеры существующего файла

до 0, не затрагивая остальных его характеристик (прав доступа, даты создания, учетной информации и т. д.), можно при открытии файла использовать в комбинации флагов системного вызова open() флаг O_TRUNC. Для изменения размеров файла до любой желаемой величины (даже для его увеличения во многих вариантах UNIX, хотя изначально этого не предусматривалось!) может использоваться системный вызов ftruncate(). При этом, если размер файла уменьшается, то вся информация в конце файла, не помещающаяся в новый размер, будет потеряна. Если же размер файла увеличивается, то это будет выглядеть так, как будто мы дополнили его до недостающего размера нулевыми байтами.

Системный вызов ftruncate()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int ftruncate(int fd, size_t length);
```

Описание системного вызова

Системный вызов **ftruncate** предназначен для изменения длины открытого регулярного файла.

Параметр **fd** является дескриптором соответствующего файла, т. е. значением, которое вернул системный вызов **open().**

Параметр **length** — значение новой длины для этого файла. Если параметр **length** меньше, чем текущая длина файла, то вся информация в конце файла, не влезающая в новый размер, будет потеряна. Если же он больше, чем текущая длина, то файл будет выглядеть так, как будто дополнили его до недостающего размера нулевыми байтами.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Операции чтения из файла и записи в файл. Для операций чтения из файла и записи в файл применяются системные вызовы read() и write().

Необходимо отметить, что их поведение при работе с файлами имеет определенные особенности, связанные с понятием указателя текущей позиции в файле.

При работе с файлами информация записывается в файл или читается из него, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов read возвращает значение 0, то это означает, что достигнут конец файла.

Операция изменения указателя текущей позиции. Системный вызов lseek(). С точки зрения процесса все регулярные файлы являются файлами прямого доступа. В любой момент процесс может изменить положение указателя текущей позиции в открытом файле с помощью системного вызова lseek().

Особенностью этого системного вызова является возможность помещения указателя текущей позиции в файле за конец файла (т. е. возможность установления значения указателя большего, чем длина файла).

При любой последующей операции записи в таком положении указателя файл будет выглядеть так, как будто возникший промежуток от конца файла до текущей позиции, где начинается запись, был заполнен нулевыми байтами. Если операция записи в таком положении указателя не производится, то никакого изменения файла, связанного с необычным значением указателя, не произойдет (например, операция чтения будет возвращать нулевое значение для количества прочитанных байтов).

Системный вызов lseek()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Описание системного вызова

Системный вызов **lseek** предназначен для изменения положения указателя текущей позиции в открытом регулярном файле.

Параметр **fd** является дескриптором соответствующего файла, т. е. значением, которое вернул системный вызов **open()**.

Параметр **offset** совместно с параметром **whence** определяют новое положение указателя текущей позиции следующим образом.

Если значение параметра whence равно **SEEK_SET**, то новое значение указателя будет составлять **offset** байт от начала файла. Естественно, что значение **offset** в этом случае должно быть не отрицательным.

Если значение параметра **whence** равно **SEEK_CUR**, то новое значение указателя будет составлять старое значение указателя + offset байт. При этом новое значение указателя не должно стать отрицательным.

Если значение параметра **whence** равно **SEEK_END**, то новое значение указателя будет составлять длина файла + offset байт. При этом новое значение указателя не должно стать отрицательным.

Системный вызов **lseek** позволяет выставить текущее значение указателя за конец файла (т. е. сделать его превышающим размер файла). При любой последующей операции записи в этом положении указателя файл будет выглядеть так, как будто возникший промежуток был заполнен нулевыми битами.

Тип данных **off_t** обычно является синонимом типа long.

Возвращаемое значение

Системный вызов возвращает новое положение указателя текущей позиции в байтах от начала файла при нормальном завершении и значение –1 при возникновении ошибки.

Операция добавления информации в файл. Флаг O_APPEND. Если открытие файла системным вызовом open() производилось с установленным флагом O_APPEND, то любая операция записи в файл будет всегда добавлять новые данные в конец файла, независимо от предыдущего положения указателя текущей позиции (как если бы непосредственно перед записью был выполнен вызов lseek() для установки указателя на конец файла).

Операции создания связей. Команда ln, системные вызовы link() и symlink(). Операции создания связи служат для проведения новых именованных

ребер в уже существующей структуре без добавления новых узлов или для опосредованного проведения именованного ребра к уже существующему узлу через файл типа «связь» и неименованное ребро. Допустим, что несколько программистов совместно ведут работу над одним и тем же проектом. Файлы, относящиеся к этому проекту, вполне естественно могут быть выделены в отдельную директорию так, чтобы не смешиваться с файлами других пользователей и другими файлами программистов, участвующих в проекте. Для удобства каждый из разработчиков, конечно, хотел бы, чтобы эти файлы находились в его собственной директории. Этого можно было бы добиться, копируя по мере изменения новые версии соответствующих файлов из директории одного исполнителя в директорию другого исполнителя. Однако тогда, во-первых, возникнет ненужное дублирование информации на диске. Во-вторых, появится необходимость решения тяжелой задачи: синхронизации обновления замены всех копий этих файлов новыми версиями.

Существует другое решение проблемы. Достаточно разрешить файлам иметь несколько имен. Тогда одному физическому экземпляру данных на диске могут соответствовать различные имена файла, находящиеся в одной или в разных директориях. Подобная операция присвоения нового имени файлу (без уничтожения ранее существовавшего имени) получила название операции создания связи.

В операционной системе UNIX связь может быть создана двумя различными способами.

Первый способ, наиболее точно следующий описанной выше процедуре, получил название способа создания жесткой связи (hard link). С точки зрения логической структуры файловой системы этому способу соответствует проведение нового именованного ребра из узла, соответствующего некоторой директории, к узлу, соответствующему файлу любого типа, получающему дополнительное имя. С точки зрения структур данных, описывающих строение файловой системы, в эту директорию добавляется запись, содержащая дополнительное имя файла и номер его индексного узла (уже существующий!). При таком подходе и новое имя файла, и его старое имя, или имена, абсолютно равноправны для операционной системы и могут взаимозаменяемо использоваться для осуществления всех операций.

Использование жестких связей приводит к возникновению двух проблем.

Первая проблема связана с операцией удаления файла. Если необходимо удалить файл из некоторой директории, то после удаления из ее содержимого записи, соответствующей этому файлу, нельзя освободить логические блоки, занимаемые файлом, и его индексный узел, не убедившись, что у файла нет дополнительных имен (к его индексному узлу не ведут ссылки из других директорий), иначе нарушится целостность файловой системы. Для решения этой проблемы файлы получают дополнительный атрибут — счетчик жестких связей (или именованных ребер), ведущих к ним, который, как и другие атрибуты, располагается в их индексных узлах. При создании файла этот счетчик получает значение 1. При создании каждой новой жесткой связи, ведущей к файлу, он увеличивается на 1. При удалении файла из некоторой директории из ее содер-

жимого удаляется запись об этом файле, и счетчик жестких связей уменьшается на 1. Если его значение становится равным 0, происходит освобождение логических блоков и индексного узла, выделенных этому файлу.

Вторая проблема связана с опасностью превращения логической структуры файловой системы из ациклического графа в циклический и с возможной неопределенностью толкования записи с именем « ..» в содержимом директорий. Для их предотвращения во всех существующих вариантах операционной системы UNIX запрещено создание жестких связей, ведущих к уже существующим директориям (несмотря на то, что POSIX-стандарт для операционной системы UNIX разрешает подобную операцию для пользователя root). В операционной системе Linux по непонятной причине дополнительно запрещено создание жестких связей, ведущих к специальным файлам устройств.

Команда ln

Синтаксис команды

```
ln [options] source [dest]
ln [options] source ... directory
```

Описание команды

Команда **ln** предназначена для реализации операции создания связи в файловой системе. В курсе будет использоваться две формы этой команды.

Первая форма команды, когда в качестве параметра **source** задается имя только одного файла, а параметр **dest** отсутствует. Или когда в качестве параметра **dest** задается имя файла, не существующего в файловой системе, создает связь к файлу, указанному в качестве параметра source, в текущей директории с его именем (если параметр **dest** отсутствует) или с именем dest (полным или относительным) в случае наличия параметра dest.

Вторая форма команды, когда в качестве параметра source задаются имена одного или нескольких файлов, разделенные между собой пробелами. А в качестве параметра directory задается имя уже существующей в файловой системе директории, создаются связи к каждому из файлов, перечисленных в параметре source, в директории **directory** с именами, совпадающими с именами перечисленных файлов.

Команда **ln** без опций служит для создания жестких связей (hard link), а команда **ln** с опцией **-s** — для создания мягких (soft link) или символических (symbolic) связей. Для создания жестких связей применяются команда операционной системы **ln** без опций и системный вызов **link()**.

Необходимо отметить, что системный вызов **link()** является одним из немногих системных вызовов, совершающих над файлами операции, которые не требуют предварительного открытия файла, поскольку он подразумевает выполнение единичного действия только над содержимым индексного узла, выделенного связываемому файлу.

Системный вызов link()

Прототип системного вызова

```
#include <unistd.h>
int link(char *pathname,
char *linkpathname);
```

Описание системного вызова

Системный вызов **link** служит для создания жесткой связи к файлу с именем, на которое указывает параметр pathname. Указатель на имя создаваемой связи задается параметром linkpathname (полное или относительное имя связи). Во всех существующих реализациях операционной системы UNIX запрещено создавать жесткие связи к директориям. В операционной системе Linux (по непонятной причине) дополнительно запрещено создавать жесткие связи к специальным файлам устройств.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение –1 при возникновении ошибки.

Жесткая связь файлов является аналогом использования прямых ссылок (указателей) в современных языках программирования, символическая или мягкая связь, до некоторой степени, напоминает косвенные ссылки (указатель на указатель). При создании мягкой связи с именем symlink из некоторой директории к файлу, заданному полным или относительным именем linkpath. В этой директории действительно создается новый файл типа «связь» с именем symlink со своими собственными индексным узлом и логическими блоками. При тщательном рассмотрении можно обнаружить, что все его содержимое составляет только символьная запись имени linkpath.

Операция открытия файла типа «связь» устроена таким образом, что в действительности открывается не сам этот файл, а тот файл, чье имя содержится в нем (при необходимости рекурсивно!). Поэтому операции над файлами, требующие предварительного открытия файла, в реальности будут совершаться не над файлом типа «связь», а над тем файлом, имя которого содержится в нем (или над тем файлом, который, в конце концов, откроется при рекурсивных ссылках). Отсюда в частности следует, что попытки прочитать реальное содержимое файлов типа «связь» с помощью системного вызова read() обречены на неудачу. Как видно, создание мягкой связи, с точки зрения изменения логической структуры файловой системы, эквивалентно опосредованному проведению именованного ребра к уже существующему узлу через файл типа «связь» и неименованное ребро. Создание символической связи не приводит к проблеме, связанной с удалением файлов. Если файл, на который ссылается мягкая связь, удаляется с физического носителя, то попытка открытия файла мягкой связи (а следовательно, и удаленного файла) приведет к ошибке «Файла с таким именем не существует», которая может быть аккуратно обработана приложением. Таким образом, удаление связанного объекта, как упоминалось ранее, лишь отчасти и не фатально нарушит целостность файловой системы.

Неаккуратное применение символических связей пользователями операционной системы может привести к превращению логической структуры файловой системы из ациклического графа в циклический граф. Это, конечно, нежелательно, но не носит столь разрушительного характера, как циклы, которые могли бы быть созданы жесткой связью, если бы не был введен запрет на организацию жестких связей к директориям. Поскольку мягкие связи принципиально отличаются от жестких связей и связей, возникающих между директорией и

файлом при его создании, мягкая связь легко может быть идентифицирована операционной системой или программой пользователя. Для предотвращения зацикливания программ, выполняющих операции над файлами, обычно ограничивается глубина рекурсии по прохождению мягких связей. Превышение этой глубины приводит к возникновению ошибки «Слишком много мягких связей», которая может быть легко обработана приложением. Поэтому ограничения на тип файлов, к которым может вести мягкая связь, в операционной системе UNIX не вводятся.

Для создания мягких связей применяются уже знакомая нам команда операционной системы ln с опцией -s и системный вызов symlink(). Надо отметить, что системный вызов symlink() также не требует предварительного открытия связываемого файла, поскольку он вообще не рассматривает его содержимое.

Системный вызов symlink()

Прототип системного вызова

```
#include <unistd.h>
int symlink(char *pathname,
            char *linkpathname);
```

Описание системного вызова

Системный вызов **symlink** служит для создания символической (мягкой) связи к файлу с именем, на которое указывает параметр pathname. Указатель на имя создаваемой связи задается параметром **linkpathname** (полное или относительное имя связи).

Никакой проверки реального существования файла с именем **pathname** системный вызов не производит.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Операция удаления связей и файлов. Системный вызов unlink(). При рассмотрении операции связывания файлов мы уже почти полностью рассмотрели, как производится операция удаления жестких связей и файлов. При удалении мягкой связи, т. е. фактически файла типа «связь», все происходит как и для обычных файлов. Единственным изменением, с точки зрения логической структуры файловой системы, является то, что при действительном удалении узла, соответствующего файлу типа «связь», вместе с ним удаляется и выходящее из него неименованное ребро. Дополнительно необходимо отметить, что условием реального удаления регулярного файла с диска является не только равенство 0 значения его счетчика жестких связей, но и отсутствие процессов, которые держат этот файл открытым. Если такие процессы есть, то удаление регулярного файла будет выполнено при его полном закрытии последним использующим файл процессом. Для осуществления операции удаления жестких связей и/или файлов можно задействовать команду операционной системы rm или системный вызов unlink(). Системный вызов unlink() также не требует предварительного открытия удаляемого файла, поскольку после его удаления совершать над ним операции бессмысленно.

Системный вызов unlink()

Прототип системного вызова

```
#include <unistd.h>
int unlink(char *pathname);
```

Описание системного вызова

Системный вызов **unlink** служит для удаления имени, на которое указывает параметр **pathname**, из файловой системы.

Если после удаления имени счетчик числа жестких связей у данного файла стал равным 0, то возможны следующие ситуации.

Если в операционной системе нет процессов, которые держат данный файл открытым, то файл полностью удаляется с физического носителя.

Если удаляемое имя было последней жесткой связью для регулярного файла, но какой-либо процесс держит его открытым, то файл продолжает существовать до тех пор, пока не будет закрыт последний файловый дескриптор, ссылающийся на данный файл.

Если имя относится к файлу типа **socket**, **FIFO** или к специальному файлу устройства, то файл удаляется независимо от наличия процессов, держащих его открытым, но процессы, открывшие данный объект, могут продолжать пользоваться им.

Если имя относится к файлу типа «связь», то он удаляется, и мягкая связь оказывается разорванной.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение –1 при возникновении ошибки.

Специальные функции для работы с содержимым директорий

Стандартные системные вызовы open(), read() и close() не могут помочь программисту изучить содержимое файла типа «директория». Для анализа содержимого директорий используется набор функций из стандартной библиотеки языка Си.

Функция opendir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(char *name);
```

Описание функции

Функция **opendir** служит для открытия потока информации для директории, имя которой расположено по указателю name. Тип данных **DIR** представляет собой некоторую структуру данных, описывающую такой поток. Функция opendir подготавливает почву для функционирования других функций, выполняющих операции над директорией, и позиционирует поток на первой записи директории.

Возвращаемое значение

При удачном завершении функция возвращает указатель на открытый поток директории, который будет в дальнейшем передаваться в качестве парамет-

ра всем другим функциям, работающим с этой директорией. При неудачном завершении возвращается значение **NULL**.

С точки зрения программиста в этом интерфейсе директория представляется как файл последовательного доступа, над которым можно совершать операции чтения очередной записи и позиционирования на начале файла. Перед выполнением этих операций директорию необходимо открыть, а после окончания — закрыть. Чтение очередной записи из директории осуществляет функция **readdir()**, одновременно позиционируя нас на начале следующей записи (если она, конечно, существует). Для операции нового позиционирования на начале директории (если вдруг понадобится) применяется функция **rewinddir()**. После окончания работы с директорией ее необходимо закрыть с помощью функции **closedir()**.

Функция readdir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

Описание функции

Функция **readdir** служит для чтения очередной записи из потока информации для директории.

Параметр **dir** представляет собой указатель на структуру, описывающую поток директории, который вернула функция **opendir()**.

Тип данных **struct dirent** представляет собой некоторую структуру данных, описывающую одну запись в директории. Поля этой записи сильно варьируются от одной файловой системы к другой, но одно из полей, которое собственно и будет нас интересовать, всегда присутствует в ней. Это поле **char d_name[ ]** неопределенной длины, не превышающей значения **NAME_MAX+1**, которое содержит символьное имя файла, завершающееся символом конца строки. Данные, возвращаемые функцией **readdir**, переписываются при очередном вызове этой функции для того же самого потока директории.

Возвращаемое значение

При удачном завершении функция возвращает указатель на структуру, содержащую очередную запись директории. При неудачном завершении или при достижении конца директории возвращается значение **NULL**.

Функция rewinddir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dir);
```

Описание функции

Функция **rewinddir** служит для позиционирования потока информации для директории, ассоциированного с указателем **dir** (т. е. с тем, что вернула функция **opendir()**), на первой записи (или на начале) директории.

Функция closedir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

Описание функции

Функция **closedir** служит для закрытия потока информации для директории, ассоциированного с указателем **dir** (т. е. с тем, что вернула функция **opendir()**). После закрытия поток директории становится недоступным для дальнейшего использования.

Возвращаемое значение

При успешном завершении функция возвращает значение 0, при неудачном завершении — значение –1.

Понятие о файлах, отображаемых в память (memory mapped файлах). Системные вызовы mmap(), munmap()

С помощью системного вызова open() операционная система отображает файл из пространства имен в дисковое пространство файловой системы, подготавливая почву для осуществления других операций. С появлением концепции виртуальной памяти, когда физические размеры памяти перестали играть роль сдерживающего фактора в развитии вычислительных систем, стало возможным отображать файлы непосредственно в адресное пространство процессов. Иными словами, появилась возможность работать с файлами как с обычной памятью, заменив выполнение базовых операций над ними с помощью системных вызовов на использование операций обычных языков программирования. Файлы, чье содержимое отображается непосредственно в адресное пространство процессов, получили название файлов, отображаемых в память (по-английски — memory mapped файлов). Необходимо отметить, что такое отображение может быть осуществлено не только для всего файла в целом, но и для его части. С точки зрения программиста работа с такими файлами выглядит следующим образом: отображение файла из пространства имен в адресное пространство процесса происходит в два этапа — сначала выполняется отображение в дисковое пространство, а уже затем из дискового пространства в адресное. Поэтому вначале файл необходимо открыть, используя обычный системный вызов open(). Вторым этапом является отображение файла целиком или частично из дискового пространства в адресное пространство процесса. Для этого используется системный вызов mmap(). Файл после этого можно и закрыть, выполнив системный вызов close(), так как необходимую информацию о расположении файла на диске мы уже сохранили в других структурах данных при вызове mmap().

Системный вызов mmap()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
```

```
void *mmap (void *start, size_t length,
    int prot, int flags, int fd,
    off_t offset);
```

Описание системного вызова

Системный вызов **mmap** служит для отображения предварительно открытого файла (например, с помощью системного вызова **open()**) в адресное пространство вычислительной системы. После его выполнения файл может быть закрыт (например, системным вызовом **close()**), что никак не повлияет на дальнейшую работу с отображенным файлом. Настоящее описание не является полным описанием системного вызова. Для получения полной информации обращайтесь к **UNIX Manual**. Параметр **fd** является файловым дескриптором для файла, который нужно отобразить в адресное пространство (т. е. значением, которое вернул системный вызов **open()**).

Ненулевое значение параметра **start** может использоваться только очень квалифицированными системными программистами, поэтому в практикуме будем всегда полагать его равным значению **NULL**, позволяя операционной системе самой выбрать начало области адресного пространства, в которую будет отображен файл. В память будет отображаться часть файла, начиная с позиции внутри него, заданной значением параметра **offset** — смещение от начала файла в байтах, и длиной, равной значению параметра **length** (естественно, тоже в байтах). Значение параметра **length** может и превышать реальную длину от позиции **offset** до конца существующего файла. На поведении системного вызова это никак не отразится, но в дальнейшем при попытке доступа к ячейкам памяти, лежащим вне границ реального файла, возникнет сигнал **SIGBUS** (реакция на него по умолчанию — прекращение процесса с образованием **core** файла). Параметр **flags** определяет способ отображения файла в адресное пространство. Будем использовать только два его возможных значения: **MAP_SHARED** и **MAP_PRIVATE**. Если в качестве его значения выбрано **MAP_SHARED**, то полученное отображение файла впоследствии будет использоваться и другими процессами, вызвавшими mmap для этого файла с аналогичными значениями параметров, а все изменения, сделанные в отображенном файле, будут сохранены во вторичной памяти. Если в качестве значения параметра **flags** указано **MAP_PRIVATE**, то процесс получает отображение файла в свое монопольное распоряжение, но все изменения в нем не могут быть занесены во вторичную память (т. е., проще говоря, не сохранятся). Параметр prot определяет разрешенные операции над областью памяти, в которую будет отображен файл. В качестве его значения мы будем использовать значения **PROT_READ** (разрешено чтение), **PROT_WRITE** (разрешена запись) или их комбинацию через операцию «побитовое или» — « | ». Необходимо отметить две существенные особенности системного вызова, связанные с этим параметром.

Значение параметра **prot** не может быть шире, чем операции над файлом, заявленные при его открытии в параметре **flags** системного вызова **open()**. Например, нельзя открыть файл только для чтения, а при его отображении в память использовать значение prot = **PROT_READ | PROT_WRITE**.

В результате ошибки в операционной системе Linux при работе на 486-х и 586-х процессорах попытка записать в отображение файла, открытое только для записи, более 32 байт одновременно приводит к ошибке (возникает сигнал о нарушении защиты памяти).

Возвращаемое значение

При нормальном завершении системный вызов возвращает начальный адрес области памяти, в которую отображен файл (или его часть), при возникновении ошибки — специальное значение **MAP_FAILED**.

После этого с содержимым файла можно работать, как с содержимым обычной области памяти.

По окончании работы с содержимым файла, необходимо освободить дополнительно выделенную процессу область памяти, предварительно синхронизировав содержимое файла на диске с содержимым этой области (если, конечно, необходимо). Эти действия выполняет системный вызов **munmap()**.

Системный вызов munmap

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
int munmap (void *start, size_t length);
```

Описание системного вызова

Системный вызов **munmap** служит для прекращения отображения **memory mapped** файла в адресное пространство вычислительной системы. Если при системном вызове **mmap()** было задано значение параметра **flags**, равное **MAP_SHARED**, и в отображении файла была разрешена операция записи (в параметре **prot** использовалось значение **PROT_WRITE**), то **munmap** синхронизирует содержимое отображения с содержимым файла во вторичной памяти. После его выполнения области памяти, использовавшиеся для отображения файла, становятся недоступны текущему процессу. Параметр **start** является адресом начала области памяти, выделенной для отображения файла, т. е. значением, которое вернул системный вызов **mmap()**. Параметр **length** определяет ее длину, и его значение должно совпадать со значением соответствующего параметра в системном вызове **mmap()**.

Возвращаемое значение

При нормальном завершении системный вызов возвращает значение 0, при возникновении ошибки — значение –1.

Программа 29 для иллюстрации работы с memory mapped файлом

```
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
#include <iostream>
#include <stdio.h>
using namespace std;
int main(void)
{
```

```
int fd; /* Файловый дескриптор для файла, в котором будет
храниться информация. */
size_t length; /* Длина отображаемой части файла. */
int i;
/* Ниже следует описание типа структуры, которым мы забьем
файл, и двух указателей на подобный тип. Указатель ptr будет
использоваться в качестве начального адреса выделенной области
памяти, а указатель tmpptr — для перемещения внутри этой
области.*/
struct A
{
     double f;
     double f2;
}
*ptr, *tmpptr;
/* Открытие файла или сначала создание его (если такого файла
не было). Права доступа к файлу при создании определяются как
read-write для всех категорий пользователей (0666). Из-за
ошибки в Linux необходимо ниже в системном вызове mmap()
разрешить в отображении файла и чтение, и запись, хотя реально
нужна только запись. Поэтому и при открытии файла необходимо
задавать O_RDWR. */
fd = open(«mapped.dat», O_RDWR | O_CREAT, 0666);
if( fd == -1)
{
     /* Если файл открыть не удалось, вывод сообщения об
     ошибке и завершение работы. */
     cout<<«File open failed!»<<endl;
     exit(1);
}
/* Вычисление будущей длины файла (для 100 000 структур.) */
length = 100000*sizeof(struct A);
/* Вновь созданный файл имеет длину 0. Если его отобразить в
память с такой длиной, то любая попытка записи в выделенную
память приведет к ошибке. Увеличение длины файла с помощью
вызова ftruncate(). */
ftruncate(fd,length);
/* Отображаем файл в память. Разрешенные операции над
отображением указываем как PROT_WRITE | PROT_READ по уже
названным причинам. Значение флагов ставится в MAP_SHARED, так
как необходимо сохранить информацию, которая занесена в
отображение, на диске. Файл отображаем с его начала
(offset = 0) и до конца (length =длине файла). */
ptr = (struct A *)mmap(NULL, length, PROT_WRITE | PROT_READ,
MAP_SHARED, fd, 0);
/* Файловый дескриптор более не нужен, и его можно закрыть. */
close(fd);
if( ptr == MAP_FAILED )
{
     /* Если отобразить файл не удалось, сообщение об ошибке
     и завершение работы. */
     cout<<«Mapping failed!»<< endl;
     exit(2);
```

```
    }
    /* В цикле заполнение образа файла числами от 1 до 100 000 и
    их квадратами. Для  перемещения по области памяти используется
    указатель tmpptr, так как указатель ptr на начало образа файла
    понадобится для прекращения и отображения вызовом munmap(). */
    tmpptr = ptr;
    for(i = 1; i <=100000; i++)
    {
         tmpptr->f = i;
         tmpptr->f2 = tmpptr->f*tmpptr->f;
         tmpptr++;
    }
    /* Прекращение отображения файла в память, запись содержимого
    отображения на диск и освобождение памяти. */
    munmap((void *)ptr, length);
return 0;
}
```

Эта программа создает файл, отображает его в адресное пространство процесса и заносит в него информацию с помощью обычных операций языка Си. Обратите внимание на необходимость увеличения размера файла перед его отображением. Созданный файл имеет нулевой размер, и, если его с этим размером отобразить в память, можно записать в него или прочитать из него не более 0 байт, т. е. ничего. Для увеличения размера файла использован системный вызов ftruncate(), хотя это можно было бы сделать и любым другим способом. При отображении файла необходимо разрешить в нем и запись, и чтение, хотя реально происходит только запись. Это сделано для того, чтобы избежать ошибки в операционной системе Linux, связанной с использованием 486-х и 586-х процессоров. Такой список разрешенных операций однозначно требует, чтобы при открытии файла системным вызовом open() файл открывался и на запись, и на чтение. Поскольку информацию мы желаем сохранить на диске, при отображении использовано значение флагов MAP_SHARED.

Операции над файловыми системами. Монтирование файловых систем

В предыдущих разделах рассматривалась только одна файловая система, расположенная в одном разделе физического носителя. Как только предполагаем сосуществование нескольких файловых систем в рамках одной операционной системы, то встает вопрос о логическом объединении структур этих файловых систем. При работе операционной системы изначально доступна лишь одна, так называемая, корневая, файловая система. Прежде, чем приступить к работе с файлом, лежащим в некоторой другой файловой системе, мы должны встроить ее в уже существующий ациклический граф файлов. Эта операция над файловой системой называется монтированием файловой системы (mount). Для монтирования файловой системы в существующем графе должна быть найдена или создана некоторая пустая директория — точка монтирования, к которой и присоединится корень монтируемой файловой системы. При операции монтирования в ядре заводятся структуры данных, описывающие файловую систему, а в vnode для точки монтирования файловой системы помещается специальная

информация. Монтирование файловых систем обычно является прерогативой системного администратора и осуществляется командой операционной системы mount в ручном режиме либо автоматически при старте операционной системы. Использование этой команды без параметров не требует специальных полномочий и позволяет пользователю получить информацию обо всех смонтированных файловых системах и соответствующих им физических устройствах. Для пользователя также обычно разрешается монтирование файловых систем, расположенных на гибких магнитных дисках. Для первого накопителя на гибких магнитных дисках такая команда в Linux будет выглядеть следующим образом:

```
mount /dev/fd0 <имя пустой директории>,
```

*где <**имя пустой директории**> описывает точку монтирования, а **/dev/fd0** — специальный файл устройства, соответствующего этому накопителю.*

Команда mount

Синтаксис команды

```
mount [-hV]
mount [-rw] [-t fstype] device dir
```

Описание команды

Команда **mount** предназначена для выполнения операции монтирования файловой системы и получения информации об уже смонтированных файловых системах.

Опции **-h**, **-V** используются при вызове команды без параметров и служат для следующих целей:

-h — вывести краткую инструкцию по пользованию командой;

-V — вывести информацию о версии команды mount.

Команда **mount** без опций и без параметров выводит информацию обо всех уже смонтированных файловых системах.

Команда **mount** с параметрами служит для выполнения операции монтирования файловой системы.

Параметр **device** задает имя специального файла для устройства, содержащего файловую систему.

Параметр **dir** задает имя точки монтирования (имя некоторой уже существующей пустой директории). При монтировании могут использоваться следующие опции:

-r — смонтировать файловую систему только для чтения (read only);

-w — смонтировать файловую систему для чтения и для записи (read/write);

-t fstype — задать тип монтируемой файловой системы как fstype.

Поддерживаемые типы файловых систем в операционной системе Linux: **adfs, affs, autofs, coda, coherent, cramfs, devpts, efs, ext, ext2, ext3, hfs, hpfs, iso9660 (для CD), minix, msdos, ncpfs, nfs, ntfs, proc, qnx4, reiserfs, romfs, smbfs, sysv, udf, ufs, umsdos, vfat, xenix, xfs, xiafs**. При отсутствии явно заданного типа команда для большинства типов файловых систем способна опознать его автоматически. Если не собираемся использовать смонтированную файловую систему в дальнейшем (например, хотим вынуть ранее смонтированный

диск), то необходимо выполнить операцию логического разъединения смонтированных файловых систем (umount). Для этой операции, которая тоже, как правило, является привилегией системного администратора, используется команда umount (может выполняться в ручном режиме или автоматически при завершении работы операционной системы).

Для пользователя обычно доступна команда отмонтирования файловой системы на диске в форме

```
umount <имя точки монтирования>
```

где <**имя точки монтирования**> — это <**имя пустой директории**>, использованное ранее в команде mount, или в форме umount **/dev/fd0**, где **/dev/fd0** — специальный файл устройства, соответствующего первому накопителю.

Заметим, что для последующей корректной работы операционной системы при удалении физического носителя информации необходимо предварительное логическое разъединение файловых систем, если они перед этим были объединены.

Команда umount

Синтаксис команды

```
umount [-hV]
umount device
umount dir
```

Описание команды

Команда **umount** предназначена для выполнения операции логического разъединения ранее смонтированных файловых систем.

Опции **-h**, **-V** используются при вызове команды без параметров и служат для следующих целей:

-h — вывести краткую инструкцию по пользованию командой;

-V — вывести информацию о версии команды umount.

Команда **umount** с параметром служит для выполнения операции логического разъединения файловых систем. В качестве параметра может быть задано либо имя устройства, содержащего файловую систему — **device**, либо имя точки монтирования файловой системы (т. е. имя директории, которое указывалось в качестве параметра при вызове команды mount) — **dir**.

Файловая система не может быть отмонтирована до тех пор, пока она находится в использовании (**busy**) — например, когда в ней существуют открытые файлы, какой-либо процесс имеет в качестве рабочей директории директорию в этой файловой системе и т. д.

Блочные, символьные устройства. Понятие драйвера. Блочные, символьные драйверы, драйверы низкого уровня. Файловый интерфейс

Все устройства ввода-вывода можно разделить на относительно небольшое число типов в зависимости от набора операций, которые могут ими выполняться. Такое деление позволяет организовать «слоистую» структуру подсистемы ввода-вывода, вынеся все аппаратно-зависимые части в драйверы уст-

ройств, с которыми взаимодействует базовая подсистема ввода-вывода, осуществляющая стратегическое управление всеми устройствами.

В операционной системе UNIX принята упрощенная классификация: все устройства разделяются по способу передачи данных на символьные и блочные. Символьные устройства осуществляют передачу данных байт за байтом, в то время как блочные устройства передают блок байт как единое целое. Типичным примером символьного устройства является клавиатура, примером блочного устройства — жесткий диск. Непосредственное взаимодействие операционной системы с устройствами ввода-вывода обеспечивают их драйверы. Существует пять основных случаев, когда ядро обращается к драйверам.

- Автоконфигурация. Происходит в процессе инициализации операционной системы, когда ядро определяет наличие доступных устройств.
- Ввод-вывод. Обработка запроса ввода-вывода.
- Обработка прерываний. Ядро вызывает специальные функции драйвера для обработки прерывания, поступившего от устройства, в том числе, возможно, для планирования очередности запросов к нему.
- Специальные запросы. Например, изменение параметров драйвера или устройства.
- Повторная инициализация устройства или останов операционной системы.

Как и устройства подразделяются на символьные и блочные, так и драйверы существуют символьные и блочные. Особенностью блочных устройств является возможность организации на них файловой системы, поэтому блочные драйверы обычно используются файловой системой UNIX. При обращении к блочному устройству, не содержащему файловой системы, применяются специальные драйверы низкого уровня, как правило, представляющие собой интерфейс между ядром операционной системы и блочным драйвером устройства. Для каждого из этих трех типов драйверов были выделены основные функции, которые базовая подсистема ввода-вывода может совершать над устройствами и драйверами: инициализация устройства или драйвера, временное завершение работы устройства, чтение, запись, обработка прерывания, опрос устройства и т. д. Эти функции были систематизированы и представляют собой интерфейс между драйверами и базовой подсистемой ввода-вывода. Каждый драйвер определенного типа в операционной системе UNIX получает собственный номер, который по сути дела является индексом в массиве специальных структур данных операционной системы — коммутаторе устройств соответствующего типа. Этот индекс принято также называть старшим номером устройства, хотя на самом деле он относится не к устройству, а к драйверу. Несмотря на наличие трех типов драйверов, в операционной системе используется всего два коммутатора: для блочных и символьных драйверов. Драйверы низкого уровня распределяются между ними по преобладающему типу интерфейса (к какому типу ближе — в такой массив и заносятся).

Каждый элемент коммутатора устройств обязательно содержит адреса (точки входа в драйвер), соответствующие стандартному набору функций интерфейса, которые и вызываются операционной системой для выполнения тех или иных действий над устройством и/или драйвером.

Помимо старшего номера устройства существует еще и младший номер устройства, который передается драйверу в качестве параметра и смысл которого определяется самим драйвером. Например, это может быть номер раздела на жестком диске (partition), доступ к которому должен обеспечить драйвер (надо отметить, что в операционной системе UNIX различные разделы физического носителя информации рассматриваются как различные устройства). В некоторых случаях младший номер устройства может не использоваться, но для единообразия он должен присутствовать. Таким образом, пара драйвер-устройство всегда однозначно определяется в операционной системе заданием пары номеров (старшего и младшего номеров устройства) и типа драйвера (символьный или блочный). Для связи приложений с драйверами устройств операционная система UNIX использует файловый интерфейс. Каждой тройке тип-драйвер-устройство в файловой системе соответствует специальный файл устройства, который не занимает на диске никаких логических блоков, кроме индексного узла. В качестве атрибутов этого файла помимо обычных атрибутов используются соответствующие старший и младший номера устройства и тип драйвера (тип драйвера определяется по типу файла: ибо есть специальные файлы символьных устройств и специальные файлы блочных устройств, а номера устройств занимают место длины файла, скажем, для регулярных файлов). Когда открывается специальный файл устройства, операционная система, в числе прочих действий, заносит в соответствующий элемент таблицы открытых виртуальных узлов указатель на набор функций интерфейса из соответствующего элемента коммутатора устройств. При попытке чтения из файла устройства или записи в файл устройства виртуальная файловая система будет транслировать запросы на выполнение этих операций в соответствующие вызовы нужного драйвера.

Аппаратные прерывания (interrupt), исключения (exception), программные прерывания (trap, software interrupt) и их обработка

После выдачи запроса ввода-вывода у процессора существует два способа узнать о том, что обработка запроса устройством завершена. Первый способ заключается в регулярной проверке процессором бита занятости в регистре состояния контроллера соответствующего устройства (polling). Второй способ заключается в использовании прерываний. При втором способе процессор имеет специальный вход, на который устройства ввода-вывода, используя контроллер прерываний или непосредственно, выставляют сигнал запроса прерывания (interrupt request) при завершении операции ввода-вывода. При наличии такого сигнала процессор после выполнения текущей команды не выполняет следующую, а, сохранив состояние ряда регистров и, возможно, загрузив в часть регистров новые значения, переходит к выполнению команд, расположенных по некоторым фиксированным адресам. После окончания обработки прерывания можно восстановить состояние процессора и продолжить его работу с команды, выполнение которой было отложено. Аналогичный механизм часто используется при обработке исключительных ситуаций (exception), возникающих при выполнении команды процессором (неправильный адрес в команде, защита па-

мяти, деление на ноль и т. д.). В этом случае процессор не завершает выполнение команды, а поступает, как и при прерывании, сохраняя свое состояние до момента начала ее выполнения. Этим же механизмом часто пользуются и для реализации так называемых программных прерываний (software interrupt, trap), применяемых, например, для переключения процессора из режима пользователя в режим ядра внутри системных вызовов. Для выполнения действий, аналогичных действиям по обработке прерывания, процессор в этом случае должен выполнить специальную команду. Как правило, обработку аппаратных прерываний от устройств ввода-вывода производит сама операционная система, не доверяя работу с системными ресурсами процессам пользователя. Обработка же исключительных ситуаций и некоторых программных прерываний вполне может быть возложена на пользовательский процесс через механизм сигналов.

МЕТОДЫ СЕТЕВЫХ СОЕДИНЕНИЙ

Компьютерные сети глобально изменили жизнь человека. Они были практически неизвестны в 70-х и даже в 80-х гг. прошлого века. Но в начале 1990-х гг. произошел резкий скачок. Вероятнее всего это связано с появлением World Wide Web (WWW) и графических Web-браузеров, благодаря которым Интернет «пришел в каждый дом». Возможно, количество сетевых соединений достигло какого-то критического предела. Может быть, этот предел превысил число сетевых программ. Как бы то ни было, сейчас о сетях знают все. А самое главное, что каждый знает о существовании Интернета.

Интернет объединяет тысячи компьютеров и гаджетов, на многих из которых выполняются серверы — программы, принимающие запросы от клиентов и обрабатывающие их. Благодаря тому, что протоколы, на которых базируется Интернет, допускают межплатформенное взаимодействие, в обмене данными могут участвовать серверы и клиенты, выполняющиеся на различных компьютерах и в разных ОС. В последнее время одной из самых популярных ОС стала Linux. Установленная на дешевом компьютере x86 операционная система Linux обеспечивает безотказную работу серверов, поддерживающих узлы малого и среднего размеров. С ростом производительности компьютеров появляется возможность выполнения в системе Linux-серверов, обрабатывающих большие объемы данных. В результате от системного администратора часто требуется умение настраивать ОС Linux и серверы, выполняющиеся в ее среде.

Сетевые соединения представлены двумя основными вариантами: локальными сетями (LAN, Local Area Networks), работающими на территории здания, и глобальными сетями (WAN, Wide Area Networks), которые покрывают территорию города, страны или даже всего мира. Наиболее известной и важной разновидностью локальной сети является Ethernet, поэтому ее рассмотрим в качестве примера для изучения. А в качестве примера глобальной сети будет взят Internet, несмотря на то, что технически он не является единой сетью, а представляет собой объединение тысяч отдельных сетей. Но для нашего случая достаточно будет представлять его единой глобальной сетью.

Ethernet

Классическая сеть Ethernet, описание которой дано в стандарте IEEE Standard 706.3, состоит из коаксиального кабеля, соединяющего несколько компьютеров. Он называется Ethernet, по которому, как считалось ранее, распространяются электромагнитные волны.

Основные операции сети Ethernet подчиняются простому набору правил. Чтобы лучше понять эти правила, важно разобраться в основной терминологии Ethernet.

Канал передачи. Устройства сети Ethernet подключаются к общему каналу передачи, по которому передаются электрические сигналы. Исторически сложилось, что каналом передачи раньше был медный коаксиальный кабель,

однако в наше время для этих целей чаще используется витая пара или волоконно-оптический кабель.

Сегмент. Сегментом сети Ethernet называют один совместно используемый канал передачи.

Узел. Узлами называются устройства, подключаемые к сегменту.

Кадр (или фрейм). Узлы обмениваются короткими информационными сообщениями, которые называют кадрами. Кадр — порция информации, размер которой может меняться.

Ethernet — семейство технологий пакетной передачи данных для компьютерных сетей. Довольно любопытна история протокола **IEEE-802.3**. Первая версия Ethernet была создана в 1960-х гг., в гавайском университете, базировалась на алгоритме доступа ALOHA и стала первой пакетной радиосетью, в которой использовался метод множественного доступа с контролем несущей и обнаружения конфликтов (CSMA/CD). Позже компания Xerox изобрела систему на базе алгоритма CSMA/CD с быстродействием 2,94 Мбит/с. Окончательные принципы сети Ethernet были разработаны в 1976 г. Меткальфом и Боггсом. Ethernet занимает в данный момент лидирующее положение вместе со своими скоростными версиями Fast Ethernet (FE), Giga Ethernet (GE) и 10GE. Недостаток данной сети — отсутствие гарантии времени доступа к среде (и механизмов, снабжающих приоритетное обслуживание). Это делает сеть менее перспективной для решения технологических задач нашего времени. Некоторые затруднения иногда оказывает ограничение на максимальное поле данных, равное примерно 1500 байт.

Для технологий, существовавших в момент разработки стандарта Ethernet, выбор длины поля данных определялся уровнем ошибок (BER).

В качестве среды передачи данных первое время использовался толстый коаксиальный кабель (Z=50 Ом) и подключение к нему выполнялось через отдельные устройства (трансиверы). В дальнейшем сети стали строиться на базе тонкого коаксиального кабеля, но и данное решение было довольно дорогим. Создание дешевых широкополосных скрученных пар и соответствующих разъемов расширила границы Ethernet. Однако и эта технология отдает свои позиции оптоволоконным кабелям.

Для различного быстродействия Ethernet применяются различные схемы кодирования, но алгоритм доступа и формат кадра сохраняется, что гарантирует программную совместимость.

Однако важным препятствием замены стандарта на более совершенный является наличие сотен миллионов интерфейсов Ethernet.

Архитектура сетей Ethernet

Последовательный формат передачи информации применяется в большинстве нынешних физических сетевых средах. Ethernet относится к этой разновидности. Компания Xerox реализовала разработку протокола Ethernet в 1973 г., а объединение компаний Xerox, Intel и DEC (DIX) в 1979 г. предоставило документ для стандартизации протокола в IEEE. С внесением небольших изменений предложение было принято комитетом 802.3 в 1983 г.

Рис. 22
Формат кадра сетей Ethernet (цифры в верхней части рисунка показывают размер поля в байтах)

Рассмотрим поля, представленные на данном рисунке:

7 — преамбула (предназначена для стабилизации и синхронизации среды);

1 — **SFD** (Start Frame Delimiter; служит для обнаружения начала кадра);

1 — **EFD** (End Frame Delimiter; устанавливает конец кадра);

4 — поле контрольной суммы (**CRC** — Cyclic Redundancy Check; формируется и контролируется на аппаратном уровне).

Далее следует межпакетная пауза IPG (Inter Packet Gap; длина от 96 бит-тактов). Максимальный размер кадра — 1518 байт. Интерфейс отслеживает все пакеты, следующие по кабельному сегменту, к которому он подключен. Корректность пакета по CRC определяется после проверки адреса места назначения, по длине и кратности целому числу байт. Вероятность возникновения ошибки передачи — примерно 2-32.

Для пересылки данных в сети и синхронизации (с быстродействием менее 1 Гбит/с) применяется манчестерский код. Его суть заключается в следующем: бит-символ делится на две части, из которых в первой половине кодируемый сигнал представлен в логически дополнительном виде, а во второй — в обычном (сигнал логического 0 — CD0 характеризуется в первой половине уровнем HI (+0,85 В), а во второй — LO (–0,85 В)). Соответственно сигнал CD1 характеризуется в первой половине бит-символа уровнем LO, а во второй — HI. Формы сигналов при манчестерском кодировании представлены на рисунке 23.

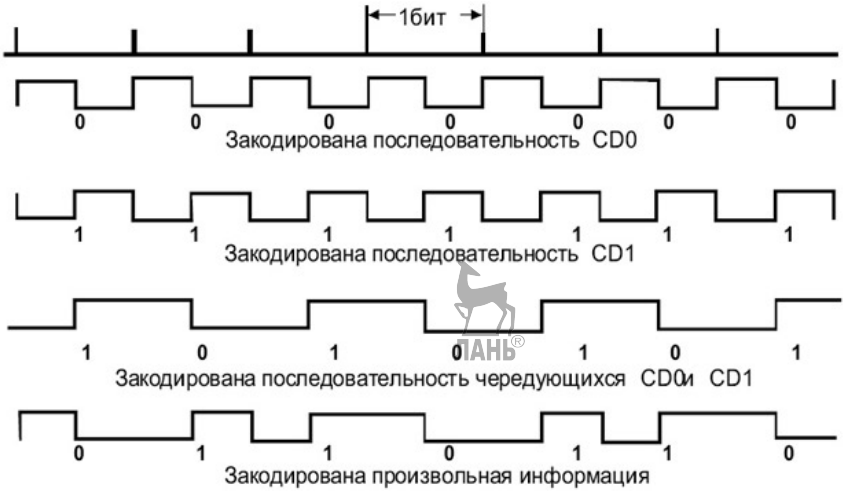


Рис. 23
Примеры кодировки с использованием манчестерского кода

Наименьшая длительность пакета в сети характеризуется тем, что если столкновение пакетов произошло раньше, чем закончит передачу кадра, то отправитель обязан узнать об этом. Наряду с этим длительность передаваемого пакета должна быть больше удвоенного максимального времени распространения кадра до самой удаленной точки сетевого сегмента.

В данном случае имеется ввиду сегмент, организуемый кабелями и повторителями. Минимальная длительность кадра для конфигураций 10 Мбит/с сети с четырьмя повторителями и 500-метровыми кабельными сегментами равна 64 байтам. Повторители вносят существенный вклад в задержку.

Так, чтобы кадр в произвольном исходе обладал должным размером (при ограничении размера пакета менее 64 байт), прибавляются байты-заполнители. Длина пакета при приеме контролируется. В том случае, если длина пакета превышает 1518 байт, он считается избыточным и обрабатываться не будет. Происходит та же ситуация, если кадры короче 64 байт. Каждый пакет должен иметь длину, кратную 8 бит. Адрес считается широковещательным, если в поле адресата содержатся все единицы, то есть обращенным ко всем рабочим станциям локального сегмента сети.

Предел на наименьшую длину кадра теоретически снимается при подключении ЭВМ к сети посредством переключателя. Тем не менее, работа с более короткими кадрами при этом станет осуществимой только при замене сетевого интерфейса на нестандартный (как у отправителя, так и получателя).

Пакет Ethernet может передавать данные в пределах от 46 до 1500 байт.

Рис. 24
Формат MAC-адреса

Рассмотрим субполя, представленные на рисунке 24:

– **I/G** — это флаг индивидуального или группового адреса (I/G = 0 — адрес при данном условии представляет собой индивидуальный адрес сетевого объекта; I/G = 1 — адрес представляется как мультикастинговый, и в этом случае последующее разбиение адреса на субполя нецелесообразно). В пределах подсети мультикастинговые адреса дают возможность обращения одновременно к нескольким станциям.

– **U/L** — флаг местного или универсального управления, который выражает механизм присвоения адреса сетевому интерфейсу (U/L=1 определяет локальную адресацию, U/L=I/G=0 типично для стандартных уникальных адресов, присваиваемых интерфейсу его изготовителем).

– **OUI** позволяет установить производителя сетевого интерфейса. Каждому изготовителю присваивается один или несколько OUI. Размер субполя разрешает идентифицировать около четырех миллионов всевозможных изготовителей. Одинаковых интерфейсов одного и того же изготовителя с теми же но-

мерами быть не должно. Размер поля позволяет произвести около 16 миллионов интерфейсов. Объединение **OUI** и **OUA** образовывает **UAA** (IEEE-адрес).

Если в поле кадра записан код протокола менее 1500, то это поле характеризует длину кадра. В ином случае — это код протокола, пакет которого инкапсулирован в поле данных кадра.

Доступ к каналу Ethernet базируется на методе CSMA/CD. В Ethernet любая станция, подсоединенная к сети, может попытаться начать передачу пакета в случае, если кабельный сегмент, к которому она подключена, свободен. Интерфейс уточняет освобожденность сегмента по недостатку «несущей» в течение 96 бит-тактов. Случается, что попытку передачи совершат 2 или более станций, к тому же что задержки в повторителях и кабелях имеют шансы стать довольно большими. Повторения попыток называются коллизиями. Коллизия идентифицируется по присутствию в канале сигнала, уровень которого отвечает работе 2 или более трансиверов одновременно. При обнаружении столкновения станция прекращает передачу. Повторение попытки осуществимо после выдержки (кратной 51,2 мкс, но не превосходящей 52 мкс). Ее величина — псевдослучайное значение, которое рассчитывается каждой станцией самостоятельно (T = RAND(0,2min(N,10)), где N — содержимое счетчика попыток, а 10 — backoffLimit).

Преимущественно, впоследствии возникновения столкновения время делится на ряд дискретных доменов с величиной, равной удвоенному времени распространения пакета в сегменте (RTT; для максимального — 512 бит-тактов). Каждая станция ждет 0 или 2 временного домена, до того как осуществить еще одну попытку после первого столкновения. Впоследствии второго столкновения любая станция может выждать 0, 1, 2 или 3 временного домена и т. д. После n-го столкновения случайная величина располагается в пределах 0 – (2n – 1). Максимум случайной выдержки перестает возрастать и остается на уровне 1023 после 10 столкновений.

Согласно расчетам, зависимость среднего времени доступа от длины кабельного сегмента прямо пропорциональна: чем длиннее кабельный сегмент, тем больше среднее время доступа. Далее при столкновении станция увеличивает на единицу счетчик попыток и начинает следующую передачу. Максимальное число попыток по умолчанию равно 16. Если число попыток исчерпано, связь обрывается и выводится сообщение о недоступности. В данном случае передаваемый кадр будет невозвратимо утерян.

«Синхронизации» начала передачи пакетов несколькими станциями способствует длинный кадр (за время передачи может возникнуть необходимость передачи у двух и более станций). В момент обнаружения завершения пакета включатся таймеры IPG. При этом информация об окончании передачи пакета достигает станций сегмента не одновременно. В данном случае задержки являются причиной того, что обстоятельство начала передачи очередного пакета одной из станций не становится известным сразу. При включении в столкновение нескольких станций они могут оповестить остальные станции об этом, подав сигнал «затора» (JAM — не менее 32 бит). Содержание этих 32 бит не регулируется. В данной схеме повторение столкновений маловероятно. Запредель-

ная суммарная длина логического кабельного сегмента, слишком большое число повторителей, обрыв кабеля или неисправность одного из интерфейсов могут стать источником огромного числа столкновений. Однако сами по себе столкновения — это инструмент, стабилизирующий доступ к сетевой среде.

Логический кабельный сегмент (область столкновений) предполагает наличие одного или нескольких кабельных сегментов, связанных повторителями. Оценка столкновений — способ эффективной диагностики сети. Фрагменты (укороченные пакеты) являются причиной локальных столкновений (столкновения на сегменте, к которому напрямую подключена рабочая станция). Основная масса трансиверов и репитеров имеют на собственных лицевых панелях индикаторы столкновений.

Далее обратим внимание на влияние сигнала **JAM**.

Если узлов, желающих что-то передать во время пересылки столкнувшихся пакетов и за время передачи сигнала **JAM**, больше одного, то это послужит причинами синхронизации начала передачи этими узлами и увеличения вероятности столкновения. Данная синхронизация представляет собой источник «коллапса» сети при большой загрузке.

Если допустимое число повторителей в сетевом сегменте превышено (или превышена предельно допустимая длина кабелей), то это приведет к увеличению задержки RTT. При наименьшей величине пакета (64 байта) возможна ситуация, когда отправитель обнаружит столкновение позже, чем завершится передача данного пакета. В такой ситуации кадр не будет доставлен, а отправитель сможет узнать об этом лишь на практике. После этого, кадр может быть послан снова и в конечном итоге доставлен. Однако на это потребуется ресурс центрального процессора, увеличится задержка доставки, в то время как при выполнении сетевых регламентаций задача решается на уровне сетевой карты. А в случае UDP-дейтограмм при передаче, к примеру, голосовых данных, это послужит утерей информации. Столкновение также реально, если машины соединены кабелем нулевой длины (из-за возможности одновременной передачи кадра). Это становится невозможным, если прием и передача осуществляется через разные скрученные пары, а повторители не применяются. Синхронизирующей причиной может быть передача кадра третьей машиной. В этом случае не важно, когда эти две машины захотели что-либо передать, — начнут они свою передачу по окончании передачи кадра третьей машиной.

Имеется возможность избежать проблемы роста вероятности столкновений в сети. Для этого можно удалить из сети повторители и построить сеть только на базе переключателей и маршрутизаторов, работающих в полнодуплексном режиме.

Для проверки работы физического уровня могут использоваться особые сетевые тестеры или анализаторы. Для поиска обрывов или коротких замыканий в кабелях в большей степени производительны доменные рефлектометры.

При потребности передать кадр активируется программа и счетчик попыток обнуляется. Анализ загруженности сегмента выполняется постоянно, и, если в течение 96 бит-данный сегмент свободен, то, если необходимо, начинается передача.

Если в процессе передачи зафиксировано столкновение, то посылается сигнал JAM и счетчик попыток увеличивается на 1. Затем вычисляется задержка следующей попытки, и выполнение алгоритма возвращается в первоначальную точку. Если число попыток превысит 16, сессия прекращается.

Метод CSMA/CD не удобен для решения некоторых задач управления реального масштаба времени, где нужно малое время реакции системы на внешнее воздействие, так как данный подход порождает неопределенность времени доступа к сети.

Первой возникла схема подключения к толстому пятидесятиомному коаксиальному кабелю (сегмент 1 на рис. 26; Z = 50 Ом) через трансивер и многожильный кабель типа AUI (наибольшая длина 50 м). Необходимо, чтобы кабельный сегмент был согласован с обеих сторон посредством терминаторов (50 Ом).

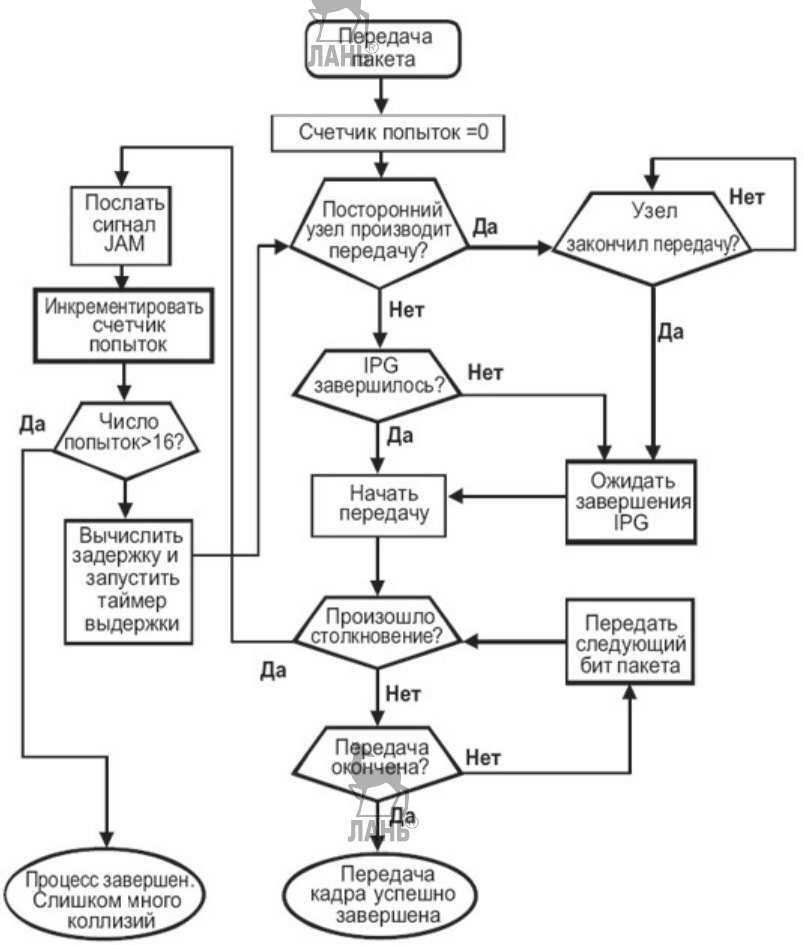


Рис. 25
Алгоритм доступа CSMA/CD

Типовые современные варианты сетевых сегментов обозначены на рисунке 26 номерами 3 и 4.

На основании вышесказанного можно предположить, что в будущем вместо витой пары предпочтение будет отдано оптоволоконным кабелям (когда их цена будет низкой, а стандартное быстродействие станет больше или равно 1 Гбит/с).

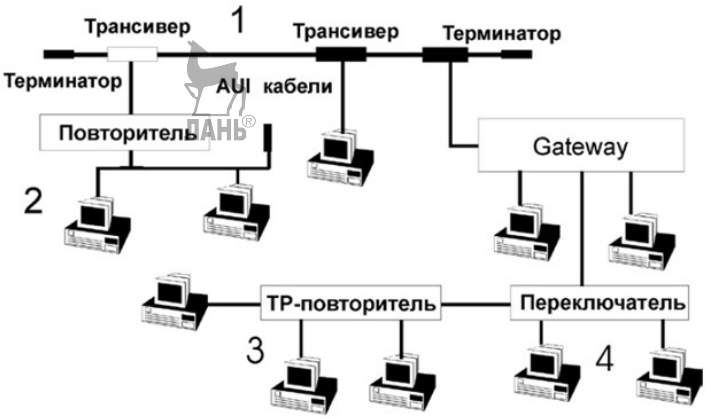


Рис. 26
Схема некоторых возможных вариантов подключения рабочих станций к Ethernet

В настоящий момент существуют следующие модификации сети Ethernet:
– **Fast Ethernet (FE)** с частотой 100 Мбит/с;
– **Gigabit Ethernet (GE)** с частотой 1000 Мбит/с;
– **Ethernet 10G (10GE)** с частотой 10 Гбит/с;
– **Ethernet 100G**, с частотой 10 Гбит/с.

Оптоволоконные кабели дают преимущество при объединении сегментов сети, расположенных в различных зданиях. С их помощью увеличивается надежность сети из-за ослабления влияния электромагнитных наводок. Облегчается переход от 10- к 100-мегагерцному Ethernet, так как имеется возможность использовать уже имеющиеся оптоволоконные каналы, они будут работать и на 1 Гбит/с. На программном уровне 10-, 100- и 1000-МГц версии Ethernet неразличимы.

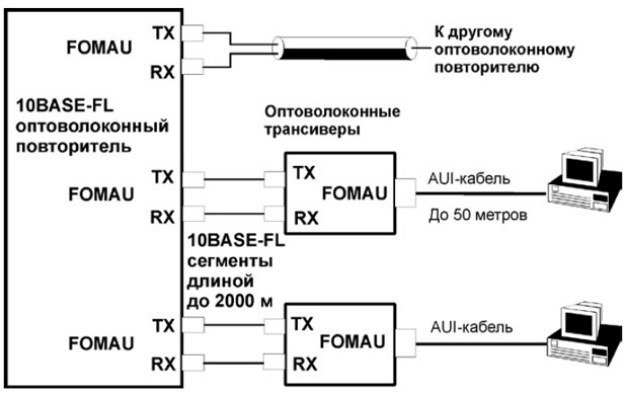


Рис. 27
Схема 10-мегагерцнового оптоволоконного Ethernet

Правила к параметрам оптоволоконных кабелей формируются документом EN 50173 (European Norm). Однако это не относится к топологии кабельных связей, так как в общем случае они зависят от применяемого протокола. В оптоволоконных системах требуются особые тестеры, которые могут рассчитывать потери света и отражения методом OTDR (рефлектометрия с применением метода временных доменов). В пассивной звездообразной схеме длины оптоволоконных сегментов могут достигать 500 метров, а число подключенных ЭВМ — 33. Для того чтобы передать сигналы на расстояния до 2 км, применяются многомодовые волокна (диаметр ядра 62,5 микрон, клэдинг 125 микрон). При ослаблении сигнала в кабельном сегменте не более, чем до 12,5 дБ, длина волны излучения равна 850 (или 1350) нанометров при том, что обычный кабель имеет ослабление 4–5 дБ/км. Оптические разъемы должны отвечать правилам стандарта ISO/IEC BFOC/2,5 и вводить ослабление не более 0,5–2,0 дБ. В логическом сегменте число используемых MAU не должно превышать двух.

На рисунке 28 отмечено, что соединение повторителя с FOMAU (Fiber Optic Medium Attachment Unit) является дуплексным. Те же возможности обеспечивают многие современные переключатели. Практическое удвоение скорости обмена и исключить столкновения пакетов может предоставить полнодуплексное подключение оборудования. Кабель AUI возможно заменить скрученной парой, что является более современным.

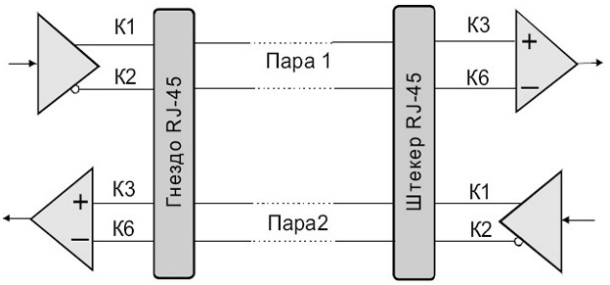


Рис. 28
Схема реализации полнодуплексного канала Ethernet

На практике во время реализации локальной сети обычно обнаруживаются проблемы защиты и заземления. В данной проблеме можно выделить три аспекта: безопасность работников, работающих с ЭВМ и сетевым оборудованием, устойчивость к внешним наводкам и помехам, а также безопасность самого сетевого оборудования. Безопасность работников обуславливается тем, что все, до чего может дотронуться человек, должно иметь одинаковые потенциалы и в любом случае разница потенциалов не должна превышать 50 вольт. Стоит отметить, что нужно избегать применения экранированных и неэкранированных скрученных пар в границах одной системы.

Сети Ethernet могут быть не только локальными, но и общегородскими и даже междугородними. Это осуществляется за счет оптоволоконных переключателей.

Несмотря на свою популярность, Ethernet — отнюдь не единственное устройство, позволяющее создавать локальную сеть. Linux имеет возможность

поддерживать различные типы сетей. Число драйверов для устройств, отличных от Ethernet, невелико, но это не означает, что средства для организации соответствующих сетей разработаны недостаточно хорошо. Ниже приведены аналогичные опции, присутствующие в меню Network Device Support.

• **LocalTalk.** Для компьютеров Macintosh компания Apple разработала сетевую технологию, включающую протоколы как программного (AppleTalk), так и аппаратного (LocalTalk) уровня. Для взаимодействия с сетями LocalTalk были разработаны компьютеры x86; некоторые из них поддерживает операционная система Linux. Соответствующие опции находятся в меню AppleTalk Devices. Как ни странно, версия ОС Linux, разработанная для Macintosh, не поддерживает LocalTalk. На данный момент максимальная скорость обмена данными в сети LocalTalk составляет 230,4 kbit/s.

• **Token Ring.** В течение многих лет технология Token Ring, которую разработала компания IBM, была главным конкурентом Ethernet, однако, начиная с 1990 г., преимущество Ethernet стало очевидным. Большое количество карт Token Ring поддерживают скорость передачи до 16 Мбод, но в настоящее время появились модели со скоростью более 100 Мбод. Максимальное расстояние между устройствами в сети Token Ring составляет 150–300 метров. Средства поддержки устройств Token Ring сосредоточены в подменю Token Ring Devices меню Network Device Support.

• **ARCnet.** Это сетевая технология, которая в основном используется в разделенных целях, например, для сбора данных в производственных условиях или для подключения охранных устройств. Устройства ARCnet обеспечивают скорость обмена от 19 Кбод до 10 Мбод. Соединение компьютеров осуществляется с помощью оптоволоконного кабеля, витой пары или коаксиального кабеля. Опции поддержки ARCnet находятся в подменю ARCnet Devices. Помимо драйвера устройства, необходимо включить драйвер, предназначенный для поддержки формата ARCnet-пакетов (RFC 1051 или RFC 1201).

• **HIPPI. HIPPI** (High Performance Parallel Interface — высокопроизводительный параллельный интерфейс) обеспечивает скорость обмена данными 850 или 1500 Кбод. При соединении с помощью витой пары максимальная дальность составляет до 30 метров, многорежимное оптоволоконное соединение обеспечивает дальность до 400 метров, а однорежимное оптоволоконное соединение — до 10 километров. Ядро 2.4.26 поддерживает единственное устройство HIPPI Essential RoadRunner. Нужно заметить, что драйвер данного устройства считается экспериментальным.

• **FDDI** и **CDDI. FDDI** (Fiber Distributed Data Interface — оптоволоконный интерфейс распределенных данных) и CDDI (Copper Distributed Data Interface — «медный» интерфейс распределенных данных) предназначены для создания сетей со скоростью обмена информацией порядка 100 Мбод. Преимущество FDDI перед 10 мегабитовой Ethernet состоит в том, что данная технология обеспечивает связь на расстоянии до 2 километров. Следует заметить, что гигабитовая Ethernet с передачей данных по оптоволоконному кабелю обеспечивает дальность до 5 километров. Для того чтобы опции ядра 2.4.26,

предназначенные для поддержки FDDI/CDDI, стали доступны, надо установить опцию FDDI Driver Support.

• **Fiber Channel**. Данный тип сетевого интерфейса поддерживает как соединение оптоволоконным кабелем, так и соединение с помощью обычного кабеля и обеспечивает скорость передачи данных 125–1084 Мбод. При использовании оптоволоконного кабеля максимальная дальность составляет до 10 километров. Ядро поддерживает единственное устройство Fiber Channel Interphase 5436 Tachyon.

Некоторые из описанных выше сетевых сред, например Token Ring, используются для создания локальных сетей, компоненты которых размещаются в одном здании либо в нескольких зданиях, расположенных рядом. Другие, например FDDI и HIPPI, чаще применяются для организации соединения между компьютерами, расположенными на большом расстоянии, например, находящимися в различных зданиях на территории университетского городка. Поддержка этих технологий ОС Linux означает, что компьютер под управлением Linux может выступать в роли маршрутизатора, связывающего между собой различные типы сетей.

Интернет

Интернет стал развитием ARPANET, экспериментальной сети с коммутацией пакетов, который профинансировало Агентство по перспективным исследовательским проектам МО США. Эта сеть была запущена в декабре 1968 г. и связала три компьютера в Калифорнии и один в штате Юта. Она была разработана во время холодной войны и рассматривалась как сеть, имеющая высокую отказоустойчивость и способность к продолжению передачи информации военного характера даже в условиях нанесения точных ядерных ударов по нескольким составляющим сети, за счет автоматического перенаправления трафика в обход вышедших из строя машин.

В 1970-х гг. ARPANET быстро разрасталась, охватив со временем сотни компьютеров. Когда к ней были подключены сеть пакетной радиосвязи, спутниковая сеть и, в конечном счете, тысячи сетей Ethernet, это привело к объединению сетей, известному теперь как Интернет.

Интернет состоит из двух типов компьютеров: хостов и маршрутизаторов. Хостами (hosts) являются персональные ноутбуки, компьютеры, гаджеты, универсальные машины, серверы и другие компьютеры, владельцами которых являются компании и частные лица, пожелавшие подключится к Интернету. Маршрутизаторы (routers) — это специализированные коммутирующие компьютеры, принимающие входящие пакеты по одной или нескольким линиям и отправляющие их по их маршруту по одной из многих входящих линий. Маршрутизатор похож на коммутатор, показанный ранее на рисунке, но имеет также и некоторые отличия, которые нас в данном случае не интересуют. Маршрутизаторы соединены друг с другом в большие сети, в которых каждый маршрутизатор связан электрическими или волоконно-оптическими кабелями со многими другими маршрутизаторами и хостами. Большие национальные или всемирные

сети маршрутизаторов управляются поставщиками услуг Интернета и телефонными компаниями.

Региональные сети и маршрутизаторы поставщиков услуг Интернета подключаются к магистральным маршрутизаторам с помощью оптоволоконных кабелей со средней пропускной способностью. В свою очередь у корпоративных сетей Ethernet имеются маршрутизаторы, подключенные к маршрутизаторам региональной сети. Маршрутизаторы, имеющиеся у поставщиков услуг Интернета, подключены к группам модемов, используемым клиентами этих поставщиков. Таким образом, каждый хост Интернета имеет как минимум один путь, а зачастую и несколько путей к любому другому хосту.

Весь поток данных в Интернете отправляется в форме пакетов. Каждый пакет несет в себе адрес назначения, который используется для маршрутизации. Когда пакет попадает в маршрутизатор, тот извлекает адрес назначения и ищет одну из его частей в таблице, чтобы определить, по какой из выходящих линий, а значит, к какому маршрутизатору его отправить. Эта процедура повторяется до тех пор, пока пакет не дойдет до хоста назначения. Таблицы маршрутизации имеют высокодинамичный характер и постоянно обновляются по мере того, как маршрутизаторы и линии связи отключаются и подключаются вновь, и по мере изменения условий передачи данных.

Оборудование, используемое в сети Интернет, можно разделить на три категории:

1) **Клиент** — рабочая станция (персональный компьютер), ноутбук, телефон, телевизор... То есть любое устройство, которое может сформировать по команде пользователя или автоматически запрос на получение информации из сети, получить ответ на свой запрос и отобразить полученную информацию в вид, доступный для потребителя информации.

2) **Сервер** — компьютер, выделенный из группы персональных компьютеров (или рабочих станций) для выполнения какой-либо сервисной задачи без непосредственного участия человека. Сервер и рабочая станция могут иметь одинаковую аппаратную конфигурацию, так как различаются лишь по участию в своей работе человека за консолью.

3) **Сетевое Оборудование** — оборудование, которое обеспечивает передачу информации по сети между Клиентами и Серверами, и собственно сами каналы связи.

Архитектура и функционирование dns

Чтобы организовать связь между компьютерами, используются IP-адреса — уникальные сетевые адреса узлов в компьютерной сети, построенной по протоколу IP, которые имеют вид: 192.168.0.3.

Зачастую пользователь знает только имя хостинга и ничего не знает о его адресе. Следовательно конечному пользователю или приложению нужно каким-либо способом получить адреса по имени хостинга. В этом ему помогает DNS — Domain Name System, которая выполняет функцию записной книжки для IP-адресов.

DNS — система доменных имен, которая выполняет функцию передачи имени в IP-адрес, который нужен для организации связи между компьютерами в сети.

У любого сайта есть свое доменное имя (к примеру, www.twich.tv). DNS связывает это имя с IP-адресом сервера, на котором находится сайт. При вводе какого-либо домена в адресную строку он автоматически преобразовывается в IP-адрес, с помощью которого уже и осуществляется связь.

В отдельной изолированной сети преобразование проводится с помощью общей таблицы, а различные системы имеют возможность получать информацию, загружая содержимое таблицы на свои информационные носители.

В самом начале существования Интернета использовалась централизованная система, созданная Министерством обороны США, которая позволяла преобразовывать имена в адреса другим системам, периодически копировавшим содержимое главной системы. Но по ходу развития Интернета этот метод перестал использоваться, так как потерял свою эффективность.

Архитектура DNS

DNS — это распределенная, иерархическая система серверов имен. То есть нет базы данных, которая хранила бы сведения обо всех IP-адресах и именах, которые им соответствуют. Большое множество баз данных, содержащих сведения о каждом домене, составляют собой DNS.

Рассмотрим иерархию DNS с помощью доменного имени, например, имени веб-сайта www.twich.tv. Оно содержит в себе три части. Каждая часть разделяется точкой. Однако полное доменное имя должно заканчиваться точкой, которая обозначает корневую зону DNS и является отдельной частью имени.

www — имя вэб-сервера и соответствующие ему IP адреса.

twich — домен второго уровня twich, имеющий сведения обо всех поддоменах, именах серверов, зарегистрированных в этом домене, например, www.twich.tv.

tv — домен первого уровня tv содержит в себе сведения обо всех поддоменах, которые зарегистрированы в ней. Также из него можно получить адреса серверов, которые предоставляют нам дополнительные сведения о содержимом поддоменов.

. — корневая зона, содержит в себе сведения обо всех поддоменах: net, com, org, ru, su, и т. д. Точнее, сведения о серверах, которые обслуживают эти домены.

Передача имени www.twich.tv в IP-адреса, которые соответствуют ему, происходит в несколько шагов. Во-первых, запрашиваются серверы, которые обслуживают корневую зону. У серверов нет никакой информации о существовании домена twich и тем более адреса www.twich.tv. Через них мы можем узнать, каким способом можно связаться с серверами, которые обслуживают домен другого уровня — tv. Также можно узнать адреса серверов домена twich, которые и отвечают на запрос о IP-адресе сервера www.twich.tv.

С помощью такой архитектуры системы доменных имен можно распределить всю нагрузку между администраторами доменов. Обязательствами ад-

министраторов являются поддержка хорошей производительности функции ответа на запросы к зоне, обеспечение уникальности имен в рамках зоны, уведомление администратора родительской зоны о каких-либо изменениях серверов, которые обслуживают зону. Уже более двадцати лет такая архитектура DNS обеспечивает долгую жизнь системы, а также ее развитие.

Корневой уровень DNS

Корневые серверы DNS являются главной составляющей системы, так как поддерживают доступ к корневой зоне DNS.

Корневая зона имеет сведения обо всех доменах наивысшего уровня: национальные домены (например, .ru), общего назначения (например, .com) и спонсированные домены (например, .museum).

Эта информация используется для того, чтобы узнать на какие серверы DNS следует послать последующий запрос для того, чтобы продолжить разрешение полного доменного имени. То есть, любой новый (то есть который не сохранен в кэше клиента) запрос имеет в начале обращение к новому серверу.

Самый первый запрос, который реализуется клиентом, производится по IP-адресу сервера, а не по его имени, потому что для передачи имени в IP-адрес нужна система доменных имен, для начала работы с которой нужен доступ к корневому серверу. Это также является одной из главных отличительных особенностей DNS.

Так как изначальную информацию о корневых серверах и их адресах невозможно получить из системы доменных имен, то для хранения этой информации предназначен специальный файл hints, который находится у клиента и обычно его можно получить вместе с программным обеспечением (операционная система, программное обеспечение для DNS и так далее).

Функционирование DNS

Как уже говорилось, DNS-сервер обеспечивает разрешение имен в сетях, основанных на протоколах TCP/IP. То есть дает возможность пользователям использовать для идентификации удаленных узлов наименования, а не IP-адреса, записанные в числовом виде.

Компьютер клиента отправляет имя удаленного узла DNS-серверу, а тот, в свою очередь, возвращает компьютеру соответствующий IP-адрес. После этого компьютер клиента имеет возможность отправить сообщения непосредственно на IP-адрес удаленного узла. Если в базе данных нет информации, которая соответствует удаленному узлу, то он возвратит клиенту адрес DNS-сервера, который, вероятно, имеет сведения об этом удаленном узле, или самостоятельно запросит эти сведения у другого DNS-сервера.

До тех пор, пока клиентский компьютер не получит нужный ему IP-адрес или если не окажется, что указанное имя не относится к узлу в конкретном пространстве имен DNS, процесс продолжает выполняться рекурсивно.

В действительности DNS-система не зависит от протокола сетевого уровня, то есть она реализуется не только TCP/IP. Но существует еще ряд других функций. Также с помощью DNS можно приобрести нижеследующие сведения:

– IP-адрес хоста;

– доменное наименование хоста;

– псевдонимы хоста, тип центрального процессора и операционной системы хоста;

– сетевые протоколы, которые поддерживает хост;

– почтовый шлюз;

– почтовый ящик;

– почтовую группу;

– IP-адрес и доменное имя сервера имен доменов.

DNS и Active Directory

Служба DNS-сервер встроена в структуру и реализацию службы каталогов **Active Directory**. Служба каталогов **Active Directory** представляет собой инструментом организации, управления и выбора местоположения ресурсов в сети на уровне предприятия.

После установки Active Directory на сервер роль этого сервера возрастает до роли контроллера указанного домена. После завершения процесса пользователь получает предложение написать доменное имя DNS для домена Active Directory, для которого, в свою очередь, происходит присоединение и повышение сервера.

В случае если в этом процессе удостоверяющий DNS-сервер для указанного домена отсутствует в сети или не имеет возможности для поддержки протокола динамического обновления DNS, то пользователю предлагается установить DNS-сервер. Это необходимо, так как DNS-серверу необходимо найти этот сервер или другие контроллеры домена для рядовых серверов домена Active Directory.

После того, как все компоненты Active Directory установятся, существует возможность сохранения и репликации зон при работе с DNS-сервером на «свежем» контроллере домена:

1. Стандартное сохранение зоны, которое происходит при помощи текстового файла.

При выборе этого способа зоны располагаются в файлах, имеющих расширение DNS, которые сохраняются в папке «системный_корневой_каталог\System32\Dns» на любом компьтере, на котором работает DNS-сервер. Наименование файла зоны соответствует имени, выбранному для зоны во время ее создания. К примеру, exam.micro.com.dns, если наименованием зоны является «exam.micro.com».

2. Сохранение зон с помощью базы данных Active Directory.

В этом случае зоны располагаются в дереве Active Directory под разделом каталога домена или программы. Каждая зона, встроенная в службу каталогов, располагается в контейнере dnsZone, который идентифицируется по выбранному пользователем имени во время ее создания.

Сервисы DNS в AIX

Все сервисы системы доменного именования полностью возможно реализовать в операционной системе AIX.

AIX является UNIX-подобная операционная система компании IBM.

Типы серверов имен, поддерживаемые в операционной системе AIX.

1. Первичный сервер имен.
2. Вторичный сервер имен.
3. Сервер, предназначенный для кэширования.
4. Сервер Forwarder (внутренний DNS-сервер).
5. Удаленный сервер.

Gethostbyaddr() и **Gethostbyname**, клиенты DNS в AIX, пытаются определить имена, используя следующую последовательность действий:

Если файл **/et/resolv.conf** отсутствует, клиент DNS думает, что в этой сети применяется плоская система именования. В этом случае используется для определения имен файл **/et/hosts**.

Если же файл существует, клиент DNS думает, что локальная сеть является доменной сетью и пытается применить для определения имени источники, указанные ниже:

1. Сервер DNS.
2. Локальный файл **/et/hosts**.

Все функции сервера имен в операционной системе выполняет демон, который носит название named. Он регулируется с помощью AIX SRC. Демон может запускаться автоматически при любой перезагрузке системы, при помощи команды **smit stnamed** или при редактировании файла **rc.tcpip**, то есть удалив комментарий в **строке #start /etc/named «$src_running»**. Также демон named выполняется при использовании такой команды, как: **startsrc -s named**.

Хостинг AIX настраивается для использования сервера имен при помощи следующих шагов:

1. Создание файла **/etc/resolv.conf**, записав в него имя домена и адреса до 16-ти серверов имен. К примеру:

– domain komtek.dp.ua;

– nameserver 192.168.1.190;

– nameserver 192.168.1.191.

Порядок вызова серверов определяется порядком их записей имен этих серверов: первый сервер вызывается первым, далее вызывается второй и т. д.

Как правило первым записывают самый ближний вторичный сервер имен данного домена, а после него — первичный, так как при таком порядке уменьшается нагрузка на первичный сервер.

В случае если первый в списке сервер не работает, то по истечении достаточно заметного промежутка времени (несколько секунд) клиент DNS осуществит обращение ко второму указанному в списке имен серверу.

2. Чтобы определить имя и тип локального демона **named** необходимо создать файл **/etc/name.boot**.

3. Для уточнения требуемой для демона информации необходимо создать файлы **/etc/name.***. Формат этих файлов должен полностью соответствовать формату записей стандартных ресурсов.

Для пользователя AIX/6000 доступны программы host и nslookup. В AIX/6000 также есть возможность использовать программу dig, которая нужна для отправки запросов к серверам имен.

Таким образом, чтобы снизить трудность использования сетевых ресурсов, DNS предоставляют возможность сопоставления понятного обычному пользователю имени компьютера или службы с другой информацией, связанной с этим именем, таким как IP-адрес. Наименования компьютера запоминаются легче, чем адреса, представленные в виде чисел, которые используются компьютерами при взаимодействии через сеть. Наибольшее количество людей имеют предпочтение к использованию для поиска почтового сервера или веб-сервера в сети понятные имена, которые записываются в виде: sales.fabrikam.com, а не IP-адрес, такой как 157.60.0.1. При вводе пользователем в приложении ясное, простое DNS-имя служба DNS разрешает имя в соответствующий ему численный адрес.

Почти все сети TCP/IP используют какую-либо форму дружественных имен для хост-систем и включают механизм для разрешения этих имен в IP-адреса. При соединении сети с Интернетом разрешение DNS-имен становится уже необходимостью.

Обзор стандартной модели сетевых протоколов iso

При взаимодействии двух компьютеров использование стандартных моделей не обязательно, но на практике при реализации сетей стараются использовать стандартные протоколы. Разработкой занялась Международная Организация Стандартов (International Standards Organization) ISO. Эта организация занялась разработкой модели, которая разграничивала уровни во взаимодействии систем, присваивала стандартные имена и давала указания о том, какая работа должна выполняться каждым уровнем. Этой модели дали название модель взаимодействия открытых систем или модель Open System Interconnection.

При разработке этой модели ее уровни решили поделить на семь частей. В модели OSI уровни сотрудничают с частями взаимодействия. Поэтому решение возникающих проблем можно было решить в зависимости от рода проблемы. Решение принималось в зависимости от того, какой уровень выдает ошибку. Помимо того что решение делилось на семь частных проблем, уровни могли и сотрудничать с выше- и нижестоящими уровнями.

Первый уровень (Физический). На этом уровне главным является физическая среда для передачи битов и сама передача битов. Для этого может быть использован коаксиальный или оптоволоконный кабель. Главными критериями этих кабелей является их волновое сопротивление, полоса пропускания или помехозащитность. На этой ступени большую роль уделяют и характеристикам электрического сигнала. Типы кодирования и скорость передачи всегда будут являться основными характеристиками, но есть и вторичные характеристики. Например, уровни напряжения, ток передаваемого сигнала и фронты импульса.

Конечно же эти характеристики прямо влияют на скорость и типы кодирования. Но часто они являются одинаковыми, поэтому они являются вторичными. Благодаря этому можно стандартизировать все разъемы и дать им определенную функцию в зависимости от надобности.

Рис. 29
Схема стандартной модели ISO

Стек протоколов ISO включает в себя:
– **IS-IS —** взаимодействие промежуточных систем;
– **ES-IS —** взаимодействие оконечной и промежуточной системы;
– **ISO-IP —** протокол межсетевого взаимодействия;
– **ISO-TP —** транспортный протокол;
– **ISO-SP —** сеансовый протокол;
– **PP —** протокол представления;
– **CCITT X.400 —** протокол обработки почтовых сообщений CCITT.

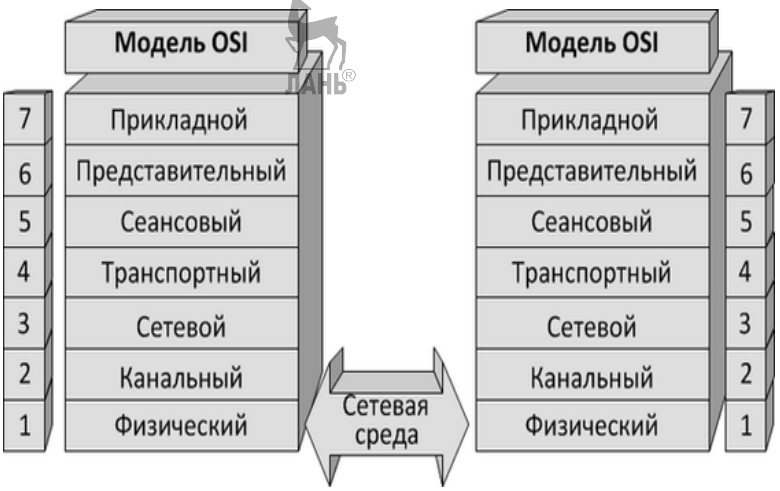


Рис. 30
Модель взаимодействия открытых систем ISO/OSI

Функции первого уровня (физического) используют на всех устройствах, которые подключены к сети. Существует устройство, которое работает только на физическом уровне и оно называется повторитель. Если рассматривать компьютер, то физическим уровнем будет являться сетевой адаптер.

Наглядным примером является технология Ethernet. Эта технология стандартизирует кабеля. Спецификация 10BASE подразумевает неэканированную витую пару, где разъем RJ-45 (100 ом) и длина сегмента кабеля достигает 100 м и т. д.

Второй уровень (Канальный). На первом уровне главным является простая пересылка битов по каналу. Физический уровень занимается взаимодействием сети, где линии связи могут быть использованы разными парами компьютеров, но не учитывает, свободна ли физическая среда для передачи данных. Еще одной важной задачей является обнаружение ошибок и их коррекция. Для этого используется механизм группировки битов в наборы, которые называются кадрами. Этот уровень обеспечивает корректность передачи кадров путем помещения специального набора бит в начало и конец кадра для того, чтобы выделить его и посчитать контрольную сумму. Подсчет контрольной суммы происходит путем сложения байтов кадра специальным способом и добавлением контрольной суммы. При получении кадра получателем происходит процесс подсчета и сравнения полученной контрольной суммы с суммой, которая находится в кадре. Если при сравнении окажется, что результат получателя и отправителя равны, то результат принимается, если же нет, то выдается ошибка.

При рассмотрении локальных сетей эти протоколы используются компьютерами, маршрутизаторами и т. д. Если рассматривать компьютер, то канальную работу выполняют сетевые адаптеры и драйверы.

Третий уровень (Сетевой). Основная функция этого уровня в том, что он образовывает транспортную систему, включающую в себя несколько различных сетей, с разными принципами передачи информации узлам. Для более удобного рассмотрения этого уровня лучше использовать локальную сеть. Для того чтобы канальный уровень мог передавать сообщения между узлами, нужно придерживаться типовой топологии. С этим строгим ограничением не получится строить развитые сети, к примеру, сети, включающие в себя еще несколько сетей компании в единую сеть, или сети, включающие в себя запасные сети для того, чтобы избежать перегрузки. Для того чтобы соблюдать топологию и простоту, и при этом можно было создавать большие сети, начали использовать дополнительный уровень. И тут мы введем более узкое понятие «сеть». Под этим понятием понимается множество компьютеров, которые соединены с использованием определенной типологии и используются для передачи данных определенного протокола канального уровня. Из этого следует, что в сети за доставку отвечает канальный уровень, а в их взаимодействии — сетевой уровень.

Пакеты — это сообщения, которые используются при передаче на сетевом уровне. Когда начинается передача пакетов на сетевом уровне, используется специальный адрес, называющийся «номером сети». Адрес того, кто будет получать сообщения, должен состоять из номера сети и адреса, который занимает этот компьютер в этой сети. Для соединения различных сетей в единое це-

лое используют маршрутизатор. При передаче сообщений между получателем и отправителем, находящихся в разных сетях, необходимо несколько транзитных операций между сетями, то есть маршрутизаторы, соединенные последовательно, будут представлять маршрут, через который проходит пакет.

Основная задача, которая должна выполняться на этом уровне, — это задача маршрутизации. Задача маршрутизации сложна тем, что короткий маршрут не всегда лучший. Почти всегда главным при выборе правильного и рационального маршрута бывает время передачи информации. Время зависит от многих физических критериев, например, пропускная способность каналов, которая в зависимости от различных условий может меняться. У маршрутизатора есть много алгоритмов, по которому они выстраивают маршрут, это может быть и ориентация на то, как изменяется нагрузка, или маршрутизаторы, выявляющие по общей статистике правильный маршрут. Конечно же, маршрут может определяться и по другим показателям, например, надежность передач сигнала. Обычно используются 2 вида протоколов. Первый, как правило, определяет правила передачи пакетов от одного маршрутизатора к другому или между узлов. Когда упоминают о протоколах этого уровня, имеют в виду этот тип. Протокол обмена маршрутной информацией — это другой вид протоколов сетевого уровня. Благодаря этим протоколам маршрутизаторы собирают важную информацию о межсетевых соединениях.

Все эти протоколы реализуются с помощью операционной системы или сторонних программ.

Четвертый уровень (Транспортный). В работе приложений одной из самых главных задач является то, чтобы информация не была искажена или вообще утеряна. У глобальных компаний присутствуют свои средства получения достоверной информации из каналов, но не очень большим приложениям не рационально это использовать, и они используют транспортный уровень. Суть этого уровня в том, чтобы снабдить приложения такой степенью надежности передачи информации, которая им необходима. Модель может предоставлять в своем классе пять очень важных функций. Функции предоставляемых услуг отличаются своими качествами.

Главными являются:

- Срочность.
- Возможности восстанавливать прерванную связь.
- Возможность обнаружить и исправлять ошибки передачи.

При выборе услуги класса транспортного уровня, с одной стороны, выбор надежности будет стоять за самим приложением и протоколов выше, чем транспортный уровень, а с другой стороны, обеспечится ли надежность самой транспортной системы при ее выборе. Например, при использовании дорогих и высококачественных каналов снимается необходимость использования сложных систем, потому что вероятность возникновения ошибок очень мала. Если же наши каналы не очень новые или обладают не очень хорошими показателями передачи сигнала, то, конечно, необходимо использовать контрольные суммы, добавлять суммы в кадры и проводить логический анализ всех сигналов. То есть необходимо использовать услуги транспортного уровня на максимуме.

Пятый уровень (Сеансовый). Сеансовый уровень является очень сложной системой, которую решаются использовать немногие, а если даже решатся, не факт, что они смогут ее реализовать. Этот уровень представляет систему для управления диалогом, то есть позволяет понять, какая система является активной, а какая используется для синхронизации. Средства синхронизации позволяют добавлять контрольные точки в сигналы, чтобы в случае некорректной работы можно было вернуть сигнал к значению контрольной точки. Обычно это используется для очень длинных сигналов.

Шестой уровень (Представления). Главная обязанность этого уровня заключается в том, что он обеспечивает надежность того, что прикладная информация, полученная одной машиной, будет понята другой. Если машины разных типов, то уровень представления может переводить сигнал в общий формат, и также, если принимает, он может перевести информацию из общего на свой язык. Это помогает избежать ошибок в синтаксисе, которые могут возникнуть в процессе передачи. Еще одно достоинство этого сервиса заключается в том, что эта среда является отличной для шифрования и дешифрования, при этом приборы в этой сети смогут читать информацию, но никто другой. Примером может являться протокол Secure Socket Layer (SSL), который работает на уровне представления. Он позволяет секретно обмениваться информацией через прикладные протоколы TCP.

Седьмой уровень (Прикладной). Этот уровень, можно сказать, наш интерфейс. С помощью прикладного уровня пользователь получает доступ к файлам, веб-страницам, используя множество протоколов для доступа пользователя, куда ему необходимо. Например, для того чтобы получить доступ к принтеру или приводу. Как правило, основной единицей, которой оперирует этот уровень, является «сообщение».

Эталонная модель TCP/IP

Эталонная модель TCP/IP по функционалу похожа на модель OSI, они обе основаны на концепции стека независимых протоколов. Но несмотря на это у TCP/IP и OSI имеются некоторые отличия. В OSI протоколы скрыты лучше, чем в TCP/IP, и, прежде всего, эти модели различаются количеством уровней. Так как в эталонной модели OSI 7 уровней, а в TCP/IP только 4. В обеих имеются транспортный, прикладной и межсетевой уровни, а все остальные уровни различаются. Все же, несмотря на все недостатки, эталонная модель OSI (кроме сеансового и уровня представления) показала себя с полезной точки зрения для компьютерных сетей, а вот протоколы OSI, наоборот, не получили популярность. С точки зрения TCP/IP верно обратное: модель практически не используется, в то время как протоколы очень популярны.

Протокол TCP (Transmission Control Protocol — протокол контроля передачи)

Протокол TCP работает в паре с протоколом IP для того, чтобы обеспечить надежную доставку. Протокол контроля передачи данных предлагает средства обеспечения надежности того, что разные дейтаграммы составляют

сообщения и собирают их в правильном порядке на принимающем аппарате и что некоторые пропущенные дейтаграммы будут посланы снова, пока они правильно не будут приняты. В первую очередь TCP обеспечивает надежность и безопасность сервиса виртуального контура связи между парами привязанных процессов на уровне ненадежных внутрисетевых пакетов. Здесь могут быть потери, уничтожения, дублирования, задержки или потери упорядоченности пакетов. Исходя из этого, можно сказать, что обеспечение безопасности может выполняться с помощью TCP, например, с помощью этого протокола можно ограничить доступ пользователю к каким-либо машинам. Этот протокол используется только для общей надежности. От этого можно привести пример: если дейтограмма отправлена через локальную сеть к удаленному главному аппарату, то промежуточные сети могут гарантировать доставку. Машина, которая посылает данные, не знает маршрут отправления дейтограммы. Надежность пути «источник-приемник» обеспечивается с помощью TCP на фоне ненадежности среды.

Исходя из этого, можно сказать, что TCP хорошо приспособлен к широкому разнообразию приложений многомашинных связей. Надежность обеспечивается посредством контроля суммы (коды обнаружения ошибок) последовательных чисел в заголовке TCP, прямого подтверждения получения данных и повторной передачи неподверженных данных.

TCP используют такие системы, как FTP; TELENET; X-Window. Для исполнения TCP требуется большая производительность системы и большая пропускная способность сети.

Задачи TCP:

1) мультиплексирование данных между приложениями и сетью;

2) проверка целостности полученных данных;

3) восстановление нарушенного порядка данных;

4) подтверждение успешного получения данных;

5) регулирование скорости передачи данных;

6) измерение временных характеристик;

7) координация повторной передачи данных, поврежденных или потерянных в процессе пересылки.

В состав TCP/IP входят протоколы, такие как: ARP; UDP; TELNET; ICMP; FTP; IP; TCP и др.

Межсетевой протокол IP (Internet Protocol)

Протокол IP вычисляет непривязанную пакетную доставку. Пакетная доставка привязывает одну или более пакетно-управляемые сети во всемирную сеть. Термин «непривязанную» означает, что принимающая и отправляющая аппараты не привязаны собой непосредственным контуром. Здесь дейтаграммы маршрутизируются через разные аппараты всемирной сети к локальной сети-получателю и получающему аппарату. От этого выходит, что сообщения разделяются на несколько дейтаграмм, которые отправляются раздельно. Исходя из этого, можно сказать, что непривязанная пакетная доставка надежна. Раздельные дейтаграммы после отправления могут не дойти до назначенного места,

и если и дойдут, то могут быть получены не по порядку, по которому были отправлены. TCP увеличивает надежность. Дейтаграммы состоят из заголовков информации и области данных. Этот заголовок используется для того, чтобы маршрутизировать дейтаграммы и для процесса дейтаграммы. Сами дейтаграммы могут быть разбиты на маленькие части, зависящие от физической возможности локальной сети, по которой они отправляются.

В то время как шлюз отправляет дейтаграмму к локальной сети, которая не в силе поместить дейтаграмму как целый пакет, она должна быть разделена на части, которые достаточно маленькие для отправки по этой сети. Заголовки элементов дейтаграммы содержат данные, которые необходимы для того, чтобы собрать элементы в законченную дейтаграмму. Элементы могут прийти не по порядку, по которому они были отправлены. Программный модуль, выполняющий IP-протокол на принимающем аппарате, должен собрать элементы в исходную форму. В то время если элементы потерялись, то полная дейтаграмма сбрасывается.

Протокол IP — это основной и популярный элемент технологии Internet. Благодаря усердной работе IETF протокол IP постоянно развивается. А с архитектурной точки зрения сейчас версия IP-Ipv4, но со временем более новая версия (Ipv6) постепенно вытеснит Ipv4. Ipv6 — это обновленная версия, разработанная на основе Ipv4. Пакеты Ipv6 передаются с использованием идентификатора типа 86DD вместо 0800 для IPv4. Благодаря Ipv6 используется более расширенное адресное пространство за счет использования 128-битовых адресов вместо IPv4 32-битовых.

Задачи IP:

1) адресация и маршрутизация;
2) фрагментация и повторная сборка;
3) выявление и исправление данных, поврежденных в процессе пересылки.

Протокол UDP (User Datagram Protocol — протокол пользовательских дейтаграмм)

В UDP предоставляются транспортные услуги к прикладным процессам, которые чуть отличаются от услуг IP. Протокол UDP отвечает за надежную доставку дейтаграмм. Примерами сетевых приложений, использующих UDP, являются NFS и SNMP.

Протокол NFS (Network File System — сетевая файловая система)

При работе с NFS пользователь может даже не узнать то, что он работает с сетью. Главная особенность NFS — это ее устойчивость. Например, при выключении сервера пользователь просто ждет, когда он заново включится и продолжит работу с того места, где он остановился. В версии протокола NFS 1 и 2 в качестве коренного транспортного протокола используется UDP. Начиная с версии 3, в качестве транспортного протокола используется TCP.

Протокол SNMP (Simple Network Management Protocol — простой протокол управления сетью)

SNMP также работает на базе UDP и используется для сетевых управляющих станций. Это позволяет управляющим станциям собирать данные о положении дел в сети Internet. Протокол SNMP формирует данные, а их обрабатывание или объяснение остается на усмотрение управляющих станций. При помощи этого протокола управляющие станции собирают информацию о положении дел в сети Internet.

Протокол FTP (File Transfer Protocol — протокол передачи файлов)

Этот протокол является одним из самых старых протоколов Internet. FTP используется для эффективной и точной передачи данных. Обмен данными используется с помощью TCP.

Протокол Telnet

Telnet — это протокол эмуляции терминала. Он обычно используется в сети Internet и в таких сетях как основанные на TCP/IP. С помощью Telnet пользователь может удаленно управлять компьютером.

Протокол TFTP

TFTP создан для того, чтобы передавать файлы. Он используется при загрузке бездисковых систем. По сравнению с FTP, который использует TCP, протокол TFTP использует UDP. Так его сделали для того, чтобы он был простой и весил мало. Он может поместиться в постоянной памяти (ПЗУ). Чтобы несколько пользователей могли одновременно загружаться, TFTP может работать в нескольких формах. Из-за того, что UDP не может дать для клиента и сервера уникальное соединение, TFTP создает новый UDP-порт для каждого пользователя. Это дает возможность пользователю выдавать дейтаграммы, которые будут демультиплексированы UDP-модулем сервера.

Протокол SMTP

Этот протокол создан для общения между собой почтовых серверов. В модели OSI SMTP находится на уровне приложения. Протокол SMTP является очень экономичной системой. В начале SMTP поддерживал мало наборов команд и сервисов для приема и передачи данных. После него был разработан обновленный вариант ESMTP.

World Wide Web

Это на сегодняшний день самая новая и очень быстро развивающая служба Internet. С помощью нее можно собирать, распространять, изучать информацию. World Wide Web — это просто большой объем информации, которую пользователи могут использовать в своих делах.

Протокол HTTP (Hyper Text Transfer Protocol — «протокол передачи гипертекста»)

Название протокола хоть и называется передачей гипертекста, но на самом деле HTTP может использоваться для передачи почти любых данных по сети. Например: тексты, изображения, файлы и др. Популярность этого протокола связана с тем, что можно предавать любые файлы, использовать универсальные URL-адресации, а также работать в режиме онлайн. В этом протоколе информацию можно отправить в двух направлениях: и от клиента к серверу, и, наоборот, от сервера к клиенту. В целом HTTP нацелен только на передачу данных от клиента к серверу.

Обзор GPRS

GPRS является пакетом ориентированных услуг мобильной передачи данных 2G- и 3G-связи глобальной системы мобильной связи (GSM). GPRS был первоначально стандартизирован Европейским институтом телекоммуникационных стандартов (ETSI).

GPRS позволяет пользователю сети сотовой связи производить обмен данными с другими устройствами в сети GSM и с внешними сетями, в том числе, сети Интернет. GPRS предполагает тарификацию по объему переданной/полученной информации, а не по времени, проведенному онлайн.

Система GPRS является неотъемлемой частью GSM-подсистемы коммутации сети.

Предлагаемые услуги:

- **SMS**-сообщения и радиовещание.
- «Всегда» доступ в **Интернет**.
- **Служба обмена сообщениями мультимедиа (MMS)**.
- **Push-to-talk** (PTT) — полудуплексный стандарт голосовой связи с двусторонним радиоинтерфейсом и возможностью передачи сигнала одновременно только в одном направлении.
- Система мгновенных сообщений и сетевого присутствия **wireless village**.
- **Интернет-приложения** для интеллектуальных устройств через беспроводной протокол приложений (WAP).
- **PPP** (*англ.* Point-to-Point Protocol) — двухточечный протокол канального уровня.
- **(Data Link)** сетевой модели OSI.
- **Point-to-multipoint** — используется для указания многоточечного типа сети OSPF.
- **Спутниковый мониторинг транспорта**.

Для передачи SMS на основе GPRS скорость передачи SMS приблизительно может быть достигнута 30 SMS-сообщений в минуту. Это гораздо быстрее, чем с помощью обычного SMS с использованием GSM, чья скорость передачи SMS составляет около 6–10 SMS-сообщений в минуту.

Поддерживаемые протоколы

GPRS поддерживает следующие протоколы:

Интернет-протокол (IP), встроенный в мобильных браузерах, используется как IPv4, так и IPv6;

Point-to-point protocol — сетевой протокол канального уровня передачи кадров PPP. В основном используется xDSL-сервисами. Предоставляет дополнительные возможности (аутентификация, сжатие данных, шифрование).

X.25 — стандарт канального уровня сетевой модели OSI. Предназначался для организации WAN на основе телефонных сетей. Ориентирован на работу с установлением соединений. X.25 широко использовался как в корпоративных сетях, так и во всемирных специализированных сетях предоставления услуг, таких как SWIFT (банковская платежная система) и SITA (система информационного обслуживания воздушного транспорта).

Механизм работы

При использовании GPRS информация собирается в пакеты и передается через неиспользуемые на этот момент голосовые каналы. Передача осуществляется по двум назначениям: прием на конечное устройство и передача от него в сеть. Скорость в этом случае зависит от количества свободных каналов и работоспособности самих устройств по обработке данных. Ценность голосовой передачи либо передачи данных выбирается оператором связи. Возможность использования сходу нескольких каналов обеспечивает высокие скорости передачи данных. Есть разные классы GPRS, различающиеся скоростью передачи данных и возможностью совмещения передачи данных с одновременным голосовым вызовом. У современных телефонов, планшетов и смартфонов есть возможность использования до 4 каналов на прием информации и до 2 на отдачу. Более новые модели используют до 5 одновременно работающих каналов.

Мобильники с поддержкой GPRS делятся на три класса:

– **телефоны класса A** — возможность параллельной работы телефона как для разговора, так и для передачи информации;

– **телефоны класса B** — работает только один канал: либо голосовой, либо для передачи информации (GPRS). Если поступил голосовой вызов или SMS-сообщение, передача GPRS данных приостанавливается и возобновляется после окончания звонка или получения SMS-сообщения;

– **телефоны класса C** — поддерживают только прием/передачу данных.

Интеграция с Интернетом

GPRS по принципу работы схож с Интернетом: сведения разбиваются на пакеты и отправляются получателю (не обязательно одним и тем же маршрутом), где происходит их компоновка. При нахождении соединения каждому устройству назначается уникальный адрес, из-за чего происходит преобразование его в сервер. Протокол GPRS прозрачен для TCP/IP, поэтому интеграция GPRS с Интернетом незаметна конечному пользователю. Пакеты данных могут иметь формат IP или X.25, при этом не имеет значения, какие протоколы используются поверх IP. При использовании GPRS с телефонов, планшетов или

смартфонов они считаются, в данном случае, как клиенты внешней сети, и им присваивается определенный IP-адрес.

Принцип передачи информации

В своей работе она использует протокол IP и переносит пакеты между мобильными устройствами (сотовыми телефонами) и сетями пакетной передачи данных. Всем мобильным устройствам, подключающимся к сети, таким образом, присваиваются IP-адреса (динамические или статические, в зависимости от настроек конкретного сотового оператора), и пользователь получает возможность работать с ресурсами Интернета и корпоративных сетей. В приложении № 1 показана технология построения системы пакетной передачи данных в GPRS.

Пакетный коммутатор оператора сотовой связи выполняет функции обработки пакетной информации и преобразует фреймы GSM (порции сигналов) в форматы, используемые в Интернете. Подсистема базовых станций, в которую входят все базовые станции, работающие в системе GPRS, принимают радиосигналы от сотовых телефонов, которые выполняют функции GPRS-модемов.

Принцип передачи информации по GPRS. Весьма упрощенно это можно описать следующим образом: для абонентов работают радиоканалы на основе Использования определенных частотных диапазонов. — Абонент А1 ведет в сети телефонный сеанс в голосовом режиме. — Абонент А2 подключается к WAP-сайту Internet. — Абоненты А3 и А4 ведут в сети телефонные сеансы в режиме GPRS.

Пакетный коммутатор сотового оператора анализирует загрузку частотных диапазонов и в пустующие временные интервалы соответствующего частотного диапазона «вставляет» отдельные фреймы (пакеты радиосообщений). Таким образом, отдельные фреймы абонента А3 распределяются по радиоканалам соседних абонентов в свободные временные промежутки и в результате как бы освобождается один радиоканал. А раз канал незанятый, неразумно за него брать деньги. В этом варианте абонент А3 будет оплачивать только трафик (количество переданных/принятых мегабайт).

СОКЕТЫ (SOCKETS) В UNIX И ОСНОВЫ РАБОТЫ С НИМИ

Транспортный уровень. Протоколы TCP и UDP. TCP- и UDP-сокеты. Адресные пространства портов. Понятие encapsulation

К протоколам транспортного уровня относятся протоколы TCP и UDP. Протокол TCP реализует потоковую модель передачи информации, хотя в его основе, как и в основе протокола UDP, лежит обмен информацией через пакеты данных. Он представляет собой ориентированный на установление логической связи (connection-oriented), надежный (обеспечивающий проверку контрольных сумм, передачу подтверждения в случае правильного приема сообщения, повторную передачу пакета данных в случае неполучения подтверждения в течение определенного промежутка времени, правильную последовательность получения информации, полный контроль скорости передачи данных) дуплексный способ связи между процессами в сети. Протокол UDP, наоборот, является способом связи ненадежным, ориентированным на передачу сообщений (датаграмм). От протокола IP он отличается двумя основными чертами: использованием для проверки правильности принятого сообщения контрольной суммы, насчитанной по всему сообщению, и передачей информации не от узла сети к другому узлу, а от отправителя к получателю. Полный адрес удаленного процесса или промежуточного объекта для конкретного способа связи с точки зрения операционных систем определяется парой адресов: <числовой адрес компьютера в сети, локальный адрес>.

Такая пара получила название socket (гнездо, панель), так как по сути дела является виртуальным коммуникационным узлом (можно представить себе виртуальный разъем или ящик для приема/отправки писем), ведущим от объекта во внешний мир и наоборот. При непрямой адресации сами промежуточные объекты для организации взаимодействия процессов также именуются сокетами.

Поскольку уровень Internet семейства протоколов TCP/IP умеет доставлять информацию только от компьютера к компьютеру, данные, полученные с его помощью, должны содержать тип использованного протокола транспортного уровня и локальные адреса отправителя и получателя. И протокол TCP, и протокол UDP используют непрямую адресацию.

Для того чтобы избежать путаницы, в дальнейшем в лабораторной работе термин «сокет» будет употребляться только для обозначения самих промежуточных объектов, а полные адреса таких объектов будут называться адресами сокетов.

Для каждого транспортного протокола в стеке TCP/IP существуют собственные сокеты: UDP-сокеты и TCP-сокеты, имеющие различные адресные пространства своих локальных адресов — портов. В семействе протоколов TCP/IP адресные пространства портов представляют собой положительные значения целого 16-битового числа. Поэтому, говоря о локальном адресе сокета, часто будет использоваться термин «номер порта». Из различия адресных про-

странств портов следует, что порт 1111 TCP — это совсем не тот же самый локальный адрес, что и порт 1111 UDP.

Иерархическая система адресации, используемая в семействе протоколов TCP/IP включает в себя несколько уровней.

Физический пакет данных, передаваемый по сети, содержит физические адреса узлов сети (MAC-адреса) с указанием на то, какой протокол уровня Internet должен использоваться для обработки передаваемых данных (поскольку пользователя интересуют только данные, доставляемые затем на уровень приложений/процессов, то для него это всегда IP).

IP-пакет данных содержит 32-битовые IP-адреса компьютера-отправителя и компьютера-получателя и указание на то, какой вышележащий протокол (TCP, UDP или еще что-нибудь) должен использоваться для их дальнейшей обработки.

Служебная информация транспортных протоколов (UDP-заголовок к данным и TCP-заголовок к данным) должна содержать 16-битовые номера портов для сокета-отправителя и сокета-получателя.

Добавление необходимой информации к данным при переходе от верхних уровней семейства протоколов к нижним принято называть английским словом encapsulation (дословно: герметизация). На рисунке 31 приведена схема encapsulation при использовании протокола UDP на сети Ethernet.

Поскольку между MAC-адресами и IP-адресами существует взаимно однозначное соответствие, известное семейству протоколов TCP/IP, то фактически для полного задания адреса доставки и адреса отправления, необходимых для установления двусторонней связи, нужно указать пять параметров:

<транспортный протокол, IP-адрес отправителя, порт отправителя, IP-адрес получателя, порт получателя>.

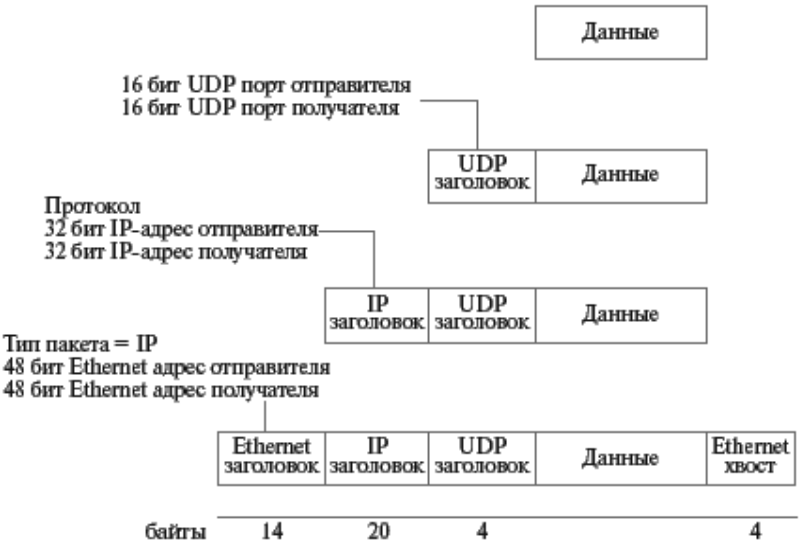


Рис. 31
Encapsulation для UDP-протокола на сети Ethernet

Использование модели клиент-сервер для взаимодействия удаленных процессов

Модель клиент-сервер, изначально предполагающая неравноправность взаимодействующих процессов, наиболее часто используется для организации сетевых приложений. Основные отличия процессов клиента и сервера применительно к удаленному взаимодействию.

Сервер, как правило, работает постоянно, на всем протяжении жизни приложения, а клиенты могут работать эпизодически.

Сервер ждет запроса от клиентов, инициатором же взаимодействия выступает клиент.

Как правило, клиент обращается к одному серверу за раз, в то время как к серверу могут одновременно поступить запросы от нескольких клиентов.

Клиент должен знать полный адрес сервера (его локальную и удаленную части) перед началом организации запроса (до начала общения), в то время как сервер может получить информацию о полном адресе клиента из пришедшего запроса (после начала общения).

И клиент, и сервер должны использовать один и тот же протокол транспортного уровня.

Неравноправность процессов в модели клиент-сервер накладывает свой отпечаток на программный интерфейс, используемый между уровнем приложений/процессов и транспортным уровнем.

Поступающие запросы сервер может обрабатывать последовательно — запрос за запросом — или параллельно, запуская для обработки каждого из них свой процесс или thread. Как правило, серверы, ориентированные на связь клиент-сервер с помощью установки логического соединения (TCP-протокол), ведут обработку запросов параллельно, а серверы, ориентированные на связь клиент-сервер без установления соединения (UDP-протокол), обрабатывают запросы последовательно.

Организация связи между удаленными процессами с помощью датаграмм

Наиболее простой для взаимодействия удаленных процессов является схема организации общения клиента и сервера с помощью датаграмм, т. е. использование протокола UDP.

С точки зрения обычного человека общение процессов посредством датаграмм напоминает общение людей в письмах. Каждое письмо представляет собой законченное сообщение, содержащее адрес получателя, адрес отправителя и указания, кто написал письмо и кто должен его получить. Письма могут теряться, доставляться в неправильном порядке, быть поврежденными в дороге и т. д.

Что в первую очередь должен сделать человек, проживающий в отдаленной местности, чтобы принимать и отправлять письма? Он должен изготовить почтовый ящик, который одновременно будет служить и для приема корреспонденции, и для ее отправки. Пришедшие письма почтальон будет помещать в этот ящик и забирать из него письма, подготовленные к отправке.

Изготовленный почтовый ящик нужно где-то прикрепить. Это может быть парадная дверь дома или вход со двора, изгородь, столб, дерево и т. п. Потенциально может быть изготовлено несколько почтовых ящиков и размещено в разных местах с тем, чтобы письма от различных адресатов прибывали в различные ящики. Этим ящикам будут соответствовать разные адреса: «Иванову, почтовый ящик на конюшне», «Иванову, почтовый ящик, что на дубе».

После закрепления ящика мы готовы к обмену корреспонденцией. Человек-клиент пишет письмо с запросом по заранее известному ему адресу человека-сервера и ждет получения ответного письма. После получения ответа он читает его и перерабатывает полученную информацию.

Человек-сервер изначально находится в состоянии ожидания запроса. Получив письмо, он читает текст запроса и определяет адрес отправителя. После обработки запроса он пишет ответ и отправляет его по обратному адресу, после чего начинает ждать следующего запроса.

Все эти модельные действия имеют аналоги при общении удаленных процессов по протоколу UDP.

Процесс-сервер должен сначала совершить подготовительные действия: создать UDP-сокет (изготовить почтовый ящик) и связать его с определенным номером порта и IP-адресом сетевого интерфейса (прикрепить почтовый ящик в определенном месте) — настроить адрес сокета. При этом сокет может быть привязан к конкретному сетевому интерфейсу (к конюшне, к дубу) или к компьютеру в целом, то есть в полном адресе сокета может быть либо указан IP-адрес конкретного сетевого интерфейса, либо дано указание операционной системе, что информация может поступить через любой сетевой интерфейс, имеющийся в наличии. После настройки адреса сокета операционная система начинает принимать сообщения, пришедшие на этот адрес и складывать их в сокет. Сервер дожидается поступления сообщения, читает его, определяет, от кого оно поступило и через какой сетевой интерфейс, обрабатывает полученную информацию и отправляет результат по обратному адресу. После чего процесс готов к приему новой информации от того же или другого клиента.

Процесс-клиент должен сначала совершить те же самые подготовительные действия: создать сокет и настроить его адрес. Затем он передает сообщение, указав, кому оно предназначено (IP-адрес сетевого интерфейса и номер порта сервера), ожидает от него ответа и продолжает свою деятельность.

Схематично эти действия выглядят так, как показано на рисунке 32. Каждому из них соответствует определенный системный вызов. Названия вызовов написаны справа от блоков соответствующих действий. Создание сокета производится с помощью системного вызова socket(). Для привязки созданного сокета к IP-адресу и номеру порта (настройка адреса) служит системный вызов bind(). Ожиданию получения информации, ее чтению и, при необходимости, определению адреса отправителя соответствует системный вызов recvfrom(). За отправку датаграммы отвечает системный вызов sendto().

Рис. 32
Схема взаимодействия клиента и сервера для протокола UDP

Сетевой порядок байт. Функции htons(), htonl(), ntohs(), ntohl()

Передача от одного вычислительного комплекса к другому символьной информации, как правило (когда один символ занимает один байт), не вызывает проблем. Однако для числовой информации ситуация усложняется.

Как известно, порядок байт в целых числах, представление которых занимает более одного байта, может быть для различных компьютеров неодинаковым. Есть вычислительные системы, в которых старший байт числа имеет меньший адрес, чем младший байт (**big-endian byte order**), а есть вычислительные системы, в которых старший байт числа имеет больший адрес, чем младший байт (**little-endian byte order**). При передаче целой числовой информации от машины, имеющей один порядок байт, к машине с другим порядком байт мы можем неправильно истолковать принятую информацию. Для того чтобы этого не произошло, было введено понятие сетевого порядка байт, т. е. порядка байт, в котором должна представляться целая числовая информация в процессе передачи ее по сети (на самом деле — это **big-endian byte order**). Целые числовые данные из представления, принятого на компьютере-отправителе, переводятся пользовательским процессом в сетевой порядок байт, в таком виде путешествуют по сети и переводятся в нужный порядок байт на машине-получателе процессом, которому они предназначены. Для перевода целых чисел из машинного представления в сетевое и обратно используется четыре функции: **htons(), htonl(), ntohs(), ntohl()**.

Функции преобразования порядка байт

Прототипы функций

```
#include <netinet/in.h>
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Описание функций

Функция **htonl** осуществляет перевод целого длинного числа из порядка байт, принятого на компьютере, в сетевой порядок байт.

Функция **htons** осуществляет перевод целого короткого числа из порядка байт, принятого на компьютере, в сетевой порядок байт.

Функция **ntohl** осуществляет перевод целого длинного числа из сетевого порядка байт в порядок байт, принятый на компьютере.

Функция **ntohs** осуществляет перевод целого короткого числа из сетевого порядка байт в порядок байт, принятый на компьютере.

В архитектуре компьютеров i80x86 принят порядок байт, при котором младшие байты целого числа имеют младшие адреса. При сетевом порядке байт, принятом в Internet, младшие адреса имеют старшие байты числа. Параметр у них — значение, которое необходимо конвертировать. Возвращаемое значение — то, что получается в результате конвертации. Направление конвертации определяется порядком букв **h (host)** и **n (network)** в названии функции, размер числа — последней буквой названия, то есть **htons** — это **host to network short, ntohl — network to host long**.

Для чисел с плавающей точкой все обстоит гораздо хуже. На разных машинах могут различаться не только порядок байт, но и форма представления такого числа. Простых функций для их корректной передачи по сети не существует. Если требуется обмениваться действительными данными, то либо это нужно делать на гомогенной сети, состоящей из одинаковых компьютеров, либо использовать символьные и целые данные для передачи действительных значений.

Функции преобразования IP-адресов inet_ntoa(), inet_aton()

Функция **inet_aton()** переводит символьный IP-адрес в числовое представление в сетевом порядке байт.

Функция возвращает 1, если в символьном виде записан правильный IP-адрес, и 0 в противном случае — для большинства системных вызовов и функций это нетипичная ситуация.

Обратите внимание на использование указателя на структуру **struct in_addr** в качестве одного из параметров данной функции. Эта структура используется для хранения IP-адресов в сетевом порядке байт. То, что используется структура, состоящая из одной переменной, а не сама 32-битовая переменная, сложилось исторически, и авторы в этом не виноваты.

Для обратного преобразования применяется функция **inet_ntoa().**

Функции преобразования IP-адресов

Прототипы функций

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
int inet_aton(const char *strptr, struct in_addr *addrptr);
char *inet_ntoa(struct in_addr *addrptr);
```

Описание функций

Функция **inet_aton** переводит символьный IP-адрес, расположенный по указателю strptr, в числовое представление в сетевом порядке байт и заносит его в структуру, расположенную по адресу **addrptr**. Функция возвращает значение 1, если в строке записан правильный IP-адрес, и значение 0 в противном случае. Структура типа **struct in_addr** используется для хранения IP-адресов в сетевом порядке байт и выглядит так:

```
struct in_addr {
    in_addr_t s_addr;
};
```

То, что используется адрес такой структуры, а не просто адрес переменной типа in_addr_t, сложилось исторически.

Функция **inet_ntoa** применяется для обратного преобразования. Числовое представление адреса в сетевом порядке байт должно быть занесено в структуру типа **struct in_addr**, адрес которой **addrptr** передается функции как аргумент. Функция возвращает указатель на строку, содержащую символьное представление адреса. Эта строка располагается в статическом буфере, при последующих вызовах ее новое содержимое заменяет старое содержимое.

Функция bzero

Прототип функции

```
#include <string.h>
void bzero(void *addr, int n);
```

Описание функции

Функция **bzero** заполняет первые n байт, начиная с адреса **addr,** нулевыми значениями. Функция ничего не возвращает.

Создание сокета. Системный вызов socket()

Для создания сокета в операционной системе служит системный вызов **socket().** Для транспортных протоколов семейства TCP/IP существует два вида сокетов: UDP-сокет — сокет для работы с датаграммами, и TCP-сокет — потоковый сокет. Однако понятие сокета не ограничивается рамками только этого семейства протоколов. Рассматриваемый интерфейс сетевых системных вызовов (**socket(), bind(), recvfrom(), sendto()** и т. д.) в операционной системе UNIX может применяться и для других стеков протоколов (и для протоколов, лежащих ниже транспортного уровня). При создании сокета необходимо точно специфицировать его тип. Эта спецификация производится с помощью трех параметров вызова socket(). Первый параметр указывает, к какому семейству протоколов относится создаваемый сокет, а второй и третий параметры определяют конкретный протокол внутри данного семейства. Второй параметр служит для

задания вида интерфейса работы с сокетом — будет это потоковый сокет, сокет для работы с датаграммами или какой-либо иной. Третий параметр указывает протокол для заданного типа интерфейса. В стеке протоколов TCP/IP существует только один протокол для потоковых сокетов — TCP и только один протокол для датаграммных сокетов — UDP, поэтому для транспортных протоколов TCP/IP третий параметр игнорируется.

В других стеках протоколов может быть несколько протоколов с одинаковым видом интерфейса, например, датаграммных, различающихся по степени надежности.

Для транспортных протоколов TCP/IP в качестве первого параметра указываем предопределенную константу AF_INET (Address family — Internet) или ее синоним PF_INET (Protocol family — Internet).

Второй параметр будет принимать предопределенные значения SOCK_STREAM для потоковых сокетов и SOCK_DGRAM — для датаграммных.

Поскольку третий параметр в данном случае не учитывается, в него мы будем подставлять значение 0. Ссылка на информацию о созданном сокете помещается в таблицу открытых файлов процесса подобно тому, как это делалось для pip'ов и FIFO. Системный вызов возвращает пользователю файловый дескриптор, соответствующий заполненному элементу таблицы, который далее мы будем называть дескриптором сокета. Такой способ хранения информации о сокете позволяет, во-первых, процессам-детям наследовать ее от процессов-родителей, а во-вторых, использовать для сокетов часть системных вызовов, которые уже знакомы нам по работе с pip'ами и FIFO: close(), read(), write().

Системный вызов для создания сокета

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol)
```

Описание системного вызова

Системный вызов socket служит для создания виртуального коммуникационного узла в операционной системе. Параметр domain определяет семейство протоколов, в рамках которого будет осуществляться передача информации. Мы рассмотрим только два таких семейства из нескольких существующих. Для них имеются предопределенные значения параметра:

PF_INET — для семейства протоколов TCP/IP;

PF_UNIX — для семейства внутренних протоколов UNIX, иначе называемого еще UNIX domain.

Параметр type определяет семантику обмена информацией: будет ли осуществляться связь через сообщения (datagrams), с помощью установления виртуального соединения или еще каким-либо способом. Будем пользоваться только двумя способами обмена информацией с предопределенными значениями для параметра type:

SOCK_STREAM — для связи с помощью установления виртуального соединения;

SOCK_DGRAM — для обмена информацией через сообщения.

Параметр protocol специфицирует конкретный протокол для выбранного семейства протоколов и способа обмена информацией. Он имеет значение только в том случае, когда таких протоколов существует несколько.

Возвращаемое значение

В случае успешного завершения системный вызов возвращает файловый дескриптор (значение большее или равное 0), который будет использоваться как ссылка на созданный коммуникационный узел при всех дальнейших сетевых вызовах. При возникновении какой-либо ошибки возвращается отрицательное значение.

Адреса сокетов. Настройка адреса сокета

Системный вызов bind()

Когда сокет создан, необходимо настроить его адрес. Для этого используется системный вызов **bind().** Первый параметр вызова должен содержать дескриптор сокета, для которого производится настройка адреса. Второй и третий параметры задают этот адрес.

Во втором параметре должен быть указатель на структуру **struct sockaddr**, содержащую удаленную и локальные части полного адреса.

Указатели типа **struct sockaddr *** встречаются во многих сетевых системных вызовах; они используются для передачи информации о том, к какому адресу привязан или должен быть привязан сокет. Рассмотрим этот тип данных подробнее. **Структура struct sockaddr** описана в файле ***<sys/socket.h>*** следующим образом:

```
struct sockaddr
{
    short sa_family;
    char sa_data[14];
};
```

Такой состав структуры обусловлен тем, что сетевые системные вызовы могут применяться для различных семейств протоколов, которые по-разному определяют адресные пространства для удаленных и локальных адресов сокета. По сути дела, этот тип данных представляет собой лишь общий шаблон для передачи системным вызовам структур данных, специфических для каждого семейства протоколов. Общим элементом этих структур остается только поле ***short sa_family*** (которое в разных структурах, естественно, может иметь разные имена, важно лишь, чтобы все они были одного типа и были первыми элементами своих структур) для описания семейства протоколов. Содержимое этого поля системный вызов анализирует для точного определения состава поступившей информации.

Для работы с семейством протоколов TCP/IP будем использовать адрес сокета следующего вида, описанного в файле ***<netinet/in.h>:***

```
struct sockaddr _in
{
    short sin_family;
    /* Избранное семейство протоколов — всегда AF_INET */
```

```
    unsigned short sin_port;
    /* 16-битовый номер порта в сетевом порядке байт */
    struct in_addr sin_addr;
    /* Адрес сетевого интерфейса */
    char sin_zero[8];
    /* Это поле не используется, но должно всегда быть заполнено
    нулями */
};
```

Первый элемент структуры — **sin_family** — задает семейство протоколов. В него необходимо занести предопределенную константу AF_INET.

Удаленная часть полного адреса — IP-адрес — содержится в структуре типа ***struct in_addr.***

Для указания номера порта предназначен элемент структуры ***sin_port***, в котором номер порта должен храниться в сетевом порядке байт. Существует два варианта задания номера порта: фиксированный порт по желанию пользователя и порт, который произвольно назначает операционная система. Первый вариант требует указания в качестве номера порта положительного заранее известного числа и для протокола UDP обычно используется при настройке адресов сокетов и при передаче информации с помощью системного вызова ***sendto()*** (следующий раздел). Второй вариант требует указания в качестве номера порта значения 0. В этом случае операционная система сама привязывает сокет к свободному номеру порта. Этот способ обычно используется при настройке сокетов программ клиентов, когда заранее точно знать номер порта программисту необязательно.

Какой номер порта может задействовать пользователь при фиксированной настройке? Номера портов с 1 по 1023 могут назначать сокетам только процессы, работающие с привилегиями системного администратора. Как правило, эти номера закреплены за системными сетевыми службами независимо от вида используемой операционной системы, для того чтобы пользовательские клиентские программы могли запрашивать обслуживание всегда по одним и тем же локальным адресам. Существует также ряд широко применяемых сетевых программ, которые запускают процессы с полномочиями обычных пользователей (например, X-Windows). Для таких программ корпорацией Internet по присвоению имен и номеров (ICANN) выделяется диапазон адресов с 1024 по 49 151, который нежелательно использовать во избежание возможных конфликтов. Номера портов с 49 152 по 65 535 предназначены для процессов обычных пользователей. Во всех наших примерах при фиксированном задании номера порта у сервера мы будем использовать номер 51 000.

IP-адрес при настройке также может быть определен двумя способами. Он может быть привязан к конкретному сетевому интерфейсу (т. е. сетевой плате), заставляя операционную систему принимать/передавать информацию только через этот сетевой интерфейс, а может быть привязан и ко всей вычислительной системе в целом (информация может быть получена/отослана через любой сетевой интерфейс).

В первом случае в качестве значения поля структуры sin_addr.s_addr используется числовое значение IP-адреса конкретного сетевого интерфейса в се-

тевом порядке байт. Во втором случае это значение должно быть равно значению предопределенной константы INADDR_ANY, приведенному к сетевому порядку байт.

Третий параметр системного вызова bind() должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина меняется в зависимости от семейства протоколов и даже различается в пределах одного семейства протоколов. Размер структуры, содержащей адрес сокета, для семейства протоколов TCP/IP может быть определен как sizeof (struct sockaddr_in).

Системный вызов для привязки сокета к конкретному адресу

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockd,
         struct sockaddr *my_addr,
         int addrlen);
```

Описание системного вызова

Системный вызов ***bind*** служит для привязки созданного сокета к определенному полному адресу вычислительной сети.

Параметр ***sockd*** является дескриптором созданного ранее коммуникационного узла, т. е. значением, которое вернул системный вызов ***socket().***

Параметр ***my_addr*** представляет собой адрес структуры, содержащей информацию о том, куда именно мы хотим привязать наш сокет — то, что принято называть адресом сокета. Он имеет тип указателя на структуру-шаблон ***struct sockaddr***, которая должна быть конкретизирована в зависимости от используемого семейства протоколов и заполнена перед вызовом.

Параметр ***addrlen*** должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина в разных семействах протоколов и даже в пределах одного семейства протоколов может быть различной (например, для UNIX Domain).

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и отрицательное значение — в случае ошибки.

Системные вызовы sendto() и recvfrom()

Для отправки датаграмм применяется системный вызов **sendto()**. В число параметров этого вызова входят:

дескриптор сокета, через который отсылается датаграмма;

адрес области памяти, где лежат данные, которые должны составить содержательную часть датаграммы, и их длина;

флаги, определяющие поведение системного вызова (в нашем случае они всегда будут иметь значение 0);

указатель на структуру, содержащую адрес сокета-получателя, и ее фактическая длина.

Системный вызов возвращает отрицательное значение при возникновении ошибки и количество реально отосланных байт при нормальной работе.

Нормальное завершение системного вызова не означает, что датаграмма уже покинула ваш компьютер! Датаграмма сначала помещается в системный сетевой буфер, а ее реальная отправка может произойти после возврата из системного вызова. Вызов **sendto()** может блокироваться, если в сетевом буфере не хватает места для датаграммы.

Для чтения принятых датаграмм и определения адреса получателя (при необходимости) служит системный вызов **recvfrom().** В число параметров этого вызова входят:

Дескриптор сокета, через который принимается датаграмма.

Адрес области памяти, куда следует положить данные, составляющие содержательную часть датаграммы.

Максимальная длина, допустимая для датаграммы. Если количество данных датаграммы превышает заданную максимальную длину, то вызов по умолчанию рассматривает это как ошибочную ситуацию.

Флаги, определяющие поведение системного вызова (в нашем случае они будут полагаться равными 0).

Указатель на структуру, в которую при необходимости может быть занесен адрес сокета-отправителя. Если этот адрес не требуется, то можно указать значение NULL.

Указатель на переменную, содержащую максимально возможную длину адреса отправителя.

После возвращения из системного вызова в нее будет занесена фактическая длина структуры, содержащей адрес отправителя. Если предыдущий параметр имеет значение NULL, то и этот параметр может иметь значение NULL.

Системный вызов recvfrom() по умолчанию блокируется, если отсутствуют принятые датаграммы, до тех пор, пока датаграмма не появится. При возникновении ошибки он возвращает отрицательное значение, при нормальной работе — длину принятой датаграммы.

Системные вызовы sendto и recvfrom

Прототипы системных вызовов

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockd, char *buff, int nbytes, int flags, struct
sockaddr *to, int addrlen);
int recvfrom(int sockd, char *buff, int nbytes, int flags, struct
sockaddr *from, int *addrlen);
```

Описание системных вызовов

Системный вызов sendto предназначен для отправки датаграмм. Системный вызов recvfrom предназначен для чтения пришедших датаграмм и определения адреса отправителя. По умолчанию при отсутствии пришедших датаграмм вызов recvfrom блокируется до тех пор, пока не появится датаграмма. Вызов sendto может блокироваться при отсутствии места под датаграмму в сетевом буфере. Данное описание не является полным описанием системных вызовов, а предназначено только для использования в нашем курсе. За полной информацией обращайтесь к UNIX Manual.

Параметр **sockd** является дескриптором созданного ранее сокета, т. е. значением, возвращенным системным вызовом socket(), через который будет отсылаться или получаться информация.

Параметр **buff** представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр **nbytes** для системного вызова sendto определяет количество байт, которое должно быть передано, начиная с адреса памяти buff. Параметр nbytes для системного вызова recvfrom определяет максимальное количество байт, которое может быть размещено в приемном буфере, начиная с адреса buff.

Параметр **to** для системного вызова sendto определяет ссылку на структуру, содержащую адрес сокета получателя информации, которая должна быть заполнена перед вызовом. Если параметр from для системного вызова recvfrom не равен NULL, то для случая установления связи через пакеты данных он определяет ссылку на структуру, в которую будет занесен адрес сокета-отправителя информации после завершения вызова. В этом случае перед вызовом эту структуру необходимо обнулить.

Параметр **addrlen** для системного вызова sendto должен содержать фактическую длину структуры, адрес которой передается в качестве параметра to. Для системного вызова recvfrom параметр addrlen является ссылкой на переменную, в которую будет занесена фактическая длина структуры адреса сокета-отправителя, если это определено параметром from. Заметим, что перед вызовом этот параметр должен указывать на переменную, содержащую максимально допустимое значение такой длины. Если параметр from имеет значение NULL, то и параметр addrlen может иметь значение NULL.

Параметр **flags** определяет режимы использования системных вызовов. Рассматривать его применение мы в данном курсе не будем и поэтому берем значение этого параметра равным 0.

Возвращаемое значение

В случае успешного завершения системный вызов возвращает количество реально отосланных или принятых байт. При возникновении какой-либо ошибки возвращается отрицательное значение.

Определение IP-адресов для вычислительного комплекса

Для определения IP-адресов на компьютере можно воспользоваться утилитой **/sbin/ifconfig**. Эта утилита выдает всю информацию о сетевых интерфейсах, сконфигурированных в вычислительной системе.

Пример выдачи утилиты показан ниже:

```
eth0      Link encap:Ethernet HWaddr 00:90:27:A7:1B:FE
    inet addr:192.168.253.12 Bcast:192.168.253.255
    Mask:255.255.255.0
    UP BROADCAST NOTRAILERS RUNNING MULTICAST MTU:1500
    Metric:1 RX packets:122556059 errors:0 dropped:0
    overruns:0 frame:0 TX packets:116085111 errors:0
    dropped:0 overruns:0 carrier:0 collisions:0
    txqueuelen:100 RX bytes:2240402748 (2136.6 Mb)
```

```
    TX bytes:3057496950 (2915.8 Mb) Interrupt:10
    Base address:0x1000
lo      Link encap:Local Loopback
    inet addr:127.0.0.1 Mask:255.0.0.0
    UP LOOPBACK RUNNING MTU:16436 Metric:1
    RX packets:403 errors:0 dropped:0 overruns:0 frame:0
    TX packets:403 errors:0 dropped:0 overruns:0
    carrier:0 collisions:0 txqueuelen:0
    RX bytes:39932 (38.9 Kb) TX bytes:39932 (38.9 Kb)
```

Сетевой интерфейс eth0 использует протокол Ethernet. Физический 48-битовый адрес, зашитый в сетевой карте, — 00:90:27:A7:1B:FE. Его IP-адрес — 192.168.253.12.

Сетевой интерфейс lo не относится ни к какой сетевой карте. Это так называемый локальный интерфейс, который через общую память эмулирует работу сетевой карты для взаимодействия процессов, находящихся на одной машине по полным сетевым адресам. Наличие этого интерфейса позволяет отлаживать сетевые программы на машинах, не имеющих сетевых карт. Его IP-адрес обычно одинаков на всех компьютерах — 127.0.0.1.

Рассмотрим простой пример программы 30. Эта программа является UDP-клиентом для стандартного системного сервиса echo. Стандартный сервис принимает от клиента текстовую датаграмму и, не изменяя ее, отправляет обратно. За сервисом зарезервирован номер порта 7. Для правильного запуска программы необходимо указать символьный IP-адрес сетевого интерфейса компьютера, к сервису которого нужно обратиться, в качестве аргумента командной строки, например:

```
a.out 192.168.253.12
```

Программа 30. Простой пример UDP-клиента для сервиса echo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
    int sockfd; /* Дескриптор сокета */
    int n, len; /* Переменные для различных длин и количества
    символов */
    char sendline[1000], recvline[1000]; /* Массивы для отсылаемой
    и принятой строки */
    struct sockaddr_in servaddr, cliaddr; /* Структуры для адресов
    сервера и клиента */
    /* Сначала проверяем наличие второго аргумента в командной
    строке. При его отсутствии прекращаем работу */
    if(argc != 2){
cout<<<«Usage: a.out <IP address>«<<endl;
```

```
        exit(1);
    }
    /* Создание UDP сокета */
    if((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0){
        perror(NULL); /* Печать сообщения об ошибке */
        exit(1);
    }
    /* Заполнение структуры для адреса клиента: семейство
    протоколов TCP/IP, сетевой интерфейс — любой, номер порта по
    усмотрению операционной системы. Поскольку в структуре
    содержится дополнительное ненужное поле, которое должно быть
    нулевым, перед заполнением обнуляем ее всю */
    bzero(&cliaddr, sizeof(cliaddr));
    cliaddr.sin_family = AF_INET;
    cliaddr.sin_port = htons(0);
    cliaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    /* Настройка адреса сокета */
    if(bind(sockfd, (struct sockaddr *) &cliaddr,
    sizeof(cliaddr)) < 0){
        perror(NULL);
        close(sockfd); /* По окончании работы закрываем дескриптор
        сокета */
        exit(1);
    }
    /* Заполнение структуры для адреса сервера: семейство
    протоколов TCP/IP, сетевой интерфейс — из аргумента командной
    строки, номер порта 7. Поскольку в структуре содержится
    дополнительное ненужное поле, которое должно быть нулевым,
    перед заполнением обнуляем ее всю */
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(7);
    if(inet_aton(argv[1], &servaddr.sin_addr) == 0){
        cout<<<«Invalid IP address»<<endl;
        close(sockfd); /* По окончании работы закрываем дескриптор
        сокета */
        exit(1);
    }
    /* Ввод строки, которую отошлем серверу */
   cout<<<«String => «<<endl;
    fgets(sendline, 1000, stdin);
    /* Отсылка датаграммы */
    if(sendto(sockfd, sendline, strlen(sendline)+1, 0, (struct
    sockaddr *) &servaddr, sizeof(servaddr)) < 0){
    perror(NULL);
    close(sockfd);
    exit(1);
    }
    /* Ожидание ответа и чтение его. Максимальная допустимая длина
    датаграммы — 1000 символов, адрес отправителя не нужен */
    if((n = recvfrom(sockfd, recvline, 1000, 0, (struct
    sockaddr *) NULL, NULL)) < 0){
        perror(NULL);
```

```
        close(sockfd);
        exit(1);
    }
    /* Печатаем пришедший ответ и закрываем сокет */
    cout<< recvline << endl;
    close(sockfd);
    return 0;
}
```

Поскольку UDP-сервер использует те же самые системные вызовы, что и UDP-клиент, можно сразу приступить к рассмотрению примера UDP-сервера (программа 31) для сервиса echo.

Программа 31. Простой пример UDP-сервера для сервиса echo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main()
{
    int sockfd; /* Дескриптор сокета */
    int clilen, n; /* Переменные для различных длин и количества
    символов */
    char line[1000]; /* Массив для принятой и отсылаемой строки */
    struct sockaddr_in servaddr, cliaddr;
/* Структуры для адресов сервера и клиента */
    /* Заполнение структуры для адреса сервера: семейство
    протоколов TCP/IP, сетевой интерфейс — любой, номер
    порта 51000. Поскольку в структуре содержится дополнительное
    ненужное поле, которое должно быть нулевым, перед  заполнением
    обнуляем ее всю */
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(51000);
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    /* Создание UDP сокета*/
    if((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0){
        perror(NULL); /* Печать сообщения об ошибке */
        exit(1);
    }
    /* Настройка адреса сокета */
    if(bind(sockfd, (struct sockaddr *) &servaddr,
    sizeof(servaddr)) < 0){
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    while(1) {
```

```
    /* Основной цикл обслуживания*/
   /* В переменную clilen заносим максимальную длину для
   ожидаемого адреса клиента */
        clilen = sizeof(cliaddr);
        /* Ожидание прихода запроса от клиента и чтение его.
        Максимальная допустимая длина датаграммы — 999 символов,
        адрес отправителя помещаем в структуру cliaddr, его
        реальная длина будет занесена в переменную clilen */
        if((n = recvfrom(sockfd, line, 999, 0,  (struct
        sockaddr *) &cliaddr, &clilen)) < 0){
            perror(NULL);
            close(sockfd);
            exit(1);
        }
        /* Печать принятого текста на экране */
        cout<< line<<endl;
        /* Принятый текст отправляем обратно по адресу
        отправителя */
        if(sendto(sockfd, line, strlen(line), 0, (struct
        sockaddr *) &cliaddr, clilen) < 0){
            perror(NULL);
            close(sockfd);
            exit(1);
        } /* Ожидание новой датаграммы*/
    }
    return 0;
}
```

Организация связи между процессами с помощью установки логического соединения

Рассмотрим организацию взаимодействия процессов с помощью протокола TCP, то есть при помощи создания логического соединения. Если взаимодействие процессов через датаграммы напоминает общение людей по переписке, то для протокола TCP лучшей аналогией является общение людей по телефону. Какие действия должен выполнить клиент для того, чтобы связаться по телефону с сервером? Во-первых, необходимо приобрести телефон (создать сокет), во-вторых, подключить его к АТС — получить номер (настроить адрес сокета). Далее требуется позвонить серверу (установить логическое соединение). После установления соединения можно неоднократно обмениваться с сервером информацией (писать и читать из потока данных). По окончании взаимодействия нужно повесить трубку (закрыть сокет).

Первые действия сервера аналогичны действиям клиента. Он должен приобрести телефон и подключить его к АТС (создать сокет и настроить его адрес). А вот дальше поведение клиента и сервера различно. Представьте себе, что телефоны изначально продаются с выключенным звонком. Звонить по ним можно, а вот принять звонок — нет. Для того чтобы вы могли пообщаться, необходимо включить звонок. В терминах сокетов это означает, что TCP-сокет по умолчанию создается в активном состоянии и предназначен не для приема, а

для установления соединения. Для того чтобы соединение принять, сокет требуется перевести в пассивное состояние.

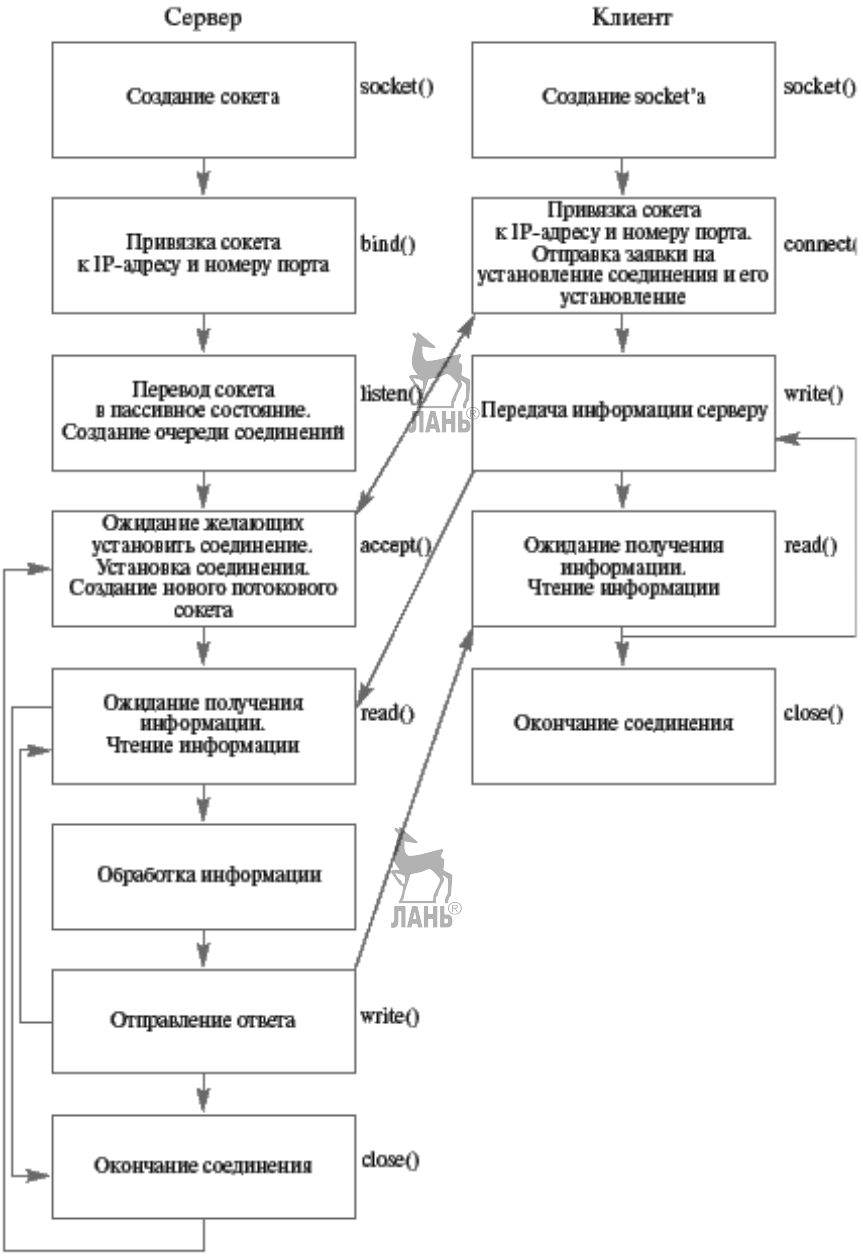


Рис. 33
Схема взаимодействия клиента и сервера для протокола TCP

Если два человека беседуют по телефону, то попытка других людей дозвониться до них окажется неудачной. Будет идти сигнал «занято», и соединение не установится. В то же время хотелось бы, чтобы клиент в такой ситуации не

получал отказ в обслуживании, а ожидал своей очереди. Подобное наблюдается в различных телефонных справочных, когда вы слышите «Ждите, пожалуйста, ответа. Вам обязательно ответит оператор». Поэтому следующее действие сервера — это создание очереди для обслуживания клиентов. Далее сервер должен дождаться установления соединения, прочитать информацию, переданную по линии связи, обработать ее и отправить полученный результат обратно. Обмен информацией может осуществляться неоднократно. Заметим, что сокет, находящийся в пассивном состоянии, не предназначен для операций приема и передачи информации. Для общения на сервере во время установления соединения автоматически создается новый потоковый сокет, через который и производится обмен данными с клиентами. По окончании общения сервер «кладет трубку» (закрывает этот новый сокет) и отправляется ждать очередного звонка.

Схематично эти действия выглядят так, как показано на рисунке 33. Как и в случае протокола UDP, отдельным действиям или их группам соответствуют системные вызовы, частично совпадающие с вызовами для протокола UDP. Их названия написаны справа от блоков соответствующих действий.

Для протокола TCP неравноправность процессов клиента и сервера видна особенно отчетливо в различии используемых системных вызовов. Для создания сокетов и там, и там по-прежнему используется системный вызов socket(). Затем наборы системных вызовов становятся различными.

Для привязки сервера к IP-адресу и номеру порта, как и в случае UDP-протокола, используется системный вызов bind(). Для процесса клиента эта привязка объединена с процессом установления соединения с сервером в новом системном вызове connect() и скрыта от глаз пользователя. Внутри этого вызова операционная система осуществляет настройку сокета на выбранный ею порт и на адрес любого сетевого интерфейса. Для перевода сокета на сервере в пассивное состояние и для создания очереди соединений служит системный вызов listen(). Сервер ожидает соединения и получает информацию об адресе соединившегося с ним клиента с помощью системного вызова accept(). Поскольку установленное логическое соединение выглядит со стороны процессов как канал связи, позволяющий обмениваться данными с помощью потоковой модели, для передачи и чтения информации оба системных вызова используют уже известные нам системные вызовы read() и write(), а для завершения соединения — системный вызов close(). При работе с сокетами вызовы read() и write() обладают теми же особенностями поведения, что и при работе с pip’ами и FIFO.

Установление логического соединения

Системный вызов connect()

Среди системных вызовов со стороны клиента появляется только один новый — connect(). Системный вызов connect() при работе с TCP-сокетами служит для установления логического соединения со стороны клиента. Вызов connect() скрывает внутри себя настройку сокета на выбранный системой порт и произвольный сетевой интерфейс (вызов bind() с нулевым номером порта и IP-адресом INADDR_ANY). Вызов блокируется до тех пор, пока не будет уста-

новлено логическое соединение или пока не пройдет определенный промежуток времени, который может регулироваться системным администратором.

Для установления соединения необходимо задать три параметра: дескриптор активного сокета, через который будет устанавливаться соединение, полный адрес сокета-сервера и его длину.

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockd, struct sockaddr *servaddr, int addrlen);
```

Описание системного вызова

Системный вызов **connect** служит для организации связи клиента с сервером. Чаще всего он используется для установления логического соединения, хотя может быть применен и при связи с помощью датаграмм (**connectionless**). Данное описание не является полным описанием системного вызова, а предназначено только для использования в нашем курсе. Полную информацию можно найти в UNIX Manual.

Параметр **sockd** является дескриптором созданного ранее коммуникационного узла, т. е. значением, которое вернул системный вызов **socket()**.

Параметр **servaddr** представляет собой адрес структуры, содержащей информацию о полном адресе сокета-сервера. Он имеет тип указателя на структуру-шаблон **struct sockaddr**, которая должна быть конкретизирована в зависимости от используемого семейства протоколов и заполнена перед вызовом.

Параметр **addrlen** должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина меняется в зависимости от семейства протоколов и различается даже в пределах одного семейства протоколов (например, для UNIX Domain).

При установлении виртуального соединения системный вызов не возвращается до его установления или до истечения установленного в системе времени — **timeout**. При использовании его в **connectionless** связи вызов возвращается немедленно.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и отрицательное значение, если в процессе его выполнения возникла ошибка.

Рассмотрим пример — программу 32. Это простой TCP-клиент, обращающийся к стандартному системному сервису echo. Стандартный сервис принимает от клиента текстовую датаграмму и, не изменяя ее, отправляет обратно. За сервисом зарезервирован номер порта 7. Заметим, что это порт 7 TCP — не путать с портом 7 UDP из примера в разделе «Пример программы UDP-клиента»! Для правильного запуска программы необходимо указать символьный IP-адрес сетевого интерфейса компьютера, к сервису которого требуется обратиться, в качестве аргумента командной строки, например:

```
$a.out 192.168.253.12
```

Для того чтобы подчеркнуть, что после установления логического соединения клиент и сервер могут обмениваться информацией неоднократно, клиент

трижды запрашивает текст с экрана, отсылает его серверу и печатает полученный ответ.

Программа 32. Простой пример TCP-клиента для сервиса echo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
using namespace std;
void main(int argc, char **argv)
{
    int sockfd; /* Дескриптор сокета */
    int n; /* Количество переданных или прочитанных символов */
    int i; /* Счетчик цикла */
    char sendline[1000],recvline[1000]; /* Массивы для отсылаемой
    и принятой строки */
    struct sockaddr_in servaddr; /* Структура для адреса
    сервера */
    /* Сначала проверка наличия второго аргумента в командной
    строке. При его отсутствии — прекращение работы */
    if(argc != 2){
       cout<<<«Usage: a.out <IP address>«<<endl;
        exit(1);
    }
    /* Обнуление символьных массивов */
    bzero(sendline,1000);
    bzero(recvline,1000);
    /* Создание TCP сокета */
    if((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0){
        perror(NULL); /* Печать сообщения об ошибке */
        exit(1);
    }
    /* Заполнение структуры для адреса сервера: семейство
    протоколов TCP/IP, сетевой интерфейс — из аргумента командной
    строки, номер порта 7. Поскольку в структуре содержится
    дополнительное ненужное  поле, которое должно быть нулевым,
    перед заполнением обнуляем ее всю */
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(51000);
    if(inet_aton(argv[1], &servaddr.sin_addr) == 0){
       cout<<<«Invalid IP address»<<endl;
        close(sockfd);
        exit(1);
    }
    /* Установка логического соединения через созданный сокет
    с сокетом сервера, адрес которого занесли в структуру */
    if(connect(sockfd, (struct sockaddr *) &servaddr,
    sizeof(servaddr)) < 0){
```

```
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    /* Три раза в цикле ввод строки с клавиатуры, отправка
    ее серверу и чтение полученного ответа */
    for(i=0; i<3; i++){
        cout<<<«String => «<<endl;
        fflush(stdin);
        fgets(sendline, 1000, stdin);
        if( (n = write(sockfd, sendline,
        strlen(sendline)+1)) < 0){
            perror(«Can\'t write\n»);
            close(sockfd);
            exit(1);
        }
        if ( (n = read(sockfd,recvline, 999)) < 0){
            perror(«Can\'t read\n»);
            close(sockfd);
            exit(1);
        }
        cout<<recvline<<endl;
    }
    /* Завершение соединения*/
    close(sockfd);
}
```

Как происходит установление виртуального соединения

Протокол TCP является надежным дуплексным протоколом. С точки зрения пользователя работа через протокол TCP выглядит как обмен информацией через поток данных. Внутри сетевых частей операционных систем поток данных отправителя нарезается на пакеты данных, которые, собственно, путешествуют по сети и на машине-получателе вновь собираются в выходной поток данных. В протоколе TCP используются приемы нумерации передаваемых пакетов и контроля порядка их получения, подтверждения о приеме пакета со стороны получателя и подсчет контрольных сумм по передаваемой информации. Для правильного порядка получения пакетов получатель должен знать начальный номер первого пакета отправителя. Поскольку связь является дуплексной и в роли отправителя пакетов данных могут выступать обе взаимодействующие стороны, они до передачи пакетов данных должны обменяться, по крайней мере, информацией об их начальных номерах. Согласование начальных номеров происходит по инициативе клиента при выполнении системного вызова connect(). Для такого согласования клиент посылает серверу специальный пакет информации, который принято называть SYN (от слова synchronize — синхронизировать). Он содержит как минимум начальный номер для пакетов данных, который будет использовать клиент. Сервер должен подтвердить получение пакета SYN от клиента и отправить ему свой пакет SYN с начальным номером для пакетов данных в виде единого пакета с сегментами

SYN и ACK (от слова acknowledgement — подтверждение). В ответ клиент пакетом данных ACK должен подтвердить прием пакета данных от сервера.

Описанная выше процедура, получившая название трехэтапного рукопожатия (three-way handshake), схематично изображена на рисунке 34. При приеме на машине-сервере пакета SYN, направленного на пассивный (слушающий) сокет, сетевая часть операционной системы создает копию этого сокета — присоединенный сокет — для последующего общения, отмечая его как сокет с не полностью установленным соединением. После приема от клиента пакета ACK этот сокет переводится в состояние полностью установленного соединения, и тогда он готов к дальнейшей работе с использованием вызовов read() и write().

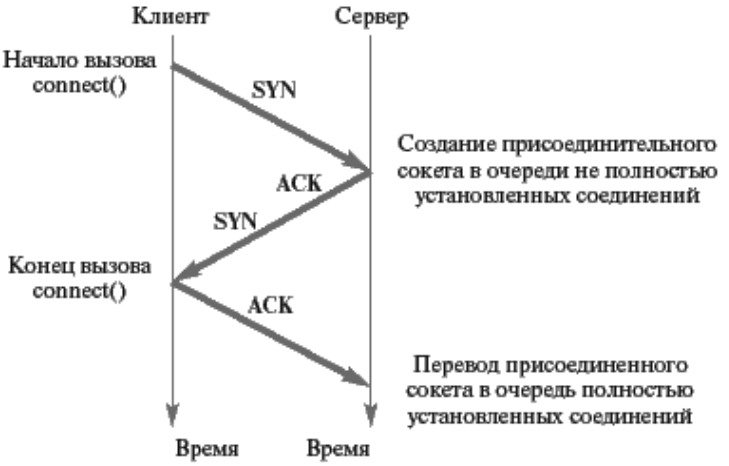


Рис. 34
Схема установления TCP-соединения

Системный вызов listen()

Системный вызов **listen()** является первым из еще неизвестных нам вызовов, применяемым на TCP-сервере. В его задачу входит перевод TCP-сокета в пассивное (слушающее) состояние и создание очередей для порождаемых при установлении соединения присоединенных сокетов, находящихся в состоянии не полностью установленного соединения и полностью установленного соединения. Для этого вызов имеет два параметра: дескриптор TCP-сокета и число, определяющее глубину создаваемых очередей.

Системный вызов listen()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockd, int backlog);
```

Описание системного вызова

Системный вызов listen используется сервером, ориентированным на установление связи путем виртуального соединения, для перевода сокета в пассивный режим и установления глубины очереди для соединений.

Параметр sockd является дескриптором созданного ранее сокета, который должен быть переведен в пассивный режим, т. е. значением, которое вернул системный вызов socket(). Системный вызов listen требует предварительной настройки адреса сокета с помощью системного вызова bind().

Параметр backlog определяет максимальный размер очередей для сокетов, находящихся в состояниях полностью и не полностью установленных соединений.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Последний параметр на разных UNIX-подобных операционных системах и даже на разных версиях одной и той же системы может иметь различный смысл. Где-то это суммарная длина обеих очередей, где-то он относится к очереди не полностью установленных соединений (например, Linux до версии ядра 2.2), где-то — к очереди полностью установленных соединений (например, Linux, начиная с версии ядра 2.2), где-то — вообще игнорируется.

Системный вызов accept()

Системный вызов **accept()** позволяет серверу получить информацию о полностью установленных соединениях. Если очередь полностью установленных соединений не пуста, то он возвращает дескриптор для первого присоединенного сокета в этой очереди, одновременно удаляя его из очереди. Если очередь пуста, то вызов ожидает появления полностью установленного соединения. Системный вызов также позволяет серверу узнать полный адрес клиента, установившего соединение. У вызова есть три параметра: дескриптор слушающего сокета, через который ожидается установление соединения; указатель на структуру, в которую при необходимости будет занесен полный адрес сокета-клиента, установившего соединение; указатель на целую переменную, содержащую максимально допустимую длину этого адреса. Как и в случае вызова recvfrom(), последний параметр является модернизируемым, а если нас не интересует, кто с нами соединился, то вместо второго и третьего параметров можно указать значение NULL.

Системный вызов accept()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockd, struct sockaddr *cliaddr, int *clilen);
```

Описание системного вызова

Системный вызов **accept** используется сервером, ориентированным на установление связи путем виртуального соединения, для приема полностью установленного соединения.

Параметр **sockd** является дескриптором созданного и настроенного сокета, предварительного переведенного в пассивный (слушающий) режим с помощью системного вызова **listen().**

Системный вызов **accept** требует предварительной настройки адреса сокета с помощью системного вызова **bind().**

Параметр **cliaddr** служит для получения адреса клиента, установившего логическое соединение, и должен содержать указатель на структуру, в которую будет занесен этот адрес.

Параметр **clilen** содержит указатель на целую переменную, которая после возвращения из вызова будет содержать фактическую длину адреса клиента. Заметим, что перед вызовом эта переменная должна содержать максимально допустимое значение такой длины. Если параметр **cliaddr** имеет значение NULL, то и параметр **clilen** может иметь значение **NULL**.

Возвращаемое значение

Системный вызов возвращает при нормальном завершении дескриптор присоединенного сокета, созданного при установлении соединения для последующего общения клиента и сервера, и значение -1 при возникновении ошибки.

Программа 33, реализующая простой TCP-сервер для сервиса echo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
void main()
{
    int sockfd, newsockfd; /* Дескрипторы для слушающего
    и присоединенного сокетов */
    int clilen; /* Длина адреса клиента */
    int n; /* Количество принятых символов */
    char line[1000]; /* Буфер для приема информации */
    struct sockaddr_in servaddr, cliaddr; /* Структуры для
    размещения полных адресов сервера и клиента */
    /* Создаем TCP-сокет */
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        perror(NULL);
        exit(1);
    }
    /* Заполнение структуры для адреса сервера: семейство
    протоколов TCP/IP, сетевой интерфейс — любой, номер
    порта 51000. Поскольку в структуре содержится дополнительное
    ненужное  поле, которое должно быть нулевым, обнуляем ее всю
    перед заполнением */
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family= AF_INET;
    servaddr.sin_port= htons(51000);
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    /* Настраиваем адрес сокета */
    if(bind(sockfd, (struct sockaddr *) &servaddr,
sizeof(servaddr)) < 0){
        perror(NULL);
```

```
    close(sockfd);
    exit(1);
}
/* Переводим созданный сокет в пассивное (слушающее)
состояние. Глубину очереди для установленных соединений
описываем значением 5 */
if(listen(sockfd, 5) < 0){
    perror(NULL);
    close(sockfd);
    exit(1);
}
/* Основной цикл сервера */
while(1){
    /* В переменную clilen заносим максимальную длину
    ожидаемого адреса клиента */
    clilen = sizeof(cliaddr);
    /* Ожидание полностью установленного соединения на
    слушающем сокете. При нормальном завершении в структуре
    cliaddr будет лежать полный адрес клиента, установившего
    соединение, а в переменной clilen — его фактическая длина.
    Вызов вернет дескриптор присоединенного сокета, через
    который будет происходить общение с клиентом. Информация
    о клиенте у нас в дальнейшем никак не используется,
    поэтому вместо второго и третьего параметров можно было
    поставить значения NULL. */
    if((newsockfd = accept(sockfd, (struct sockaddr *)
    &cliaddr, &clilen)) < 0){
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    /* В цикле принимаем информацию от клиента до тех пор,
    пока не произойдет ошибка (вызов read() вернет
    отрицательное значение) или клиент не закроет соединение
    (вызов read() вернет значение 0). Максимальную длину одной
    порции данных от клиента ограничим 999 символами.
    В операциях чтения и записи пользуемся дескриптором
    присоединенного сокета, т. е. значением, которое вернул
    вызов accept().*/
    while((n = read(newsockfd, line, 999)) > 0){
        /* Принятые данные отправить обратно */
        if((n = write(newsockfd, line, strlen(line)+1)) < 0){
            perror(NULL);
            close(sockfd);
            close(newsockfd);
            exit(1);
        }
    }
/* Если при чтении возникла ошибка — завершение работы */
if(n < 0){
        perror(NULL);
        close(sockfd);
        close(newsockfd);
```

```
            exit(1);
        }
 /* Закрытие дескриптора присоединенного сокета и ожидание нового
соединения */
        close(newsockfd);
    }
}
```

Применение интерфейса сетевых вызовов для других семейств протоколов. UNIX Domain-протоколы

Файлы типа «сокет»

Рассмотренный выше интерфейс умеет работать не только со стеком протоколов TCP/IP, но и с другими семействами протоколов. При этом требуется лишь незначительное изменение написанных с его помощью программ. Рассмотрим действия, которые необходимо выполнить для модернизации написанных для TCP/IP программ под другое семейство протоколов.

Изменяется тип сокета, поэтому для его точной спецификации нужно задавать другие параметры в системном вызове **socket().**

В различных семействах протоколов применяются различные адресные пространства для удаленных и локальных адресов сокетов. Поэтому меняется состав структуры для хранения полного адреса сокета, название ее типа, наименования полей и способ их заполнения.

Описание типов данных и предопределенных констант будет находиться в других include-файлах, поэтому потребуется заменить include-файлы <netinet/in.h> и <arpa/inet.h> на файлы, относящиеся к выбранному семейству протоколов.

Может измениться способ вычисления фактической длины полного адреса сокета и указания его максимального размера.

Семейство UNIX Domain-протоколов предназначено для общения локальных процессов с использованием интерфейса системных вызовов. Оно содержит один потоковый и один датаграммный протокол. Никакой сетевой интерфейс при этом не используется, а вся передача информации реально происходит через адресное пространство ядра операционной системы. Многие программы, взаимодействующие и с локальными, и с удаленными процессами (например, X-Windows), для локального общения используют этот стек протоколов.

Поскольку общение происходит в рамках одной вычислительной системы, в полном адресе сокета его удаленная часть отсутствует. В качестве адресного пространства портов — локальной части адреса — выбрано адресное пространство, совпадающее с множеством всех допустимых имен файлов в файловой системе. При этом в качестве имени сокета требуется задавать имя несуществующего еще файла в директории, к которой у вас есть права доступа как на запись, так и на чтение. При настройке адреса (системный вызов bind()) под этим именем будет создан файл типа «сокет» — последний еще неизвестный

нам тип файла. Этот файл для сокетов играет роль файла-метки типа FIFO для именованных pip'ов.

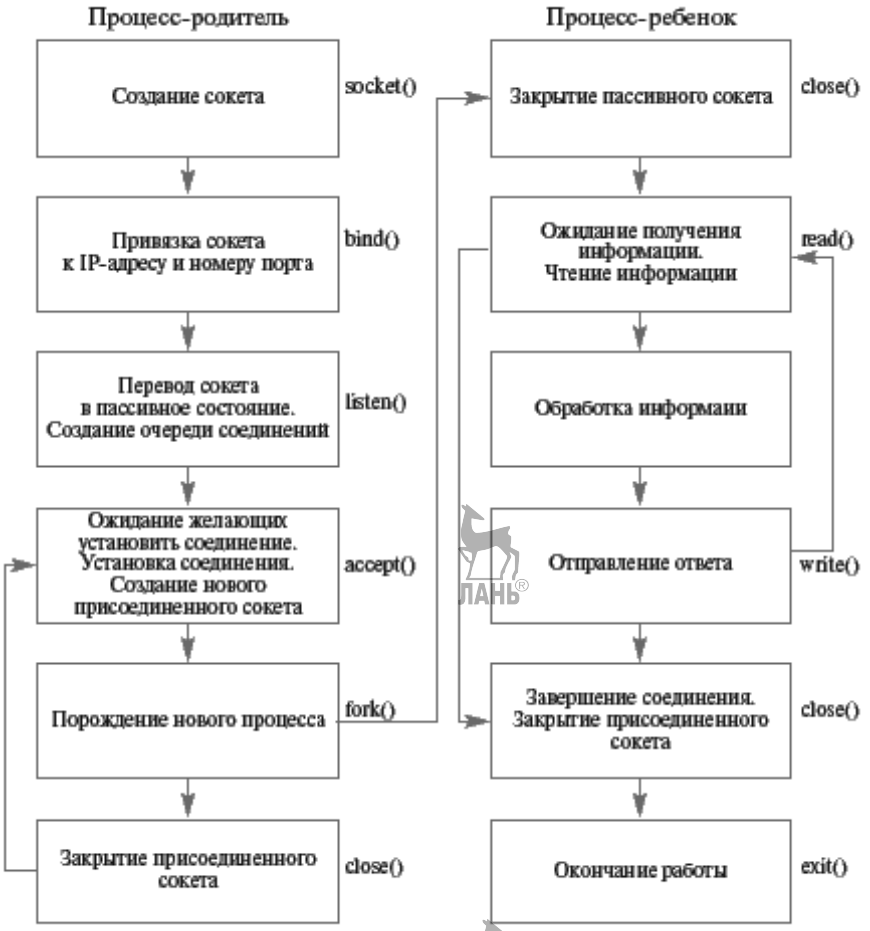


Рис. 35
Схема работы TCP-сервера с параллельной обработкой запросов

Если на вашей машине функционирует X-Windows, то вы сможете обнаружить такой файл в директории с именем /tmp/.X11-UNIX — это файл типа «сокет», служащий для взаимодействия локальных процессов с оконным сервером.

Для хранения полного адреса сокета используется структура следующего вида, описанного в файле **<sys/un.h>:**

```
struct sockaddr_un{
    short sun_family;
    /* Избранное семейство протоколов – всегда AF_UNIX */

    char sun_path[108];
    /* Имя файла типа «сокет» */
};
```

Выбранное имя будет копироваться внутрь структуры, используя функцию **strcpy().**

Фактическая длина полного адреса сокета, хранящегося в структуре с именем **my_addr,** может быть вычислена следующим образом:
`sizeof(short)+strlen(my_addr.sun_path).`

В Linux для этих целей можно использовать специальный макрос языка Си.
`SUN_LEN(struct sockaddr_un*)`

Ниже приведены тексты переписанных под семейство UNIX Domain-протоколов клиента и сервера для сервиса echo (программы 34 и 35), общающиеся через датаграммы. Клиент использует сокет с именем AAAA в текущей директории, а сервер — сокет с именем BBBB. Как следует из описания типа данных, эти имена (полные или относительные) не должны по длине превышать 107 символов. Комментарии даны лишь для изменений по сравнению с программами 31 и 32.

Программа 34. UNIX Domain протокол сервера для сервиса echo, общающегося через датаграммы

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h> /* Новый include-файл вместо netinet/in.h и
arpa/inet.h */
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
int main()
{
    int sockfd;
    int clilen, n;
    char line[1000];
    struct sockaddr_un servaddr, cliaddr;
/* новый  тип данных под адреса сокетов */
    if((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
/* Изменен тип семейства протоколов */
    {
        perror(NULL);
        exit(1);
    }
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_UNIX; /* Изменен тип семейства
    протоколов и имя поля в структуре */
    strcpy(servaddr.sun_path,»BBBB»); /* Локальный адрес сокета
    сервера – BBBB – в текущей директории */
    if(bind(sockfd, (struct sockaddr *) &servaddr,
    SUN_LEN(&servaddr)) < 0)
/* Изменено вычисление фактической длины адреса */
    {
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    while(1) {
```

```
        clilen = sizeof(struct sockaddr_un); /* Изменено
        вычисление максимальной длины для адреса клиента */
        if((n = recvfrom(sockfd, line, 999, 0, (struct sockaddr *)
        &cliaddr, &clilen)) < 0)
{
            perror(NULL);
            close(sockfd);
            exit(1);
        }
        if(sendto(sockfd, line, strlen(line), 0, (struct
        sockaddr *) &cliaddr, clilen) < 0)
{
            perror(NULL);
            close(sockfd);
            exit(1);
        }
    }
    return 0;
}
```

Программа 35. UNIX Domain-протокол клиента для сервиса echo, общающегося через датаграммы

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h> /* Новый include-файл вместо netinet/in.h и
arpa/inet.h */
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
user namespace std;
int main() /* Аргументы командной строки не нужны, так как сервис
является локальным, и не нужно указывать, к какой машине мы обра-
щаемся с запросом */
{
    int sockfd;
    int n, len;
    char sendline[1000], recvline[1000];
    struct sockaddr_un servaddr, cliaddr; /* новый тип данных под
    адреса сокетов */
    if((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
    /* Изменен тип семейства протоколов */
    {
        perror(NULL);
        exit(1);
    }
    bzero(&cliaddr, sizeof(cliaddr));
    cliaddr.sun_family= AF_UNIX; /* Изменен тип семейства
    протоколов и имя поля в структуре */
    strcpy(cliaddr.sun_path,»AAAA»); /* Локальный адрес сокета
    клиента — AAAA — в текущей директории */
    if(bind(sockfd, (struct sockaddr *) &cliaddr,
    SUN_LEN(&cliaddr)) < 0) /* Изменено вычисление фактической
```

```
    длины адреса */
    {
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_UNIX; /* Изменен тип семейства
    протоколов и имя поля в структуре */
    strcpy(servaddr.sun_path,»BBBB»); /* Локальный адрес сокета
    сервера — BBBB — в текущей директории */
cout<<<«String => «<<endl;
    fgets(sendline, 1000, stdin);
    if(sendto(sockfd, sendline, strlen(sendline)+1, 0, (struct
    sockaddr *) &servaddr, SUN_LEN(&servaddr)) < 0) /* Изменено
    вычисление фактической длины адреса */
    {
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    if((n = recvfrom(sockfd, recvline, 1000, 0,
    (struct sockaddr *) NULL, NULL)) < 0){
        perror(NULL);
        close(sockfd);
        exit(1);
    }
    recvline[n] = 0;
   cout<< recvline<< endl;
    close(sockfd);
    return 0;
}
```

Методы обнаружения ошибок в сетях и реконфигурации сетей

Цели и задачи: Изучить понятие «сеть» и его основы. Рассмотреть основные сетевые команды Linux.

1.1. Введение

В современном мире каждый день люди тратят по несколько часов, сидя в Интернете. И зачастую у многих возникают те или иные ошибки, связанные с сетью, из-за которых пропадает доступ в Интернет. К сожалению, на сегодняшний день большинство людей до сих пор не знают, как именно работают сети, и тем более как решить проблемы в сетях.

1.2. Стек протоколов TCP/IP

Что же такое сеть? **Сеть —** это два и более компьютера, соединенных между собой сетевым оборудованием и обменивающиеся друг с другом информацией по определенным правилам. Данные правила прописаны в стеке протоколов **TCP/IP**.

Transmission Control Protocol/Internet Protocol (Стек протоколов TCP/IP) — говоря простыми словами, это набор взаимодействующих протоколов разных уровней, согласно которым происходит обмен данными в сети.

Каждый проток описывает только те правила, которые в нем указаны. Все протоколы разделены по уровням функциональности. Для разделения протоколов по уровням была разработана модель сетевого взаимодействия **OSI** (Open Systems Interconnection Basic Reference Model*)*. Модель **OSI** состоит из семи различных уровней. Каждый из уровней выполняет свою определенную задачу в соответствии с набором правил, называемым протоколом.

Уровень модели OSI	Функции	Протоколы
7. Прикладной уровень	Взаимодействие с конечным пользователем	FRP, HTTP, DHCP, SNMP, SSH, telnet, SMTP, IMAP
6. Представительский уровень	Представление, кодирование и шифрование данных	ASCII, EBCDIC
5. Сеансовый уровень	Управление сеансом связи	RPC, PAP
4. Транспортный уровень	Надежная связь между двумя конечными устройствами поверх безнадежной сети	TCP, UPD
3. Сетевой уровень	Логическая адресация и определение маршрута	IP, NAT, OSPF, EIGRP, RIP, IPSec
2. Канальный уровень	Канальная адресация и обнаружение ошибок физического уровня	PPP, PPPoE, Ethernet, 802.1, ARP, HDLC, Frame Relay
1. Физический уровень	Определяет характеристики сигналов и среды передачи данных	Ethernet, xDSL, 802.11(Wi-Fi), USB, Bluetooth

1.3. Адресация

Каждому хосту, построенному на стеке протоколов TCP/IP, присваивается **IP-адрес**. Сам **IP-адрес** представляет собой 32-битовое двоичное число. **IP-адрес** (IPv4) записывает в виде 4 десятичных чисел (от 0 до 255), разделенных между собой точками. Пример: 192.168.0.1. **IP-адрес** делится на **адрес подсети** и **адрес хоста**. 192.168.0 — это адрес подсети, а 1 — это адрес хоста.

Чтобы узнать, какая часть адреса относится к **адресу подсети**, а какая к **адресу хоста,** используют **маску подсети. Маска подсети** — также 4 десятичных числа, которые складываются в двоичной форме при помощи логического «И» с IP-адресом. В результате чего выясняется, к какой подсети принадлежит адрес. Пример: адрес 192.168.0.1 с маской 255.255.255.0 принадлежит подсети 192.168.0.

1.4. Маршрутизация

Маршрутизация — это процесс, в котором определяется путь следования информации в сетях связи. **Маршрутизация** выполняет очень важную роль в сетях, поскольку она служит для приема пакета от одного устройства и передаче его другому устройству через другие сети. Маршрутизатором (шлюзом) принято называть узел сети, который имеет несколько интерфейсов. Каждая сеть имеет свой MAC-адрес и IP-адрес.

Важным понятием является **таблица маршрутизации**, хранящаяся на маршрутизаторе, которая описывает соответствие между адресами назначения и интерфейсами, через которые следует отправить пакет данных до следующего узла. В **таблице маршрутизации** содержится: адрес узла назначения, маска сети назначения, адрес шлюза (обозначающий адрес маршрутизатора в сети, на

который необходимо отправить пакет, следующий до указанного адреса назначения), интерфейс (физический порт, через который передается пакет) и метрика (числовой показатель, задающий приоритет маршрута).

Сетевой адрес	Маска	Адрес шлюза	Интерфейс	Метрика
127.0.0.0	255.0.0.0	127.0.0.1	127.0.0.1	1
0.0.0.0	0.0.0.0	198.21.17.7	198.21.17.5	1
56.0.0.0	255.0.0.0	213.34.12.4	213.34.12.3	15

Запись в таблицу маршрутизации может быть выполнена тремя способами.

Первый способ — применение прямого соединения, при котором маршрутизатор сам определяет подключенную подсеть. Прямое соединение является наиболее достоверным способом маршрутизации.

Второй способ — занесение маршрутов вручную. В данном случае имеет место статическая маршрутизация. Статический маршрут определяет IP-адрес следующего соседнего маршрутизатора или локальный выходной интерфейс, который используется для направления трафика к определенной подсети — получателю. Статические маршруты должны быть заданы на обеих концах канала связи между маршрутизаторами, иначе удаленный маршрутизатор не будет знать маршрута, по которому нужно отправлять ответные пакеты, и будет организована лишь односторонняя связь.

Третий способ — автоматическое размещение записей с помощью протоколов маршрутизации. Данный способ называется динамической маршрутизацией. Протоколы динамической маршрутизации могут автоматически отслеживать изменения в топологии сети.

1.5. Физическая сеть

Если у вас возникли проблемы с доступом в Интернет, то первым делом нужно проверить, корректно ли работают коммутаторы и кабели.

Для этого попробуйте полностью отключить коммутатор, выдернуть из него все кабели. Подождать 10 секунд и подключить все обратно.

Если же это не помогло, то далее посмотрите в документации к вашему коммутатору, какие именно индикаторы (лампочки) должны загораться на его панели, и какой индикатор за что отвечает.

Если все нужные индикаторы загорелись, но у вас до сих пор нет доступа к Интернету, то нужно приступить к устранению неполадок вашей сети.

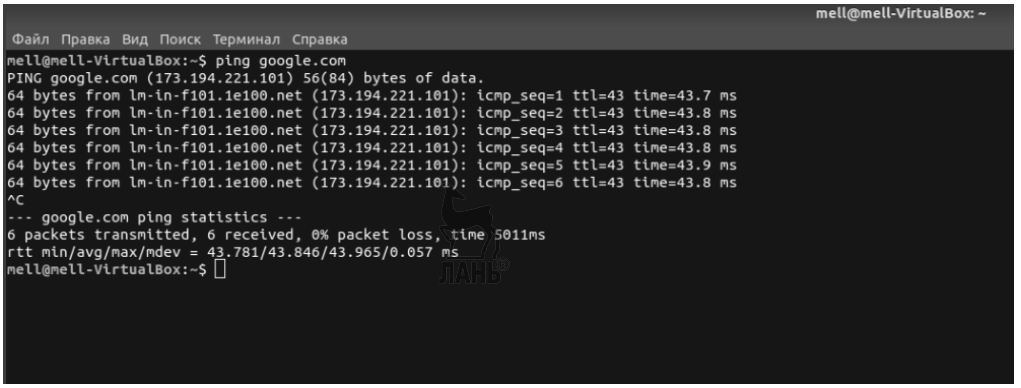
1.6. Сетевые команды Linux

Проверка сетевого подключения с помощью команды ping

Одна из наиболее часто используемых команд Linux для устранения неполадок в сети является команда **ping**. С помощью данной команды выполняется проверка, может ли быть достигнут определенный IP-адрес.

Команда **ping** работает, отправляя эхо-запрос ICMP для проверки подключения к сети.

```
$ ping google.com
```



В результате работы данной команды нам высвечивается оповещение о том, что было передано и приятно 6 пакетов, 0% потерь, время 5011 ms. Также можно посмотреть время приема-передачи каждого пакета.

Эта команда измеряет средний ответ. Если ответа нет, возможно, есть одно из следующих:

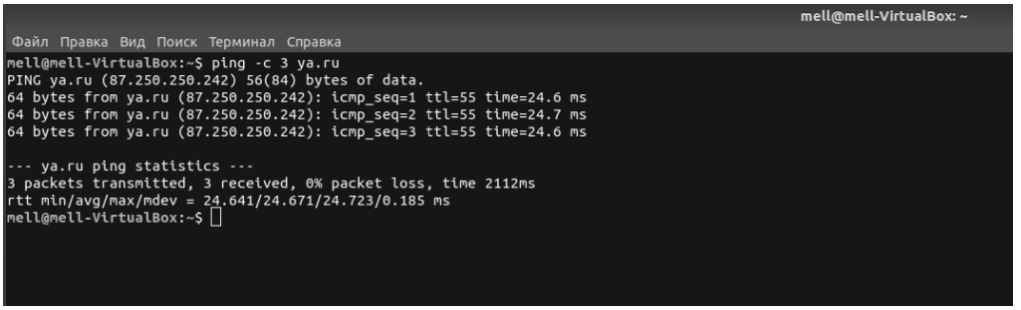
– в самой сети есть физическая проблема;

– расположение может быть неправильным или нефункциональным;

– запрос ping заблокирован целью;

– существует проблема с таблицей маршрутизации.

Если вы хотите ограничить количество эхо-запросов, то для этого после команды **ping** нужно дописать ***-c 3 ya.ru***, тем самым будет отправлено лишь 3 эхо-запроса на ya.ru.



Важные сведения, которые нужно знать о команде *ping*.

Расстояние до цели: чем дальше от вас находится сервер, который вы пингуете, тем больше времени уйдет на прием-передачу пакетов.

Скорость соединения: если у вас медленное соединение, то на прием-передачу пакетов понадобится больше времени.

Количество переходов: количество переходов относится к маршрутизаторам и серверам, через которые проходит эхо-сигнал до достижения пункта назначения.

Получение записи DNS с помощью команды dig

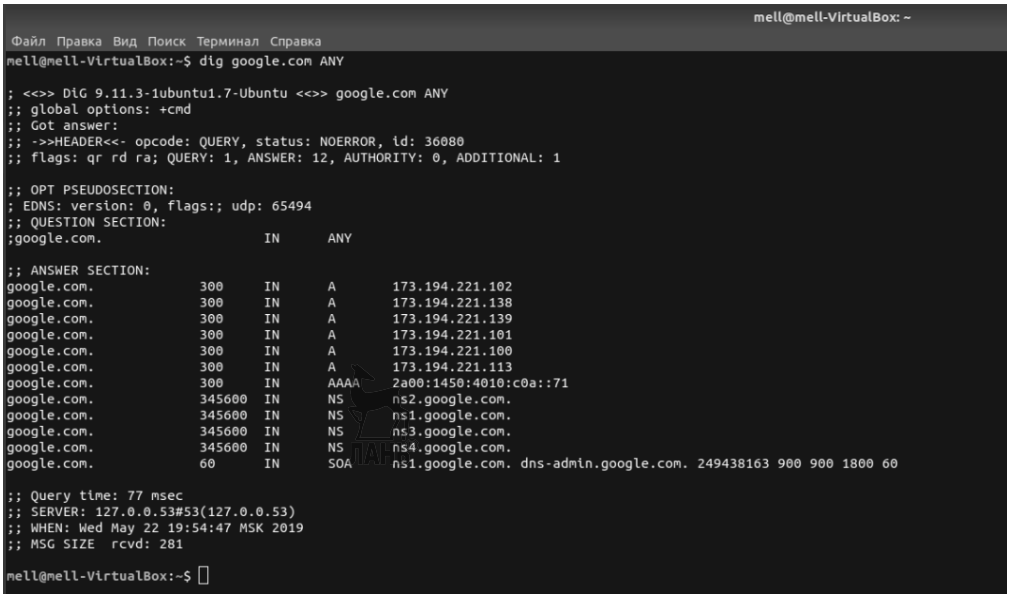
Пару слов о DNS. DNS хранит себе информацию о том, какому IP-адресу соответствует то или иное доменное имя. Например, пользователь вводит в браузере в адресную строку google.com, этот запрос отправляется на DNS-сервер, в котором выполняется проверка, существует ли такое доменное имя. Если да, то DNS-сервер отправляет на наш компьютер IP-адрес сайта google.com. Если такое доменное имя не существует, то компьютер нас оповещает о том, что не удалось найти IP-адрес введенного сайта.

Команда **dig** используется для проверки сопоставлений DNS, адресов хостов, записей MX и всех других записей DNS для лучшего понимания топографии DNS.

```
$ dig google.com
```

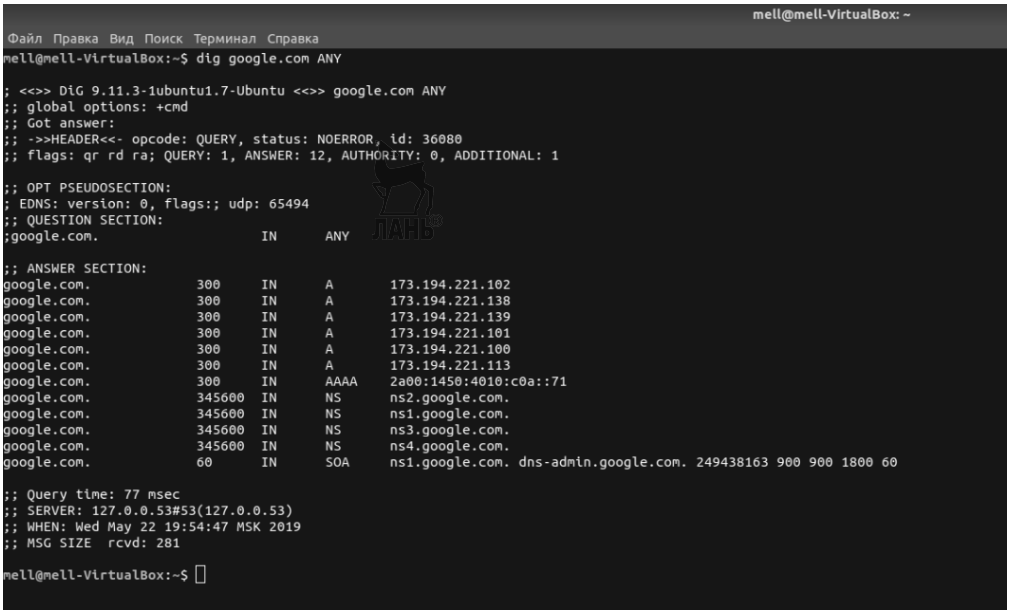
```
mell@mell-VirtualBox: ~
Файл  Правка  Вид  Поиск  Терминал  Справка
mell@mell-VirtualBox:~$ dig google.com

; <<>> DiG 9.11.3-1ubuntu1.7-Ubuntu <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 39689
;; flags: qr rd ra; QUERY: 1, ANSWER: 6, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             288     IN      A       173.194.221.139
google.com.             288     IN      A       173.194.221.113
google.com.             288     IN      A       173.194.221.102
google.com.             288     IN      A       173.194.221.100
google.com.             288     IN      A       173.194.221.101
google.com.             288     IN      A       173.194.221.138

;; Query time: 5 msec
;; SERVER: 127.0.0.53#53(127.0.0.53)
;; WHEN: Wed May 22 19:53:47 MSK 2019
;; MSG SIZE  rcvd: 135

mell@mell-VirtualBox:~$
```

В результате работы данной команды можно увидеть IP-адреса сайта google.com. Если ввести IP-адреса, которые были получены в результате работы команды **dig** в адресную строку в браузере, то откроется сайт google.com.



Чтобы получить все типы записей, нужно дописать ключ ANY

```
$diggoogle.comANY
```



Диагностика задержки в сети с помощью команды traceroute

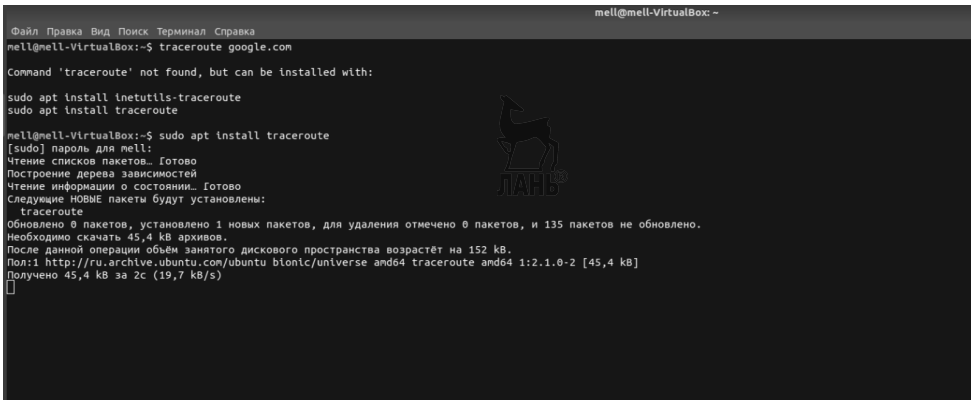
Команда **traceroute** является одной из самых полезных сетевых команд Linux. Он используется, чтобы показать путь к вашей цели и откуда берется задержка. Эта команда помогает в основном в:

– предоставлении имен и идентификаторов каждого устройства на пути;

– выводе сообщения о задержке в сети и определении того, от какого устройства возникла задержка.

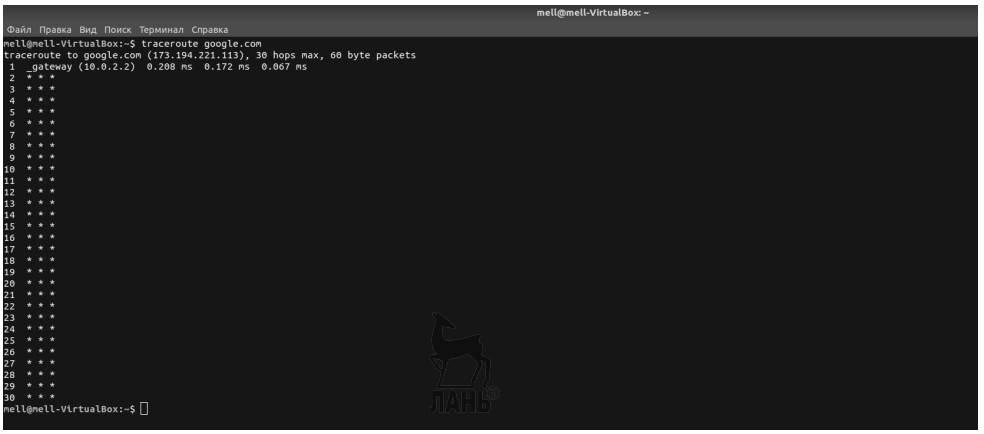
Для начала нужно установить команду **traceroute.**

```
$ sudo apt install traceroute
```



Далее прописываем команду **traceroute**

```
$ traceroute google.com
```



Большой минус этой команды в том, что она не работает, если вы используете VirtualBOX!!!

Пример того, как должна работать данная команда:

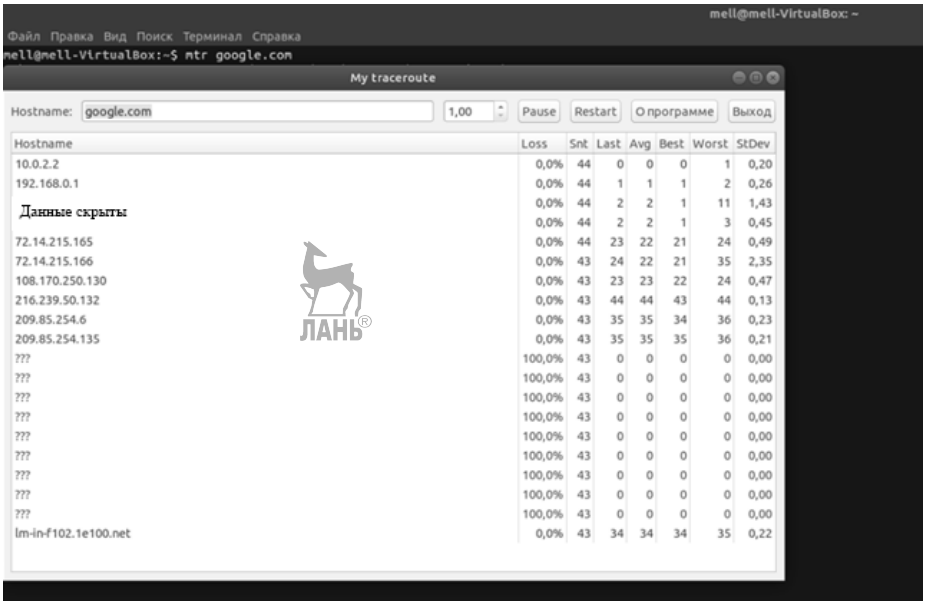
```
➜  ~
➜  ~ traceroute www.bbc.co.uk
traceroute: Warning: www.bbc.co.uk has multiple addresses; using 212.58.246.93
traceroute to www.bbc.net.uk (212.58.246.93), 64 hops max, 52 byte packets
 1  192.168.1.1 (192.168.1.1)  2.917 ms  1.932 ms  1.944 ms
 2  93.155.2.43 (93.155.2.43)  17.166 ms  15.610 ms  14.860 ms
 3  81.212.107.189.static.turktelekom.com.tr (81.212.107.189)  15.821 ms  15.382 ms  16.162 ms
 4  icmeler-ess1-t4-1-palamutrl-ess1-t4-1.turktelekom.com.tr (81.212.202.37)  16.087 ms  15.245 ms  16.222 ms
 5  195.175.174.128.34-erenkoy-t4-1.34-erenkoy-t3-1.statik.turktelekom.com.tr (195.175.174.128)  16.102 ms *  16.607 ms
 6  195.175.174.118.34-ebgp-acibadem1-k.34-acibadem-xrs-t2-2.statik.turktelekom.com.tr (195.175.174.118)  16.471 ms  16.183 ms
    195.175.173.220.34-ebgp-acibadem1-k.34-acibadem-xrs-t2-2.statik.turktelekom.com.tr (195.175.173.220)  17.894 ms
 7  212.156.101.105.301-fra-col-1.34-ebgp-acibadem1-k.statik.turktelekom.com.tr (212.156.101.105)  58.801 ms  52.060 ms  51.865 ms
 8  ffm-b11-link.telia.net (62.115.41.93)  56.407 ms  55.850 ms  55.961 ms
 9  ffm-bb3-link.telia.net (80.91.251.233)  56.336 ms
    ffm-bb3-link.telia.net (80.91.251.55)  55.888 ms
    ffm-bb3-link.telia.net (80.91.254.254)  57.072 ms
10  prs-bb3-link.telia.net (62.115.123.19)  72.977 ms
    prs-bb3-link.telia.net (62.115.123.17)  70.082 ms
    prs-bb4-link.telia.net (62.115.124.199)  73.002 ms
11  ldn-bb3-link.telia.net (62.115.122.250)  70.174 ms
    ldn-bb2-link.telia.net (62.115.114.228)  70.587 ms  70.226 ms
12  ldn-b4-link.telia.net (62.115.134.135)  74.466 ms
    ldn-b4-link.telia.net (62.115.119.145)  70.983 ms
    ldn-b4-link.telia.net (62.115.124.201)  76.658 ms
13  * * *
14  * * *
15  * * *
16  * * *
17  * * *
```

Выходные данные показывают указанный хост, размер пакета, который будет использоваться, IP-адрес и максимальное количество требуемых прыжков. Вы можете увидеть имя хоста, IP-адрес, номер перехода и время прохождения пакета. Звездочки, показанные здесь, означают потерю пакетов.

Команда mtr (трассировка в реальном времени)

Команда **mtr —** динамическая альтернатива команде **traceroute**. Команда **mtr** отображает данные в реальном времени.

```
$ mtr google.com
```

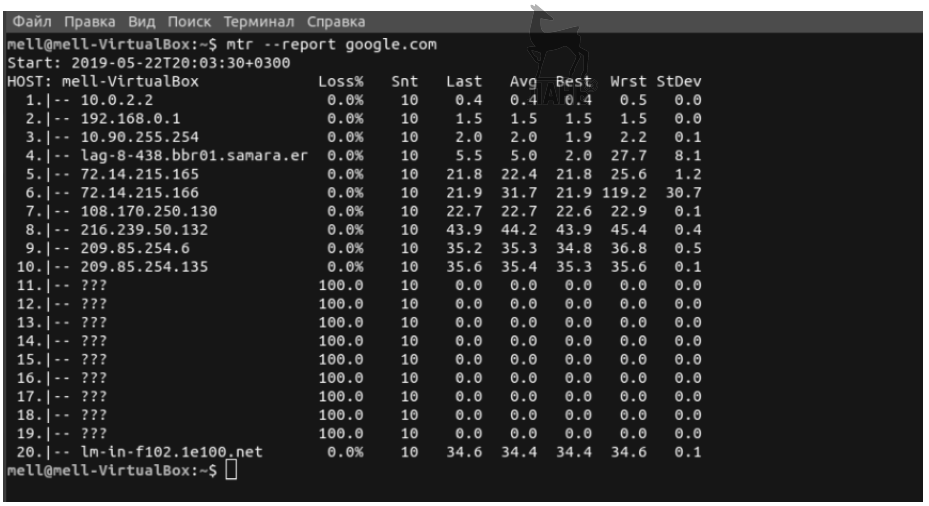


Hostname	Loss	Snt	Last	Avg	Best	Worst	StDev
10.0.2.2	0,0%	44	0	0	0	1	0,20
192.168.0.1	0,0%	44	1	1	1	2	0,26
Данные скрыты	0,0%	44	2	2	1	11	1,43
	0,0%	44	2	2	1	3	0,45
72.14.215.165	0,0%	44	23	22	21	24	0,49
72.14.215.166	0,0%	43	24	22	21	35	2,35
108.170.250.130	0,0%	43	23	23	22	24	0,47
216.239.50.132	0,0%	43	44	44	43	44	0,13
209.85.254.6	0,0%	43	35	35	34	36	0,23
209.85.254.135	0,0%	43	35	35	35	36	0,21
???	100,0%	43	0	0	0	0	0,00
???	100,0%	43	0	0	0	0	0,00
???	100,0%	43	0	0	0	0	0,00
???	100,0%	43	0	0	0	0	0,00
???	100,0%	43	0	0	0	0	0,00
???	100,0%	43	0	0	0	0	0,00
???	100,0%	43	0	0	0	0	0,00
???	100,0%	43	0	0	0	0	0,00
lm-in-f102.1e100.net	0,0%	43	34	34	34	35	0,22

Также можно использовать с командой mtr ключ -report.

```
$ mtr -report google.com
```

Эта команда отправляет 10 пакетов на каждый переход, найденный на его пути.

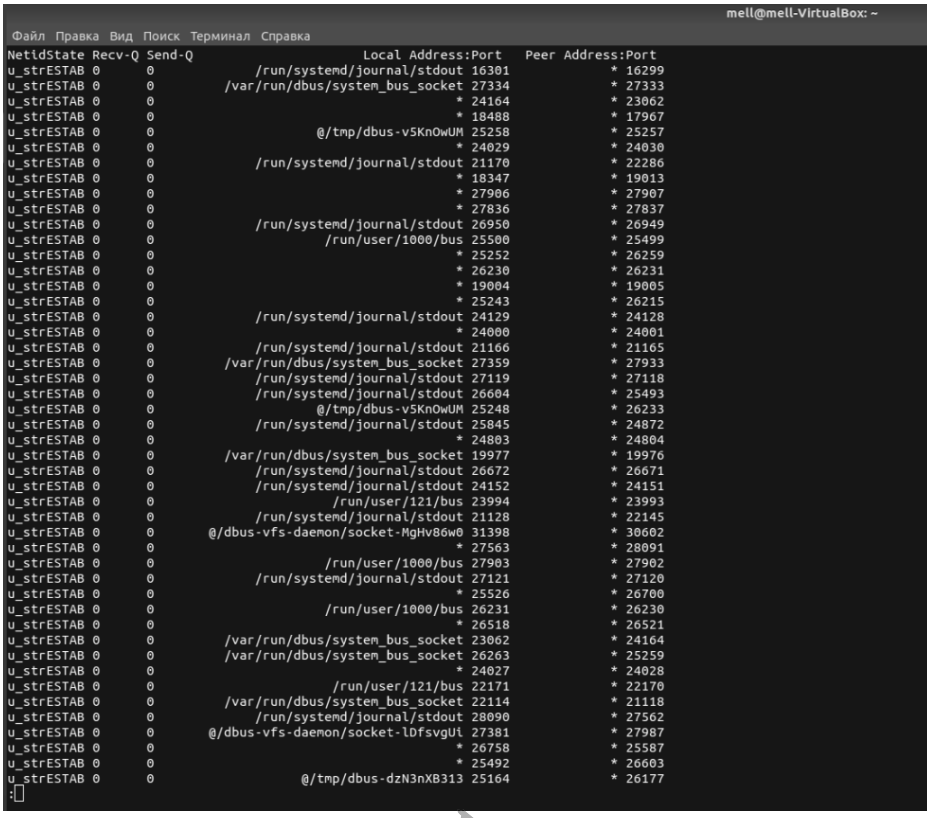


Если данная команда не работает с использованием обычной учетной записи пользователя, то тогда нужно запускать команду через root-пользователя.

Проверка производительности соединения с помощью команды ss

Команда **ss** показывает информацию о вашей сети, а именно: какие IP-адреса используются, какие сетевые, подключенные Linux, открыты, или какие порты прослушиваются. Команда **ss** получает информацию напрямую из ядра.

```
$ ss | less
```



Эта команда выводит все соединения сокетов TCP, UDP и UNIX и передает результат команде less для лучшего отображения.

Также для данной команды есть такие ключи, как:

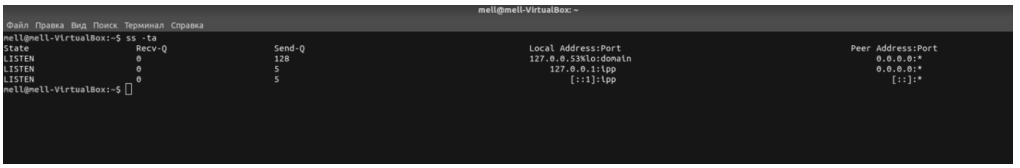
-t для отображения сокетов TCP;

-u для отображения UDP;

-x для отображения сокетов UNIX.

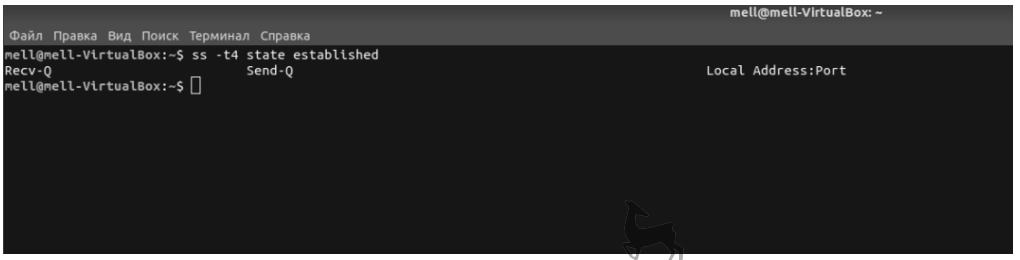
В сочетании с любым из этих ключей нужно использовать опцию -a, чтобы показать подключенные и прослушивающие сокеты.

```
$ ss -ta
```



Чтобы вывести список всех установленных сокетов TCP для IPV4, используйте следующую команду:

```
$ ss -t4 state established
```



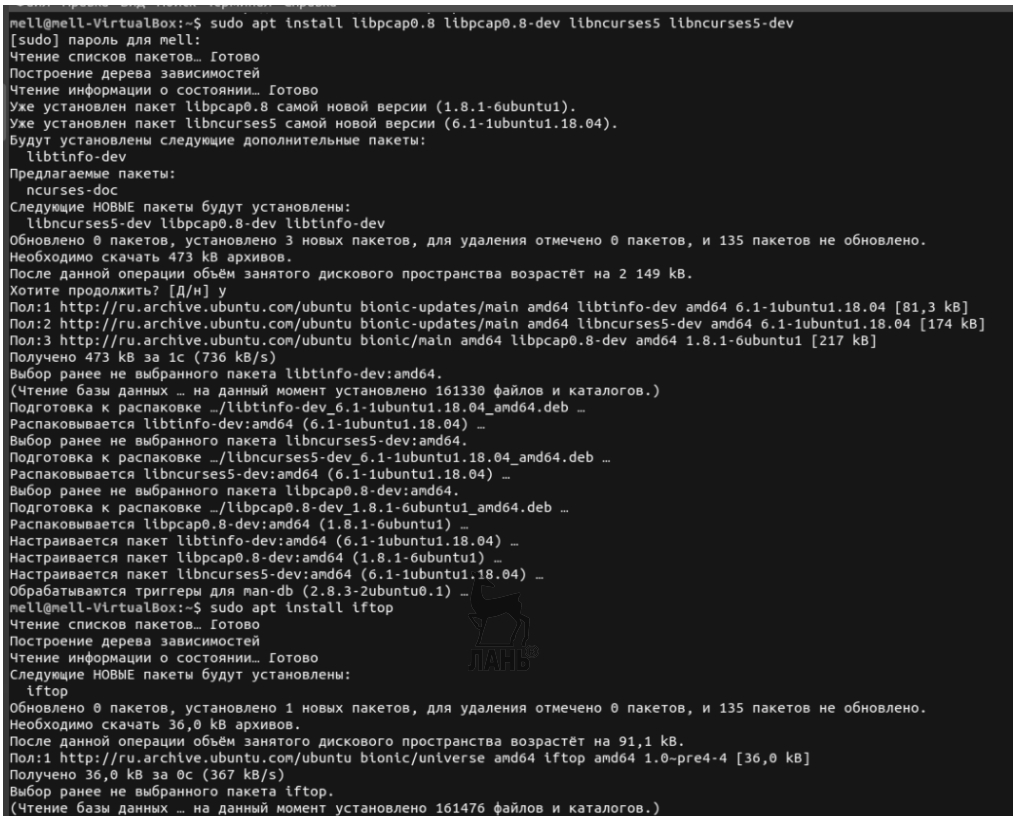
Команда iftop для мониторинга трафика

Команда **iftop** используются для мониторинга трафика и отображения результатов в реальном времени.

Перед тем как начать работать с данной командой **iftop**, ее нужно установить.

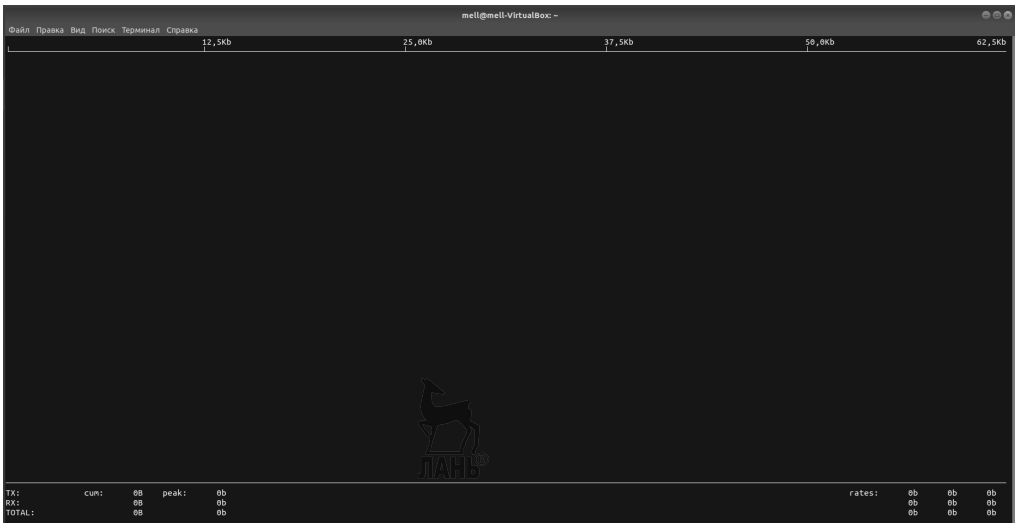
Для этого необходимо прописать две команды:

```
$ sudo apt install libcap0.8 libcap0.8-dev libncurses5
libncurses5-dev
$ sudo apt install iftop
```



После установки можно начать работать с этой командой.

```
$ sudo iftop
```



Так как отсутствует сетевая активность, в терминале ничего не отображается.

Поэтому нужно открыть несколько вкладок в браузере.

В данной таблице будет отображаться весь ваш трафик в реальном времени.

Контрольные вопросы

1. Краткая история семейства протоколов TCP/IP.
2. Общие сведения об архитектуре семейства протоколов TCP/IP.
3. Уровень сетевого интерфейса. Уровень Internet.
4. Протоколы IP, ICMP, ARP, RARP.
5. Internet-адрес. Транспортный уровень.
6. Протоколы TCP и UDP. TCP и UDP-сокеты.
7. Адресные пространства портов.
8. Понятие encapsulation.

9. Использование модели клиент-сервер для взаимодействия удаленных процессов.

10. Организация связи между удаленными процессами с помощью датаграмм.

11. Сетевой порядок байт. Функции htons(), htonl(), ntohs(), ntohl().

12. Функции преобразования IP-адресов inet_ntoa(), inet_aton().

13. Функция bzero().

14. Создание сокета. Системный вызов socket().

15. Адреса сокетов. Настройка адреса сокета.

16. Системный вызов bind().

17. Системные вызовы sendto() и recvfrom().

18. Определение IP-адресов для вычислительного комплекса.

19. Организация связи между процессами с помощью установки логического соединения.

20. Установление логического соединения. Системный вызов connect().

21. Как происходит установление виртуального соединения?

22. Системный вызов listen().

23. Системный вызов accept().

24. Пример простого TCP-сервера.

25. Применение интерфейса сетевых вызовов для других семейств протоколов. UNIX.

Коды ответов серверов и их ошибки

Согласитесь, что, когда мы сидим в Интернете, это невероятно нас захватывает и полностью окунает в мир Interneta. Электронная почта, вконтакте, Instagram, одноклассники и все остальные всемилюбимые социальные сети, Web-узлы почти на любую тему — это все, что необходимо человеку в настоящем мире (этот список можно продолжать до бесконечности), — все это сделало Internet и World Wide Web (Всемирная паутина). Но, к сожалению, не все так хорошо, как могло быть, ибо все равно приходится сталкиваться с проблемами во время каждой «прогулки» по сети. Речь пойдет о различных сбоях и неисправностях подключения к Internet, в частности, к URL. Стоит только вспомнить эту ненастную злую ошибку 404!

Нам, обычным пользователям, кажется, что Internet и World Wide Web очень простые и легкие, но на самом деле это не так: они представляют собой невероятно сложные системы, которые с течением времени могут приводить к серьезным сбоям. И если на пути от некоторого сервера Internet произошел хотя бы один серьезный сбой, адекватный прием данных станет невозможен. Пользователь же всячески должен избегать этого.

В настоящее время все больше и больше происходит стремительный прогресс в развитии аппаратных и программных сетевых средств (обеспечивающих, в частности, соединение с Internet), благодаря которым процесс использования сети стал понятным даже школьнику, но, к большому сожалению, исправления ошибок соединения далеко не продвинулись, т. е. не всегда понятны

для выпускников высших учебных заведений и даже преподавателей, знающих эту сферу очень давно. Сразу возникают вопросы: почему появляются ошибки даже при подключении Интернета? Бывают ли коды серверов также положительными?

Определение источника проблемы

В жизни случается немало ошибок, и поэтому к каждой из них надо искать свой логический подход, который будет полезен при устранении неполадок, в том числе и в сети Internet. Во время устранения неполадок можно задать следующие вопросы:

- Что работает?
- Что не работает?
- Как связаны вещи, которые работают и не работают?
- Работали ли когда-нибудь вещи, которые не работают?
- Если да, то что изменилось с тех пор, как он работал в последний раз?

Ответы на эти вопросы могут указать, где начать устранение неполадок, возможно, позволяя изолировать компонент, слой или проблему конфигурации, которая вызывает проблему.

Коды ответов сервера и ошибки. HTTP 100, 201 и т. д.

Может сейчас эта тема не особо вас волнует, однако всякий раз, когда вы будете посещать различные сайты, вас настигнут различного вида ошибки. Популярностью обладают такие ошибки, как 404 и 301, но существует множество других ошибок, о которых вы не знаете до сих пор.

Многие из них напрямую связаны с сервером или с клиентской стороной, а некоторые уже не актуальны, поэтому их редко встречают в реальной жизни, но ради интереса и фоновых знаний можно рассмотреть парочку кодов сервера.

Начнем с того, что внутренняя ошибка сервера — это класс состояния протокола http, который означает, что операция запроса пользователя выполнена неудачно и виноват в этом сам сервер. На жизненном примере это выглядит так: у вас закончилась еда и нужно идти в магазин за продуктами, а на дверях табличка «учет», и получается, что вы без продуктов. Пользователь хочет зайти на сайт, на сервер отправляет запрос, и если сервер имеет какие-то проблемы, то человек увидит сообщение об ошибке в окне браузера.

Коды ответа делятся по числовым значениям

- **1xx** — запрос получен, но обработка продолжается.
- **2xx** — запрос получен, и, значит, успешная обработка запроса.
- **3xx** — переадресация.
- **4xx** — ошибки клиента.
- **5xx** — ошибки сервера.

Информационные ответы (1xx):

- **100 Continue**

Сервер верно принял запрос и готов к дальнейшей работе, то есть процесс обработки будет продолжен. По-другому, сервер говорит: «Все ОК! Работаем дальше».

- **101 Switching Protocols**

Сервер производит переключение протоколов в соответствии с заголовком Upgrade. Пользователя это никак не касается. Этот код используется редко, поэтому про него можно забыть.

- **102 Processing**

Сервер принял запрос пользователя, но надо будет подождать определенное количество времени. Используется в тех случаях, чтобы пользователь не ушел со странички из-за превышения времени ожидания. При таком получении ответа вы должны дождаться.

Успешная обработка запроса (2xx):

- **200 OK**

Этот запрос очень похож на ответ кода 100, но уже более полный и оконченный, то есть сервер говорит: «Все хорошо! Запрос выполнен». Считается самым главным кодом для успешного выполнения работы серверов.

- **201 Created**

Данный код используется, когда происходит создание нового URL. Если сервер не уверен, что ресурс действительно будет существовать к моменту получения данного сообщения клиентом, то лучше использовать ответ 202.

- **202 Accepted**

Запрос принят и обрабатывается. Обычно можно с этим ответом увидеть дополнительную информацию для решения проблемы.

- **203 Non-Authoritative Information**

Часто злоумышленники пытаются вывести ваш сайт из строя, из-за чего происходит ошибка 203, так как информация получена из ненадежного источника. Поэтому здесь небольшой совет: проверить сайт на вирусы.

- **204 No Content**

Технический момент сайта, поэтому код ответа не сильно важен. То есть запрос обработан, но в ответ ничего не возвращается. Максимум вы увидите заголовки сайта, без его тела.

- **205 Reset Content**

Код выдает, что нужно содержимое сбросить в начальное состояние. Чаще всего используется при очистке ввода данных. В отличие от 204 кода его тут необходимо перезагружать.

Переадресация (3xx):

- **300 MultipleChoices**

Код предлагает несколько вариантов такой страницы. Например, можно было часто увидеть, что сайт создан на нескольких языках и пользователю можно выбрать нужный ему язык.

• **301 Moved Permanently**

Такой код чаще всего выходит по тем причинам, что создан новый сайт этого запроса и он уже не пользуется сервером. Обычно вас сразу переносят на новую страницу, тем самым 301 ошибка склеивает старый сайт и новый.

• **302 Moved Temporarily**

Затребованный URL перемещен, но лишь временно. Ошибка показывает, что сайт имеет новое местоположение. Изначально вы получите документ по новому адресу, а последующие действия будут по-старому.

• **303 See Other**

Данный код показывает, что указанный запрос пользователем лежит в другом месте. Обычно сразу происходит перенаправление на этот сайт, либо в самом ответе сервера будет дана ссылка, на которую надо перейти.

• **304 Not Modified**

Если странички сайта не менялись с последнего посещения пользователя, то сервер дает сигнал роботу о том, чтобы страница не загружалась повторно, тем самым, уменьшится нагрузка на сервер.

• **305 Use Proxy**

Код связан с безопасностью данных. Доступ к документу должен осуществляться через proxy-сервер, который обычно прописан в самом ответе кода.

Ошибки выполнения запроса (4xx):

• **400 Bad Request**

Любая синтаксическая ошибка, препятствующая обработке запроса. Исправление синтаксиса поможет устранить эту проблему.

• **401 Unauthorized**

Ошибка возникает при вводе неверного пароля при авторизации в системе, то есть информация доступна только зарегистрированным пользователем и запаролена.

• **402 Payment Required**

Код еще не воплощен, но по названию можно понять смысл. То бишь сайт используется несколькими популярными серверами, в частности, это бывает youtube, чтобы защитить от вирусов и спама.

• **403 Forbidden**

Запрос пользователя был не принят сервером по какой-либо причине. Некоторые сайты находятся в закрытом доступе с ограниченными возможностями.

• **404 Not Found**

Самая популярная ошибка у пользователей заключается в том, что документ просто не найден.

• **405 Method Not Allowed**

Означает, что метод, используемый клиентом, не поддерживается. Например, при попытке отправить POST-данные документу, который не является скриптом.

• **406 Not Acceptable**

Ресурс существует, но в другом, не в том формате, который запрашивает пользователь.

• **408 Request Time-out**

Время ожидания сервером передачи от клиента истекло. Клиент может повторить аналогичный предыдущему запрос в любое время.

• **409 Conflict**

Данный запрос конфликтует с другим запросом или с конфигурацией сервера. Можно почитать и быстро устранить.

• **410 Gone**

Этот документ вы больше не увидите, так как он навсегда удален с сервера.

• **411 Length Required**

Пропущено необходимое поле в заголовке запроса Content-Length.

• **412 Precondition Failed**

Возвращается, если ни одно из условных полей заголовка запроса не было выполнено.

• **413 Request Entity Too Large**

Слишком большой запрос.

• **415 Unsupported Media Type**

Сервер не поддерживает указанный формат данных. Не поддерживает и не планирует, эта ошибка почти неисправима.

• **416 Requested Range Not Satisfiable**

Сервер сообщает — форма запроса (требуемый диапазон) не выполнима.

• **417 Expectation Failed**

Время ожидания истекло.

Ошибки сервера (5xx):

• **500 Internal Server Error**

Код означает, что происходят ошибки внутри сервера, и он не может получить доступ к серверу. Исправляется программистами.

• **501 Not Implemented**

Недопустимое действие. Ошибка возникает, если распространенные протоколы не поддерживаются сервером.

• **502 Bad Gateway**

Недопустимый ответ с другого ресурса. Для исправления нужно проверить настройку прокси-сервера.

• **504 Gateway Time-out**

Код похож на ошибку 502, но здесь означает, что срок ожидания от сервера истек. Превышен тайм-аут ожидания от другого ресурса.

• **505 HTTP Version not supported**

Данная версия протокола HTTP не поддерживается сервером.

Разбор частных ошибок в серверах

Думаю, каждый сталкивался с такими ошибками на сервере, как 400, 403, 404, 500, 502, 503. Интересно ли было бы вам узнать, как это происходит? Есть ли решения на эти ошибки? И на что может влиять тот или иной код? Рассмотрим максимально детально алгоритм этих ошибок, приведя также примеры из жизни для более глубокого понимания.

Ошибка 400

Начнем с того, что ошибка 400 распространена и на сегодняшний день. В оригинале ее название «Bad request», что в переводе с английского «Плохой запрос».

Данная ошибка бывает в тех случаях, когда пользователь посылает неверный запрос серверу, где находится ресурс. Значит причина кроется не в создании сервера, а у самого пользователя. Сбои можно исправить несколькими способами, такими как настройки браузеров, установленные программы на компьютере и также антивирусы.

Для устранения ошибки 400, во-первых, можно открыть сайт через другой браузер; во-вторых, почистить кэш и файлы-куки, это делается легко в настройках браузера; в-третьих, переустановить браузер либо обновить его до новой версии если данные способы не помогли, то переходим к следующим более сложным действиям. Данная ошибка может происходить из-за антивируса, есть несколько способов ее исправить. Во-первых, проверить полностью компьютер на вирусы; во-вторых, можно попробовать отключить антивирус, ибо часто встречалось, что Dr.Web блокирует некоторые сайты, таким образом, отключив антивирус, откроете сайту более надежный доступ к нему; в-третьих, можно заменить антивирус, но это уже в крайних случаях.

Ошибка 403

Знаете ли вы, почему пользователь видит эту ошибку на своей странице? Ниже представлена парочка причин: во-первых, существуют сайты, к которым доступа у пользователя нет; во-вторых, также бывает так, что на сайт зайти можно, но на отдельную страницу у вас также нет полного комплекса прав доступа к сайту.

Эта ошибка у разных пользователей отображается по-разному на экране. Обычно встречаешь такие сообщения, как «Error 403», «Error 403 forbidden», «HTTP error 403 forbidden», также можно увидеть на русском языке, зависит от того, на каком из движков работает сайт.

Рис. 36

Пример того, как выглядит ошибка 403

Обычно сервер выдает ошибку 403, когда в какой-либо раздел вы пытаетесь зайти, но администратор вас не знает и не решается пускать. Возьмем вновь пример из жизненной ситуации: в квартире находилось два человека, ко-

гда вы вышли, забыли ключи и захлопнули дверь, единственный вариант — попросить соседа, оставшегося внутри, открыть дверь. Аналогично и с ошибкой 403, если вы обычный пользователь без прав, то никак не сможете попасть на сайт. Один из оптимальных вариантов — можно обратиться к администрации сайта и потребовать, чтобы вам дали какие-либо права к сайту/странице. Случается такое, что у вас были права, но ошибка все равно вылезла. Что же делать в этом случае? Можно предположить, что администратор вас забанил за нарушения/случайно, поэтому также стоит обратиться за помощью к нему. В таких ситуациях обычно заводят новую учетную запись, если требуется логин и пароль, или в крайнем случае меняют компьютер.

Ошибка 404

Что такое 404 код ответа? Если мы посмотрим в справку, то увидим, что данный ответ отдаётся сервером, когда данная страница (данный URL), который запрашивается на сервере, отсутствует, то есть сервер данную информацию не нашёл. Если мы усугубимся в технические детали, это означает следующее: человек вводит в адресную строку адрес, этот адрес отправляется на сервер, пытается по данному адресу запросить документ, говорит: «Есть такой документ?». Техническая сторона сервера ищет такой документ и говорит: «Нет, не найден». Отправляется это все в браузер клиента и там, на той стороне, отображается 404 страница, которая содержит в себе 404 код ответа сервера: данный материал по данному URL не найден на сервере. Обратите внимание на слово «не найден». Есть еще 410 код ответа сервера, который означает, что страница была удалена. Представьте, что у вас на сайте 20 нормальных страниц, а 200 — с ошибкой 404, робот Яндекса их все время скачивает, пытается определить, что там есть, как изменилось, и это может негативно сказываться на продвижении сайта.

Ошибка 500

Когда поисковый робот обращается к вашему хостингу, к серверной части вашего сайта и сервер не может обработать этот запрос, сервер отдает клиенту ответ: «Я этот запрос обработать не могу, вот тебе код ответа 500». Это значит, что какие-то неполадки внутри сервера, это не значит, что страницы нет, но и не значит, что она есть. Это означает, что сервер сам не работает, поэтому поисковые роботы, получая код ответа 500, помечают этот URL в своей базе, как URL, требующий переобхода, т. е. это означает, что сервер через минуту, две, три отвиснет и сможет отдать, что же поэтому URL есть, либо 200 код ответа (означает, что все в порядке, сайт загружен), либо 404 (такой сайт не найден, его не существует). Другими словами, можно сказать, что код ответа 500 — маяк для пользователя «зайди попозже, я занят» или «я сейчас не работаю», но, с другой стороны, если ваш сервер очень долго отдает код ответа 500, то страницы все равно выпадут из индекса, то есть робот Яндекса через какое-то время, когда он получает все время эту ошибку 500, начинает эти страницы выкидывать из индекса, то есть принимать, будто их уже нет. Таким образом, ваш сайт также может упасть из выдачи, поэтому необходимо правильно настраивать все коды ответов сервера, в том числе и 500, проверять нагрузку

сервера, проверять, как он реагирует на повышенные нагрузки, как реагирует в экстремальных условиях. Если сайт выпадет из индеса, это может повлиять на продвижение, потому что о нем не знает поисковая система.

Рис. 37
Пример того, как выглядит ошибка 500

Ошибка 502

Данная ошибка означает, что запрос, отправленный от пользователя к серверу, не достиг точки адресата назначения.

Новый пример из жизни: представьте, что вам посчастливилось сходить на концерт своего любимого кумира, который известен на весь мир, и к нему очень тяжело попасть, кроме вас есть еще миллион таких же людей, которые хотят к нему на концерт. Но единственный минус, что пропускают только 50 человек. Фейс-контроль пропускает их без проблем, а остальным приходится ждать, когда из клуба выйдет хотя бы один человек. Иначе сотрудники не будут успевать с таким количеством заказов, но кроме этого сохранять порядок в зале будет гораздо труднее.

Так же и с ошибкой 502: сервер не впускает новых пользователей до тех пор, пока не освободится «память» сайта (в количестве «50 человек»), по-другому можно сказать, что включается внутренняя защита.

Любой сервер содержит свой лимит по количеству пользователей сайта. Превышение максимального количества как разтаки и приводит к ошибке.

В одно время зашли много пользователей

Часто случается так, что пиарный ход работает очень хорошо, и на сайт привлекают так много посетителей, что сайт просто не выдерживает такой нагрузки. Можно привести пример: новый сезон «Игры престолов» ждали люди из разных стран. Тысячи людей, если не миллионы! Несмотря на то что выход фильма был в 4 утра, посетители настолько сильно хотели посмотреть новый сериал, что сайт не выдержал и просто не грузил.

Также есть умные хакеры, которые часто пытаются взломать ваш сайт для намеренного отключения вашего портала.

Плохо настроен сервер

Бывает, что перед тем как появилась ошибка шлюза 502, вы обновляли сайт, это и повлекло за собой проблемы несовместимости некоторых утилит или аппаратной части с программной.

Проблемы подключаемых модулей

Если ошибка обращается к различным скриптам, например, к PHP, то причина кроется, скорее всего, в неправильной работе модуля сервера PHP.

Рис. 38
Пример того, как выглядит ошибка 502

Ошибка 503

Одна из тех ошибок, которая встречается редко, но метко. Данная ошибка показывает, что сайт больше не принимает новые требования по введенному адресу, сервер, на котором располагается этот сайт, не должен превышать ограниченного запроса к нему. Появление ошибки 503 зависит от мощности, которую способен выдержать сервер: если на него отправляется большое количество запросов, то сайт по такой причине перестает работать. Допустим, что вы стоите в очереди за молоком в советское время, перед вами огромное количество людей и все они совершают один и тот же запрос, то есть хотят получить молоко. Продавец в очереди обрабатывает их запросы, и случается так, что все молоко заканчивается, лимит исчерпан, магазин закрывается на ПЕРЕРЫВ, оставшиеся люди в очереди уходят, так и не купив молока, но, как и во всех магазинах, запасы пополняются через какой-то период времени, и товары будут продаваться дальше. Аналогично работает и хостинг. Хостинг назовем магазином, пользователями будут люди, стоящие за молоком, запрос — купить желанное молоко, ошибка 503 — это табличка ПЕРЕРЫВ. Обычно ошибка 503 временная, сервер обрабатывает текущие запросы, освободив свою мощность для следующих запросов пользователя. Если удаленный сервер так и не открыл вам нужную страничку, то это значит, что сейчас очень большая очередь и он просто не может ее обработать. Причин может быть несколько, одна из них — зависли скрипты, так как к серверу сейчас направлено много обращений; решением данной проблемы должны заниматься администраторы сайта и его владелец, либо веб-мастер. Требуется автоматизация его работы. Можно привести пример из реальной жизни, когда ошибка сервера 503, наоборот, была полезна. Один сайт подклеили к другому сайту для того, чтобы не настраивать 301 код сервера (другими словами, чтобы не перенаправлять пользователя с одного сайта на другой). В силу того, что на сайте, который подклеен, были плохие отзывы, грубо говоря, об этом бренде было много нехороших отзывов, и если бы

сделали 301 код, то все бы попадали на эти отзывы, но использовать сайт для продвижения можно было, так как вес был достаточно высок, поэтому его подклеили, и для того, чтобы дальше не отклеивался, на него «подвесили» 503 код ответа. Получается, что сервер недоступен и непонятно, что там есть.

Причины подобных ошибок зависят либо от администратора, который обслуживает сайт, либо от тех, кто создавал этот сайт. Поэтому не стоит переживать, и ваш сайт в скором времени восстановят.

Методы организации безопасности в операционных системах

Информационная безопасность — молодая область информационных технологий, которая развивается быстрыми шагами. Под информационной безопасностью понимают защищенность информации, то есть целостность и секретность информации от случайных или направленных действий, которые могут нанести ущерб безопасности. Непосредственно защита информации — это мероприятия, направленные на обеспечение безопасности информационной системы.

Ценность информации определяется степенью ее полезности. Закон Российской Федерации гарантирует право собственника информации на использование и защиту от доступа к ней других лиц. Информация с ограниченным доступом называется конфиденциальной. Такая информация может содержать в себе государственную, коммерческую или военную тайну.

Целью организации мероприятий по защите информации является обеспечение сохранности данных. К ним мы можем отнести файлы и документы, которые пользователь создает, редактирует, удаляет. В файлах может содержаться различная информация, несущественная утрата которой ни на что не повлияет, а также информация, утрата которой приведет к серьезным последствиям.

В представленной статье проведен анализ различных уровней безопасности на основе материалов, имеющихся в свободном доступе в ресурсах Интернета. Работа была выполнена совместно с обучающимися на курсе «Операционные системы» и представляет интерес с 2-х точек зрения: во-первых, как пример привлечения студентов к творческим проектам, во-вторых, для формирования современного учебно-методического комплекса для представленной дисциплины.

Существует несколько уровней, обеспечивающих безопасность [1]. Но так как работа проводилась в рамках курса «Операционные системы», мы остановимся на одном из уровней, которым является как раз операционная система.

Операционная система — совокупность программ, управляющая ресурсами системы для эффективного использования и обеспечения интерфейса пользователя ресурсами. Развиваясь, ОС прошли несколько поколений.

В первом поколении ОС добивались ускорения и упрощения перехода с одной задачи на другую. Появилась проблема обеспечения безопасности данных различных задач.

В 1960 г. начали переходить к мультипроцессорной организации, и проблемы защиты ресурсов стали трудноразрешимыми. Результат решения этих проблем привел к другой организации и к применению аппаратных средств защиты, например, таких как защита памяти.

Одним из основных направлений развития вычислительной техники является ее максимальная доступность для пользователя, что противоречит принципам обеспечения безопасности данных. Под безопасностью операционной системы мы понимаем состояние ОС, при котором нельзя нарушить ее функционирование.

У угроз безопасности операционным системам в настоящее время нет единой классификации. Но их можно классифицировать по признакам [2].

По цели воздействия на операционную систему:

1) взлом, изменение, удаление данных;

2) полное или частичное разрушение операционной системы.

По принципу воздействия на операционную систему:

1) использование каналов для получения информации;

2) использование возможностей операционной системы;

3) создание утечек при помощи программ.

По характеру воздействия на операционную систему:

1) активное;

2) пассивное.

Использование слабости защиты:

1) ошибки администратора системы;

2) ошибки и не оглашенные возможности ОС;

3) внедрение программы.

По способу воздействия на объект атаки:

1) непосредственное воздействие;

2) превышение пользовательских полномочий;

3) вход под другим пользователем;

4) перехват потоков информации.

По способу действий злоумышленника:

1) вручную;

2) в пакетном режиме.

По объекту атаки:

1) ОС;

2) файлы, папки;

3) системные процессы, пользователи;

4) каналы.

Главной проблемой обеспечения безопасности операционной системы является контроль доступа к ресурсам системы. Для решения этой проблемы операционная система должна иметь собственную вспомогательную защиту (средства мониторинга, контроль и аудит и т. д.).

В основном встречаются следующие атаки на ОС.

Сканирование системы. Злоумышленник пытается прочесть, скопировать, удалить файлы. Если злоумышленнику не удастся прочесть файл, то, продол-

жив сканирование, может найти ошибку и получить доступ к конфиденциальным данным.

Попытки взлома пароля. Выделяют две основные угрозы: кража и подборка.

Существует несколько вариантов защиты операционной системы от кражи паролей:

– не выводить пароль на экран (можно выводить вместо него символы («*****») или же не выводить никакую информацию как в unix);

– блокировать ввод пароля с командной строки;

– использовать в пароле знаки разных типов.

Проинструктировать пользователей операционной системы:

– о неразглашении паролей;

– сменить пароль, если этот пароль узнали другие пользователи;

– после истечении какого-то времени обязательная смена пароля;

– хранение паролей на бумажном или электронном носителе.

На основании вышесказанного в 1975 г. были определены принципы создания защищенных систем:

- Структура системы должна быть прозрачной, т. е. не засекречена.
- Доступ к системе не должен быть открытым, по умолчанию «запрещено».
- У процесса не должно быть много привилегий.
- Защита должна быть простой и находится в нижних уровнях.
- Архитектура системы должна быть создана одним программистом.

Существуют фрагментарный и комплексный подход создания защищенной операционной системы.

При фрагментарном подходе система первым делом защищается от одной угрозы, только потом от другой. Фрагментарный подход защиты используется в операционной системе Windows 98, поверх которой устанавливаются антивирусы и т. д. Недостатком является то, что защитные программы созданы разными представителями. Программные продукты работают независимо, организовать взаимодействие этих программ становится сложной задачей.

При комплексном подходе — защита системы вкладывается в ОС во время проектирования архитектуры и является частью операционной системы. Плюсом такого подхода является то, что защита, созданная комплексным подходом, взаимодействует друг с другом и более надежно помогает организовывать защиту информации. Во время разработки подсистема проходит проверку на совместимость, и конфликтов между частями подсистемы защиты не происходит. Главной фишкой такой подсистемы защиты является то, что при сбоях, вызванных злоумышленником, система вызывает крах операционной системы и не позволяет выключить систему защиты.

Рассмотрим методы организации защиты в операционных системах Unix. Операционная система UNIX вначале создавалась как открытая ОС, некоторым элементам характерна открытость, из-за этого при попытке конфигурации случаются такие моменты, когда может быть нарушена работа механизмов безопасности системы. Отсюда следует, что важным фактором обеспечения безо-

пасности в операционной системе UNIX является правильная конфигурация и настройка.

В одной процессорной системе защита целостности структуры данных ядра обеспечивается двумя способами: 1) ядро не может вначале выгрузить один процесс и затем переключиться на другой, в том случае если работа производится в режиме ядра; 2) если возникнет прерывание, в тот момент, когда выполняется критический участок программы, то это может повредить структуру данных ядра.

В многопроцессорной системе, при выполнении нескольких процессов одновременно и при том в критических интервалах в режиме ядра, может нарушиться целостность ядра, если даже были приняты защитные меры.

В операционной системе UNIX роль администратора разделяется на несколько частей, каждая часть которой ответственна за что-то свое. Эта идея о ролях администратора является основой в безопасности операционной системы. Операционная система Linux является ядром UNIX-подобной системы, созданная Линусом Торвалдсом с помощью огромного числа людей, найденных в сети интернет. ОС Linux обладает многозадачностью, имеет хорошо развитую подсистему управления памятью. Секретом мощной интегрированной системы защиты в ОС LINUX является код фаервол, который встраивается в ядро LINUX.

В ОС LINUX используется 3 основных механизма защиты паролей:

1) пароли шифруются;

2) механизм теневых паролей;

3) PAM-модуль подключаемой аутентификации.

Целостность данных в операционной системе LINUX контролирует пакет TRIPWIRE. Этот пакет сначала вычисляет сумму двоичных файлов и конфигурационных файлов, а затем сравнивает их значения с хранящимися в базе. Администратор может контролировать системные изменения.

Для контроля целостности данных в Linux используется пакет Tripiwire. При запуске он вычисляет контрольные суммы всех основных двоичных и конфигурационных файлов, после чего сравнивает их с эталонными значениями, хранящимися в специальной базе данных. В результате администратор имеет возможность контролировать любые изменения в системе.

Для обеспечения сетевой безопасности в ОС Linux применяются следующие средства:

– защищенная оболочка ssh для предотвращения атак, в которых для получения паролей используются анализаторы протоколов;

– tcp_wrapper программа ограничивает доступ к различным службам;

– сетевые сканеры, выявляющие уязвимые места компьютера;

– демон tcpd для обнаружения попыток сканирования портов со стороны злоумышленников;

– PGP (Pretty Good Privacy) система шифрования;

– программа qmail — защищенная доставка сообщений;

– режим проверки паролей входных соединений для систем, разрешающих подключение по внешним коммутируемым линиям связи или локальной сети.

Анализ современных операционных систем. Обзор отечественных операционных систем

Развитие компьютерных технологий происходило стремительно. Интерес людей к IT-сфере не угасал, начиная с 1940 г., когда была создана первая цифровая вычислительная машина, заканчивая настоящим временем, когда испытания искусственного интеллекта идут полным ходом. Сейчас сложно представить компьютер без операционной системы, ведь она является одной из главных составляющих частей компьютера, без нее компьютер был бы крайне трудоемким в плане управления, однако история операционных систем знает и такие времена.

Необходимо отметить, что развитие отечественных IT-технологий происходило в разгар «холодной войны», когда Советский Союз был временно изолирован от каких-либо взаимодействий с другими странами. Вследствие этого невозможно было осуществлять совместные исследования и обмениваться опытом с иностранными специалистами. Из-за информационной изолированности неизбежно складывалась ситуация, когда одни и те же изобретения появлялись по обе стороны «железного занавеса» практически одновременно.

Однако даже в данной ситуации были положительные стороны — правительство страны выделяло значительные средства для создания сильнейшей команды разработчиков, которая бы создавала все быстрее и качественнее, чем в других странах. Советские IT-разработки велись в области программного и аппаратного обеспечений, особый акцент делали на компиляторы и операционные системы.

После эпохи оттепели, когда взаимодействие с другими странами возобновилось, обнаружилось, что отечественные специалисты, не зная ничего об аналогичных работах зарубежных коллег, создавали свои оригинальные ОС. К заслугам советских разработчиков можно отнести идею многопоточности, которая впервые была реализована в операционной системе «Эльбрус» еще в конце 1970-х гг. (в зарубежных ОС многопоточность появилась только в конце 1980-х — начале 1990-х гг.).

Наряду с плюсами имелись и минусы, к примеру, существенное отставание советских и российских IT-специалистов в области графических пользовательских интерфейсов, а также в области разработки элементной базы и технологии производства компьютеров.

Разберемся, что же такое ОС и для чего она нужна. Операционная система — это системное программное обеспечение, которое управляет компьютером, аппаратными и программными ресурсами, а также предоставляет общие услуги для компьютерных программ. Именно благодаря операционной системе возможно получить доступ к устройствам и выполнять различные функции. Операционная система — посредник между компьютером и пользователем, она передает инструкции, например, процессору компьютера. Процессор выполняет порученные от операционной системы ему задания, а затем отправляет результаты обратно в приложение через операционную систему. Благодаря ОС управление компьютером стало более удобным и понятным для пользователей.

Вернемся к истории. Первая известная из отечественных операционных систем появилась в 1950–1960 гг., это была ОС ДИСПАК для ЭВМ БЭСМ-6. В это же время были сформированы и реализованы главные идеи, определяющие функциональное назначение ОС. К основным возможностям классических операционных систем относят:

– мультипрограммирование — одновременная обработка нескольких заданий;

– пакетная обработка — режим работы, который предполагает наличие очереди программ на исполнение, что позволяет избежать простоя процессора;

– разделение времени — параллельная работа нескольких терминалов, что позволило создать многопользовательские системы;

– управление процессами — параллельное (или попеременное, при условии однопроцессорного компьютера) выполнение пользовательских процессов;

– файловые системы и структуры — способ хранения данных на внешних запоминающих устройствах; процесс замены носителей с последовательным доступом (перфолент, перфокарт и магнитных лент) накопителями произвольного доступа (магнитные диски).

Разработка каждой ОС требовала многих лет высококвалифицированной работы специалистов, ведь первоначально каждая операционная система разрабатывалась на языке ассемблера и только затем реализовывалась на более высокоуровневых языках.

Под руководством основоположника советской отрасли вычислительной техники, Сергея Алексеевича Лебедева, в Институте точной механики и вычислительной техники АН СССР была разработана БЭСМ-6, а также осуществляемые на ней операционные системы — ДИСПАК и ДИАПАК. Данные операционные системы поддерживали страничную организацию виртуальной памяти, пакетный и диалоговый режимы взаимодействия с компьютером, работу в локальных сетях, работу с внешними устройствами и телекоммуникационными каналами.

К каждой БЭСМ-6 были подключены десятки терминалов, работавших под управлением диалоговых систем ДИМОН и ДЖИН. Интересный факт, что объем оперативной памяти БЭСМ-6 был всего в 32 страницы по 4096 байтов, а быстродействие достигало до 1 миллиона операций в секунду. Работу БЭСМ-6 и ее операционных систем ДИСПАК и ДИАПАК отличала высокая надежность, обеспечиваемая уникальным строением модулей машины.

В 1960-х гг. возникла идея разработки мобильных ОС — это операционные системы, которые предназначены для обеспечения функционирования компьютеров различных производителей, такие ОС состоят из ядра, к которому добавляются различные модули сервиса, а также на них возможен перенос кода с более старой модели компьютерного семейства на более новую модель.

В 1970–1980 гг. в СССР велась двойная игра в сфере IT-разработок, с одной стороны, были собственные разработки, такие как «Эльбрус», с другой стороны, правительство направило немалые деньги на осуществление и адаптацию ЭВМ американских IT-специалистов.

Начнем с уникальной разработки, о которой упоминалось выше, она заслуживает отдельного внимания — это многопроцессорный вычислительный комплекс (МВК) «Эльбрус», разрабатываемый в 1970–1980-х гг. и имеющий две комплектации — «Эльбрус-1» и «Эльбрус-2». За рубежом существовало множество аналогов и прототипов, заложивших академические основы архитектуры данных машин и их операционных систем. Примером может служить серия компьютеров фирмы Burroughs. Однако только в МВК «Эльбрус» впервые была создана операционная система с рядом отличительных черт. Главная особенность МВК «Эльбрус» — реализация функции многопоточности, которая:

– во-первых, упрощала программы за счет вынесения процессов чередования выполнения подзадач, требующих одновременного выполнения;

– во-вторых, повышала производительность механизма за счет распараллеливания процессорных вычислений и операций ввода-вывода.

Помимо этого, ОС «Эльбрус» была динамической, это значит, что все программные сегменты загружались в память мгновенно, при первом же вызове. Также «Эльбрус» поддерживала математическую (виртуальную) память и распределение оперативной памяти, что делало данный многопроцессорный вычислительный комплекс передовым для своего времени.

Параллельно с разработкой «Эльбруса» руководство СССР приняло решение о создании отечественной серии компьютеров Единая Система ЭВМ (ЕС ЭВМ), примером заимствования послужила разработка американских программистов, которые разработали серию компьютеров IBM 360.

Адаптация системы, а также ее программного обеспечения и операционной системы, требовала больших вложений, вследствие чего у отечественных разработчиков возникли финансовые трудности. Первые пользователи ЕС ЭВМ испытывали сложности в освоении и использовании системы, так как существовал языковой барьер, незнание английского языка сильно ограничивало юзеров.

Для решения данной проблемы пользователей русские разработчики изобрели так называемые системы-обертки, которые служили промежуточным слоем между прикладной программой и интерфейсом программ, которые запускал пользователь. Благодаря оберткам все процессы операционной системы ЕС преобразовывались, пользователь получал уже конечный результат (переведенный текст процессов).

Операционная система ЕС ЭВМ представляла собой доработанный и русифицированный вариант OS/360 и OS/370, она обеспечивала пакетную обработку заданий, для написания которых применялся язык JCL (Job Control Language). До версии 6 применялись следующие режимы работы, операционной системы ЕС при запуске системы:

– ОС ЕС РСР — система однозадачного характера;

– ОС ЕС MFT — система многозадачного характера с предопределенным числом задач. Такая ОС осуществляла разделение памяти на две части, каждая из которых могла выполнять свою задачу;

– ОС ЕС MVT — система многозадачного характера с варьируемым числом задач. Для каждой задачи выделялся необходимый «кусок» динамической памяти.

Начиная с 6 версии, режим систем однозадачного характера не применялся, но стал доступным новый — SVS, в котором была доступна виртуальная память размером до 16 Мбайт, также пользователь мог осуществлять размещение страниц подкачки на жестких дисках.

Несмотря на все усилия русских IT-специалистов, ЕС ЭВМ так и не прижилась в СССР, основная причина заключалась в том, что полной русификации системы так не добились, поэтому большинство компаний устанавливали IBM 360, аргументируя свой выбор дополнительной надежностью системы.

Далее правительство СССР предприняло попытку заимствования американских мини-компьютеров серии PDP10 и PDP11, которое известно в России под названием «Система Мини-ЭВМ»: данная линейка имела компьютеры СМ-1, СМ-2, СМ-3 и СМ-4. Для серий машин СМ ЭВМ было создано большое количество операционных систем, каждая из которых нашла свое применение в различных сферах общества:

- ФОБОС — поначалу ОС однозадачного характера, впоследствии многопользовательская среда с разделенным захватом ресурсов. Основная цель данной ОС — узкая ориентированность на выполнение программ и приложений общего типа. ФОБОС поддерживала много языков программирования, такие как Fortran, позже Pascal, Lisp и C.
- МДОС (Малая Дисковая Операционная Система). Изначально МДОС работала только с дискетами, на которых располагались сама МДОС, а также программные компоненты и данные пользователей. В следующих версиях появилась возможность использования CD-ROM, жестких дисков.
- ДИАМС — операционная система для поддержания многотерминального режима в операционных средах. Основное назначение — редактирование и корректировка баз данных иерархического характера.

Данные машины и их операционные системы до сих пор используются в России, например, в сельском хозяйстве, медицине, промышленности и даже в центрах подготовки космонавтов.

Выше перечислены только самые известные из зарубежных архитектур и их ОС, которые получили русские аналоги в 1970–1980 гг., было много других адаптаций, которые не получили широкой огласки. В такой политике заимствования есть свои плюсы и минусы: плюс — освоение отечественными IT-разработчиками новых языков программирования, операционных систем, архитектуры ЭВМ в целом; минус — пытаясь сделать по примеру, русские IT-специалисты не могли изобретать и разрабатывать собственные машины.

В течение 1980–1990 гг. развивалось семейство переносимых, многозадачных и многопользовательских систем UNIX, которые впервые разработаны еще в 1970-х гг. К особенностям данных систем можно отнести:

– одновременная работа группы людей, у каждого из которых свой терминал;

– мультиплатформенность, система не привязана к определенному микропроцессору, является мобильной;

– наличие сервисных программ, облегчающих пользование другими программами (утилиты);

– управление системой через простейшие текстовые файлы;

– наличие программных конвейеров, выполняющих определенные задачи.

Резкий рост разнообразия ОС UNIX вынудил руководство компании «AT&T» принять стандарт «Single UNIX Specification», который разделил все аналоги на UNIX и UNIX-подобные. Система была разработана американскими специалистами, но имеет много доработанных вариаций от русских разработчиков, их мы рассмотрим чуть позже.

Новый этап развития компьютеров — изобретение больших интегральных схем (БИС), разработка стека TCP/IP, развитие Всемирной паутины. Все это привело к появлению персональных компьютеров и, как следствие, операционных систем для этих ПК. Началась эра персональных компьютеров, пользователями становились люди, которые никак не разбирались в архитектуре ЭВМ, это потребовало создание «дружественного» программного обеспечения и ОС, которые были бы понятны даже неспециалистам.

Первой версией ОС для персонального компьютера стала MS-DOS от компании Microsoft. Данная операционная система была громоздкой, отсутствовали сетевые функции, ее главная задача — манипуляция файлами, находившихся на жестких и гибких дисках файловой системы, также MS-DOS регулировала поочередный запуск программ. В 1985 г. компания Microsoft сделала так называемую «надстройку» к MS-DOS и на свет появилась самая известная во всем мире операционная среда Windows, в первой версии которой разработчики осуществили процесс поддержки мультипрограммных систем.

В 1990-е гг. все операционные системы носят характер сетевых или корпоративных, ОС получают средства для работы с локальными и глобальными сетями, а также могут создавать составные сети. Во второй половине 1990-х разработчики ОС делают акцент на поддержку средств работы с сетью Интернет, так как именно в это время она получает все наибольшее распространение. С 2000-х гг. наиболее распространенными операционными системами являются системы семейства Microsoft Windows и системы класса UNIX (особенно Linux и Mac OS).

В России же с 2000 г. до нашего времени создано много дистрибутивов, в основном на базе ОС Linux. Имеются различные версии, одни используются на технических предприятиях и в медицине («Альт»), другие — в федеральных службах и ВВС России («Ось», «МСВС»), также существует множество версий для домашних пользователей («Calculate Linux», «ROSA Linux»).

Особое внимание стоит уделить двум отечественным операционным системам, которые были сделаны с нуля, не базируясь ни на одной из известных в мире ОС, — KasperskyOS от «Лаборатории Касперского» и ОСРВ «МАКС» от «АстроСофт». Обе операционные системы не рассчитаны для домашнего использования, а для работы на предприятиях, в организациях.

Рассмотрим первую из них более подробно. В силу своей архитектуры, KasperskyOS гарантирует высокий уровень информационной безопасности, главный принцип работы компании — «запрещено все, что не разрешено», то есть операционная система исключает возможность вирусной атаки ПК. К особенностям ОС можно отнести:

– ОС основана на микроядре собственного производства «Лаборатории Касперского», благодаря чему достигается безопасность архитектуры модулей;

– функциональная начинка ОС универсальна и разрабатывается для каждой организации отдельно, с нуля, тем самым ОС не содержит в себе никакого лишнего функционала;

– домены безопасности имеют разграничения, тем самым ОС безопасно от внешних вирусных вторжений;

– наличие меток и идентификаторов системы дают возможность дополнительной защиты данных ПК.

Политика безопасности данной ОС — это запреты на выполнение определенных процессов и действий, они настраиваются в соответствии с потребностями организации, тем самым обеспечивая сохранность данных компьютерной сети компьютеров. В настоящее время ОС от «Лаборатории Касперского» внедрена в компании Kraftway в маршрутизирующий коммутатор уровня L3.

Перейдем к ОСРВ «МАКС», она является первой российской операционной системой, работающей в режиме реального времени. Особенности «МАКС»:

– ядро осуществляет планирование и взаимодействие потоков программы пользователя друг с другом;

– в ОС имеется планировщик, который осуществляет динамические манипуляции с задачами, позволяет работать ПК в режиме многозадачности;

– объекты синхронизации выполнения процессов (мьютексы, семафоры);

– аппаратные средства защиты памяти;

– отработанный механизм прерывания задач и процессов, построение оптимальной последовательности выполнения задач;

– автоматическое разбиение процессов на секции для ускорения работы процессора.

Многим рядовым пользователям ПК в России кажется, что IT-технологии в стране стоят на месте, но это не так, в настоящее время российские операционные системы используются, однако только в отдельных сферах, таких как медицина, военные, силовые и государственные структуры. Привыкшие к Windows и Mac OS, юзеры уже не хотят пользоваться более сложными в плане интерфейса русскими ОС.

Однако в 2025–2030 гг. правительство планирует запустить отечественную операционную систему, заявлено, что будут созданы версии этой ОС как для персональных компьютеров, так и для мобильных устройств.

«Chrome OS» и «Mageia»

Идея о том, что ОС, прежде всего, система, обеспечивающая удобный интерфейс пользователям, соответствует рассмотрению ее сверху вниз. Взгляд снизу вверх дает представление об ОС как о некотором механизме, распределяющим и управляющим всеми компонентами и ресурсами вычислительных машин и вычислительных систем с целью обеспечения максимальной эффективности их функционирования. С 1990-х гг. наиболее распространенными ОС для персональных компьютеров и серверов являются: ОС семейства Microsoft Windows и Windows NT; ОС семейства Mac OS и Mac OS X; системы класса UNIX и Unix-подобные (особенно GNU/Linux).

Операционная система «Chrome OS» — это облачно-ориентированная ОС, построенная на Linux-ядре и web-брузере Google Chrome с открытым исходным кодом от компании Google Inc. ОС предназначена для использования на нетбуках и устройствах с процессорами ARM или x86.2. Проект был анонсирован в 2009 г. В декабре 2010 г. Google запустил пробную версию, а первые устройства под управлением Chrome OS появились в 2011 г. В основе системы лежит идея «облачных вычислений» — вычисления, которые используют ресурсы сети Интернет. Проще говоря, приложения и информация находятся на удаленном сервере, а устройство получает к ним доступ через сеть Интернет. Сама ОС имеет хорошее быстродействие и занимает мало места на локальном хранилище. Chrome OS поддерживает многопользовательский режим. В Chrome OS так же как в других популярных ОС предусмотрен рабочий стол, однако разработчики запретили хранить здесь файлы или ярлыки, он предназначен лишь для работы с окнами. Chrome OS оснащен простым и удобным приложением, реализующим графический интерфейс доступа к файлам. Предусмотрен только стандартный набор функций, которые будут достаточны для работы обычных пользователей.

В Chrome OS приложения не строятся по стандартной схеме «исполняемые файлы + динамические библиотеки + драйверы». Теперь приложения — это просто веб-страницы. Приложения можно найти в предназначенном для этого веб-магазине Chrome Web Store. Приложения для Chrome OS можно установить исключительно из официального магазина приложений, запрет на установку сторонних программ значительно повышает безопасность всей системы. Пользователь может использовать заранее скачанные программы и без подключения к Интернету, работать с документами с помощью встроенного Google Docs. Также есть поддержка ряда приложений для Android OS.

Из облачных возможностей следует отметить возможность синхронизации между несколькими устройствами, облачное хранение файлов, виртуальный принтер. Внешний вид ОС выглядит точно так же, как браузер Google Chrome. Безопасность Chrome OS значительно выше, чем у других известных ОС по ряду причин. Одной из причин является отсутствие вредоносных программ для ядра Linux, на котором основана Chrome OS.

Chrome OS может работать на компьютерах со стандартным BIOS, однако с целью увеличения оптимизации нетбуки «Chromebook» базируются на

CoreBoot. CoreBoot — полностью 32-битный «BIOS», лишенный большого количества кода инициализации оборудования, бесполезного в наши дни. Совместно с оптимизациями Google CoreBoot способен выполнить старт до загрузки ядра от нажатия кнопки питания буквально за доли секунды. Далее CoreBoot находит загрузочный раздел GPT и загружает в память бинарный файл, содержащий загрузчик u-boot и ядро Linux, после чего отдает управление u-boot, и начинается стандартная для Linux-дистрибутивов процедура загрузки, включающая в себя монтирование корневого раздела, запуск служб, графической системы и интерфейса. На каждом этапе передачи управления от одного компонента к другому, например, от CoreBoot к загрузчику u-boot, происходит сверка цифровой подписи, при несовпадении которой система сделает загрузку системы с резервной копии к прошлой версии. На рисунке 39 представлена схема загрузки Chrome OS.

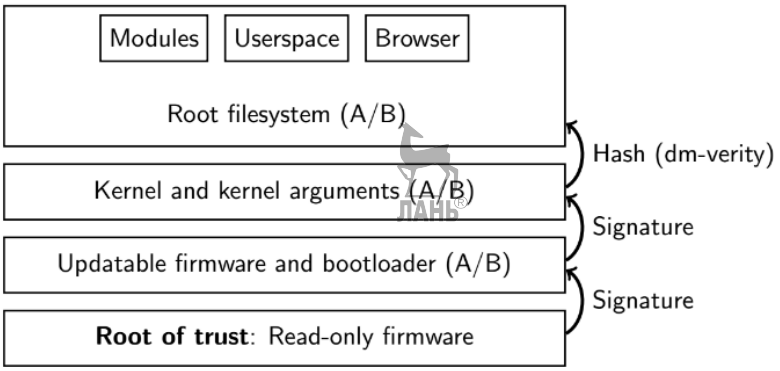


Рис. 39
Схема загрузки Chrome OS

Кроме CoreBoot, ПЗУ нетбука «Chromebook» включает в себя SeaBIOS — открытую реализацию BIOS, которая позволяет без лишних хлопот установить на устройство Windows или Linux.

Текущие версии Chrome OS основаны на Gentoo Linux. По сравнению с обычным дистрибутивом Linux система сильно урезана, поэтому она стартует буквально за секунду. От Linux ОС унаследовала возможность запуска терминала, который запускается прямо в браузере. Система состоит из известных служб rsyslogd, dbus-daemon (D-Bus используется в Chrome OS для обмена данными между браузером и остальными частями системы), wpa_supplicant (аутентификация в Wi-Fi-сетях), dhcpcd (реализация клиента DHCP и DHCPv6), иксы, ModemManager (работа с 3G-модемами), udev (менеджер устройств), ConnMan (управляет соединениями с сетью) и специальных для Chrome OS служб, отвечающих в том числе за обновление системы (update_engine), работу с TPM-модулем (chapsd), шифрование домашнего каталога (cryptohomed), отладку (debugd) и др. Особое место занимает служба session_manager, ответственная за инициализацию высокоуровневой части ОС. В ее задачи входит:

- Запуск X-сервера.
- Инициализация переменных окружения для браузера Chrome.

• Создание необходимых каталогов, файлов и правил cgroups для Chrome.
• Запуск Chrome.

На рисунке 40 изображена схема строения Chrome OS.

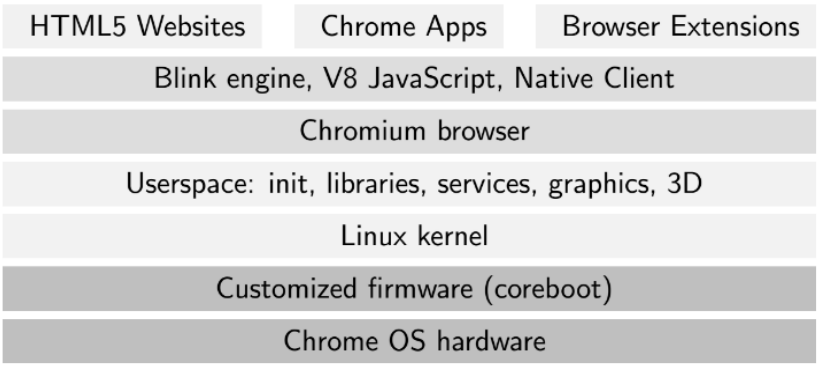


Рис. 40
Строение Chrome OS

Во время этого процесса не запускаются какие-либо компоненты, отвечающие за формирование рабочего стола. Его отрисовкой занимается сам браузер, используя фреймворк Aura, включающий в себя функции для работы с графикой и окнами и окружение рабочего стола Ash, которое отрисовывает панель задач и др.

Помимо различных методов обеспечения безопасности и целостности данных, таких как безопасная загрузка системы, зашифрованный домашний каталог с кешированными данными, а также стандартных для браузера Google Chrome методов изоляции процессов, плагинов и Native Client от системы (здесь используется механизм seccomp-bpf, позволяющий фильтровать обращения к системным вызовам), в Chrome OS задействован ряд других подходов к обеспечению безопасности.

Центральное место в безопасности системы занимает minijail — небольшое приложение, применяемое для изоляции системных служб и других компонентов системы. Это приложение позволяет выполнять такие функции, как наделение приложения «возможностями» или их отзыв (capabilities — специальная подсистема ядра Linux для наделения не SUID-бинарников некоторыми возможностями root), запереть его в chroot, отозвать права root, установить лимиты на ресурсы, разместить процесс в выделенных пространствах имен и применить к нему правила cgroups. Если изучить результат работы команды «ps aux|grep minijail» в работающей системе, то можно увидеть, что minijail используется для запуска служб с теми или иными настройками, но число таких служб по отношению ко всем работающим в системе не велико.

Из остальных средств обеспечения безопасности можно отметить использование флагов компилятора (-fno-delete-null-pointer-checks, -fstack-protector, FORTIFY_SOURCE) для минимизации риска срыва стека, задействование усиленного механизма ASLR (случайное выравнивание адресного пространства) в ядре Linux, ограничения на загрузку модулей ядра, использование модуля TPM

на материнских платах Chromebook'а для хранения ключей шифрования диска и пароля пользователя, запрет на запуск обычных ELF-бинарников пользователем и др.

Операционная система «Mageia» — это свободно распространяемая ОС с открытым исходным кодом, основанная на GNU\Linux. Начало разработки было положено в 2010 г. бывшими сотрудниками компании Mandriva, которые были уволены в ходе масштабных сокращений, и независимыми разработчиками. Mageia стала новым витком проекта «Mandriva OS» и имеет с ним много общего. Главной особенностью ОС является нацеленность на интересы пользователей. Ежегодно выходят крупные обновления, цели которых — осуществление интеграции большего количества приложений и программ, поддержка новых аппаратных средств, введение новых функций и устранение выявленных ошибок. ОС поддерживает как 32-разрядную, так и 64-разрядную версии архитектуры. Изначально устанавливаемый дистрибутив операционной системы содержит только свободное программное обеспечение. Проприетарное программное обеспечение может быть установлено пользователем самостоятельно. ОС не продвигает никаких платных программных продуктов, поддерживается за счет пожертвований от компаний и пользователей.

В ОС по умолчанию используется графическая среда рабочего стола — KDE, но имеется возможность выбирать между GNOME, XFCE и любыми другими доступными средами. Среда KDE удобна и многофункциональна. Пользователь может создать несколько рабочих столов, на которых могут быть размещены папки, ярлыки и виджеты. Имеется возможность отображать на рабочем столе содержимое любой папки. ОС имеет свой центр управления (аналог панели управления для ОС «Windows»). В центре управления Mageia предусмотрено 8 пунктов, каждый из которых предлагает набор инструментов для настройки ОС. Mageia позволяет пользователям взаимодействовать с другими ОС. Так, существует инструмент импорта документов или настроек с ОС «Windows». Поддерживается многопользовательский режим и родительский контроль. Установлен файловый менеджер Dolphin, который отличается удобным, приятным интерфейсом и в то же время достаточным многофункциональностью.

Начиная с версии Mageia 5, поддерживается UEFI — единый интерфейс расширяемой прошивки, который связывает операционную систему и аппаратную часть, и предназначен для замены стандартного BIOS. UEFI имеет высокое быстродействие, позволяет использовать графические и сетевые возможности до загрузки драйверов операционной системой. Использование функции Secure Boot Option при загрузки ОС повышает безопасность операционной системы, однако в то же время ограничивает пользователя в свободе действий в операционной системе.

Также в инталляторе реализован новый алгоритм автоматического разбиения дисковых разделов, добавлена возможность настройки RAID (dmraid и Intel soft raid), интегрирован загрузчик GRUB 2, добавлены средства отладки для URPMI, улучшен процесс автозагрузки драйверов, для обработки ввода за-

действован evdev, улучшена работа с разделами GPT, обеспечена возможность выбора btrfs как основной файловой системы для /boot или /.

С помощью инструмента drakrpm, можно устанавливать, удалять и обновлять пакеты с программным обеспечением. Программа является графическим интерфейсом к набору утилит URPMI. После каждого запуска проверяются списки пакетов в Интернете, полученные как непосредственно с официальных серверов, так и с неофициальных источников. Mageia осведомит о самых свежих пакетах с программным обеспечением для операционной системы. Сами пакеты разбиты на категории по их функционалу. Также можно ознакомиться со списком пакетов, доступных к установке, произвести поиск пакетов по их названиям, типу.

Подводя итоги, сравним рассматриваемые ОС между собой, используя несколько критериев оценки: надежность, удобство пользования ОС, эффективность, безопасность и мобильность.

Надежность. Создание операционных систем и программ к нему — очень трудоемкий процесс, который осуществляется в течение долгого времени большим числом разработчиков. В результате готовая ОС неизбежно будет содержать ошибки, большинство которых могут быть обнаружены и устранены только во время тестирования или непосредственного использования ОС многими пользователями. ОС должна быть готовой к возникновению ошибок, аварийных завершений работ программ, и продолжать нормальную работу без последствий после таких ситуаций.

Повышенную надежность Chrome OS обеспечивает использование мультипроцессной архитектуры. В данном случае браузер, движок рендеринга, расширения, подключаемые модули работают в отдельных процессах. Таким образом, нарушение работы одной программы никак не повлияет на работу всей ОС. Однако при отсутствии подключения к Интернету большинство возможностей ОС не будут доступны пользователю.

Установка программ в Mageia происходит централизованно, через менеджер установки пакетов RPM. В актуальной версии ОС, на время написания статьи, установлена последняя версия RPM, в которой существенно повышена устойчивость системы. Еще один способ повышения надежности в данной ОС заключается в том, что все свежие пакеты попадают в репозиторий «Cauldron» (котел), откуда в дальнейшем, после отладки и тестирования, попадают в репозиторий testing, а затем, достигнув стабилизации, попадают в стабильные репозитории, доступные для использования обычными пользователями. Дистрибутив предлагает пользователю возможности самому решать, какой репозиторий использовать, самостоятельно выбрав уровень надежности установленной ОС. По данному критерию нельзя выделить преимущество ни одной ОС.

Удобство пользования — несомненно важный критерий для выбора ОС. Работа в Chrome OS существенно не отличается от работы в браузере Google Chrome. Она будет привычна большинству пользователей. После установки Mageia пользователи увидят привычный рабочий стол, иконки, панель инструментов и т. д. То, что большинство функций и настроек доступны в графическом режиме, делает его существенно удобнее, чем другие дистрибутивы Линукс.

Chromebook — ПК, выпускаемый для работы именно под управлением Chrome OS, имеет весьма скромные технические характеристики. Но в то же время пользователи выделяют стабильность его работы, хорошую производительность. Отсюда следует, что разработчикам удалось добиться большей эффективности ОС по сравнению с другими популярными ОС. Mageia в отличие от Chrome OS — некоммерческий продукт, имеющий общественное начало. Из-за этого разработчики не имеют таких средств и инструментов для повышения эффективности. К тому же при установке глобальных обновлений без полной переустановки системы, могут возникнуть проблемы с совместимостью уже установленных сторонних пакетов, что приведет к ошибкам и снизит быстродействие ОС. По критерию эффективности Mageia уступает Chrome OS.

Разработчики обеих ОС уделяют большое внимание вопросу безопасности. Следует отметить общие характеристики для обеих ОС: технологии ядра Linux, ограничение установки программ от неофициальных источников, поддержка и регулярные обновления для ОС. В то же время использование песочницы для запуска программ в Chrome OS делает безопасность системы на уровень выше по сравнению с Mageia.

Мобильность — возможность переносить ОС на различные платформы. Недавнее внедрение в Mageia UEFI дала возможность пользователям установить данную ОС и работать под ее управлением на большинстве современных ПК. Компания Google Inc. введет активную политику по продвижению выпускаемых компанией ПК — Chromebook. Установка официальных релизов на другие аппаратные средства невозможна. Но в дальнейшем Google Inc. может объединить Chrome OS с Android, и тогда данный продукт можно будет устанавливать на миллионах устройствах, которые на данный момент находятся под управлением Android.

Chrome OS значительно опережает Mageia в производительности, одним из весомых факторов является использование в последнем мультипроцессной архитектуры, т. е. произведение вычислений с помощью облачных технологий. Безопасность Chrome OS и Mageia находится примерно на одинаковом уровне, последняя незначительно уступает. Выделяются такие плюсы, как технологии ядра Linux, использование песочницы, ограничение установки программ от неофициальных источников. Таким образом, можно сделать вывод, что в течение сравнительно короткого периода времени разработчикам операционных систем удалось создать продукты, которые могут в некой мере конкурировать с крупными монополистами на рынке. Сравнив коммерческую и некоммерческую ОС, можно прийти к выводу, что и без больших финансовых вложений разработчикам удалось создать надежные операционные системы, достойные внимания.

Windows CE

Система, которая нужна для кросс-разработки приложений, прошиваемых в ПЗУ, сверхпортативных компьютеров. Внедрение ПЗУ позволяет отказаться от цельного комплекта подсистем, которые обслуживают виртуальную память, загрузку исполняемых модулей и сборку в момент загрузки. Система предлагает графический пользовательский интерфейс с асинхронной очередью извеще-

ний, которая в свою очередь вытесняет многопоточность и базисный стек TCP/IP. В поставку системы входит — среда кросс-, работающая под Windows NT. Интерфейс системных вызовов данной ОС похож на Win32 API, но основным источником притязаний было не обеспечивание сопоставимости с приложениями для Win32, а просьбы создателей Mobile. Система разработана с нулевой отметки, т. е. без применения имеющегося кода Wiii32-cHcreM.

Windows NT/2000/XP

Выработки Microsoft no OS/2 New Technology были в 1993 г. представлены на рынке как Windows NT. Версии 3.x и 4.0 этой системы обеспечивали сопоставимость с 16-разрядными приложениями для OS/2 1.x в другой подсистеме, без способности обращаться из 16-разрядных приложений к 32-разрядным DLL и напротив. В этот этап из DEC в Microsoft перешла команда создателей ядра VMS под управлением Д. Катера. Microsoft обширно раскрутил данный прецедент и заявлял, что Windows NT располагается с VMS в значительно большем ближайшем родстве, чем с OS/2 1.x.

Семейство CP/M

Все началось с дисковой операционной системы CP/M (Control Program/Monitor) компании Digital Research. 1-я версия системы была разработана в 1974 г. для применения в инструментальных микропроцессорных системах на базе процессоров 18080 и 18085.

Инструментальные микрокомпьютеры применялись как средство кросс-разработки и отладки программ для встраиваемых микропроцессорных систем. Обычная система состояла из микропроцессорной платы, прибора чтения/записи магнитных или же перфолент, а позже — накопителя гибких дисков и видеотерминала.

CP/M была первая ОС для машин такого семейства, которая обеспечивала вероятность применения гибких дисков, затем она проворно купила гигантскую известность и стала стереотипом де-факто для микрокомпьютеров. Система была перенесена буквально на все 8- и 16-разрядные и почти все 32-разрядные процессоры манчестерской архитектуры. Индивидуальные компьютеры, как правило, еще были нацелены на внедрение CP/M.

С архитектурной точки зрения, CP/M представляет достаточно простую однозадачную ДОС, которая предназначена для работы на микропроцессоре без диспетчера памяти и средств базисной адресации.

Linux

В 1991 г. Л. Торвальдс начал разработку Linux — настоящей операционной системы, которая основывалась на начальных кодах Minix и распространялась на критериях GPL.

В 1992 г. была выпущена 1-я версия системы. И тут же огромное количество фирм (такие как RedHat, Caldera, SuSe и другие) начало распространение коммерчески поддерживаемых дистрибутивов ОС на базе ядра Linux.

И наконец, вышедшее в 1997 г. ядро Linux 2.0 имело абсолютно приемлемую по эталонам платных ОС надежность и практически все более прогрессивные черты иных.

Unix System Release 4

Ожидаемая в 1987 г. UNIX System VI появилась на рынке в 1989 г. под заглавием UNIX SVR4. Микроядерная система обеспечивала абсолютную бинарную сопоставимость с SVR3, и просто бинарную сопоставимость с 16- и 32-разрядными Xenix на микропроцессоре x86, и к тому же сопоставимость на уровне начальных слов с BSD Unix v4.3. Заявленная задача консолидации всех ведущих веток Unix в единственной системе была достигнута.

Операционная система Mach

Операционная система Mach включает много инноваций в исследованиях операционной системы и представляет собой полностью функциональную, технически продвинутую систему. В отличие от UNIX, который был разработан без учета многопроцессорности, Mach включает в себя многопроцессорную поддержку повсюду. Эта поддержка чрезвычайно гибкая, приспособленная для систем с общей памятью, а также для систем без памяти. Mach предназначен для работы в компьютерных системах в диапазоне от одного процессора до тысячи процессоров. Кроме того, его легко портировать на множество компьютерных архитектур. Основная цель Mach — быть распределенной системой, способной функционировать на разнородном оборудовании.

Хотя многие экспериментальные операционные системы разрабатываются, создаются и используются, Mach удовлетворяет потребностям большинства пользователей лучше, чем другие, так как он предлагает полную совместимость с UNIX 4.3 BSD. Эта совместимость также дает нам уникальную возможность сравнить две функционально схожие, но внутренне несходные операционные системы — Mach и UNIX.

Mach начинает свое происхождение от операционной системы Accent, разработанной в Университете Карнеги Меллона (КМУ). Хотя Accent был пионером в концепции операционной системы, его полезность была ограничена из-за невозможности выполнения приложений UNIX и его прочной связи с единой аппаратной архитектурой, которая затрудняла портирование. Коммуникационная система и философия Mach выведена из Accent, но многие другие важные части системы (например, система виртуальной памяти и управление задачами и потоками) были разработаны с нуля. Важной целью Mach была поддержка мультипроцессоров.

Mach развивалась от систем BSD UNIX. Код Mach изначально разрабатывался внутри ядра 4.2BSD, с BSD компоненты ядра заменены компонентами Mach. В версии 2 Mach обеспечил совместимость с соответствующими системами BSD путем включения большей части кода BSD в ядро. Новые функции и возможности Mach сделали ядра в этих версиях больше, чем соответствующие ядра BSD. В Mach 3 (рис. 41) код BSD вывели из ядра, оставляя гораздо меньшее микроядро. Эта система реализует только основные функции Mach в ядре;

весь код, специфичный для UNIX, был исключен для работы на серверах пользовательского режима. Исключение UNIX-специфичного кода из ядра позволяет заменять BSD другой операционной системой или одновременным выполнением нескольких интерфейсов операционной системы поверх микроядра. В дополнение к BSD для DOS были разработаны реализации пользовательского режима, операционные системы Macintosh и OSF/1. Этот подход имеет сходство с концепцией виртуальной машины, но виртуальная машина определяется программным обеспечением (интерфейс ядра Mach), а не аппаратно. С выходом 3.0 Mach стали доступны на самых разных системах, включая однопроцессорные Sun, Intel, IBM и DEC, а также многопроцессорные DEC, Sequent и Encore системы.

Mach был выдвинут на передний план внимания отрасли, когда OpenSoftwareFoundation (OSF) объявила в 1989 г., что будет использовать Mach 2.5 в качестве основы для его новой операционной системы OSF/1. Релиз OSF/1 через год произошел, и теперь она конкурирует с UNIX System V, версии 4, операционной системой среди членов UNIX International (UI). Члены OSF включают ключевые технологические компании, такие как IBM, DEC и HP. Mach 2.5 также является основой для операционной системы на рабочей станции NeXT, детище Стива Джобса. OSF оценивает Mach 3 как основу для будущего выпуска операционной системы, и исследования по Mach продолжаются в CMU, OSF и в других местах.

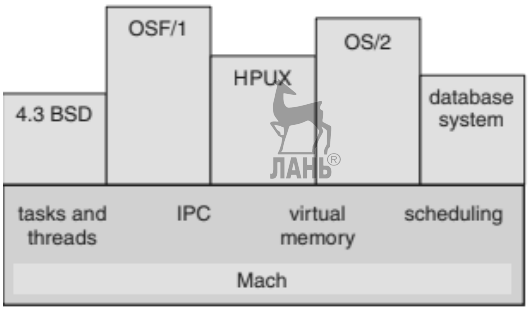


Рис. 41
Структура Mach 3

Принципы архитектуры

Операционная система Mach была разработана, чтобы обеспечить основные механизмы, которых большинству современных операционных систем не хватает. Цель состоит в том, чтобы разработать операционную систему, совместимую с BSD и, кроме того, превосходящую в следующих областях:

- Поддержка различных архитектур, в том числе многопроцессорных с различной степенью доступа к общей памяти: равномерным доступом к памяти (UMA), неоднородным доступом к памяти (NUMA) и отсутствием удаленного доступа к памяти (NORMA).
- Возможность работы с различными скоростями межкомпьютерной сети, от глобальных сетей к высокоскоростным локальным сетям и тесно связанных многопроцессорными системами.

• Упрощенная структура ядра с небольшим количеством абстракций (в свою очередь, эти абстракции являются достаточно общими, что позволяет другим операционным системам быть реализованными поверх Mach).

• Распределение операций, предоставляющих прозрачность сети для клиентов и объектно-ориентированную организацию как внутри, так и снаружи.

• Интегрированное управление памятью и межпроцессное взаимодействие, обеспечивающее эффективную передачу большого количества данных, а также управление памятью на основе связи.

• Разнородная системная поддержка, позволяющая сделать очень широкодоступные и взаимодействующие между компьютерными системами от нескольких поставщиков.

На дизайнеров Mach сильно повлияли BSD (и UNIX, в общем), чьи преимущества включают:

• Простой интерфейс для программиста, с хорошим набором примитивов и согласованным набором интерфейсов к системным средствам.

• Легкая переносимость на широком классе отдельных процессоров.

• Обширная библиотека утилит и приложений.

Конечно, дизайнеры также хотели исправить те недостатки, которые они видели в BSD:

• Ядро, которое стало хранилищем многих избыточных функций, и, следовательно, им трудно управлять и изменять.

• Исходные цели разработки, которые затруднили поддержку мультипроцессоров, распределенными системами и общими библиотеками программ (потому что ядро было разработано для отдельных процессоров, оно не имеет возможности для блокировки кода или данных, которые могут использовать другие процессоры).

• Слишком много фундаментальных абстракций, предоставляющих слишком много похожих, конкурирующих средств для выполнения тех же задач.

Системные компоненты

Чтобы достичь целей проектирования Mach, разработчики сократили функциональность операционной системы до небольшого набора основных абстракций, из которых все остальные функциональные возможности могут быть выведены. Подход Mach — разместить как можно меньше внутри ядра, но чтобы сделать так, чтобы он был достаточно мощным, чтобы все остальные функции могли быть реализованы на уровне пользователя.

Философия **архитектуры** Mach состоит в том, чтобы иметь простое, расширяемое ядро, концентрирующееся на объектах связи. Например, все запросы к ядру и все перемещения данных между процессами обрабатываются через единый механизм сообщений. Таким образом, Mach способен обеспечить защиту всей системы своим пользователям, защищая механизм связи. Оптимизация этого единого канала связи может привести к значительному повышению производительности, и это проще, чем пытаться оптимизировать несколько путей. Mach расширяемый, потому что многие традиционные функции ядра могут

быть реализованы как серверы пользовательского уровня. Например, все пейджеры (включая пейджер по умолчанию) могут быть реализованы внешне и вызываются ядром для пользователя.

Mach является примером объектно-ориентированной системы, в которой данные и операции, которые манипулируют этими данными, инкапсулированы в абстрактный объект. В нем только операции объекта могут воздействовать на определенные объекты. Детали реализации этих операций скрыты, так как являются внутренними структурами данных. Таким образом, программист может использовать только объект, вызывая его определенные, экспортированные операции. Программист может изменять внутренние операции без изменения интерфейса, поэтому изменения и оптимизации не влияют на другие аспекты работы системы. Отклоненный подход, поддерживаемый Mach, позволяет объектам находиться где угодно в сети системы Mach.

Примитивные абстракции Mach являются сердцем системы и заключаются в следующем:

- Задача — это среда выполнения, которая обеспечивает базовую единицу ресурса — распределение. Она состоит из виртуального адресного пространства и защищенного доступа к системным ресурсам через порты и может содержать один или несколько потоков.
- Поток является основной единицей выполнения и должен выполняться в контексте задачи (которая обеспечивает адресное пространство). Все потоки в рамках задачи совместно используют ресурсы задачи (порты, память и т. д.). В Mach отсутствует понятие процесс. Скорее традиционный процесс реализован как задача с одной нитью контроля.
- Порт является основным механизмом ссылки на объект в Mach и реализован как защищенный ядром канал связи. Связь осуществляется путем отправки сообщений в порты; сообщения помещаются в очередь на порт назначения, если ни один поток не готов немедленно получить их. Порты защищены возможностями, управляемыми ядром, или правами порта. Задача должна иметь право на порт для отправки сообщения в порт. Программист вызывает операции над объектом путем отправки сообщения в порт, связанный с этим объектом. Объект, представленный портом, получает сообщения.
- Набор портов — это группа портов, совместно использующих общую очередь сообщений. Поток может получать сообщения для набора портов и, таким образом, обслуживать несколько портов. Каждое полученное сообщение идентифицирует отдельный порт (в пределах набора), с которого было получено. Объект, представленный портом, получает сообщения.
- Сообщение является основным методом связи между потоками в Mach. Это типизированная коллекция объектов данных. Для каждого объекта он может содержать фактические данные или указатель на строку данных. Права порта передаются в сообщения; это единственный способ перемещать их среди задач. (Проход через порт прямо в разделяемой памяти не работает, потому что ядро Mach не будет разрешать новому заданию использовать права, полученные таким образом.)

• Объектом памяти является источник памяти. Задачи могут получить к нему доступ путем сопоставления части объекта (или весь объект) в свои адресные пространства. Объект может управляться менеджером внешней памяти в пользовательском режиме. Один пример — файл, управляемый файловым сервером; однако объект памяти может быть любым объектом, для которого имеет смысл отображаемый в памяти доступ. Сопоставленная буферная реализация канала UNIX является еще одним примером.

Рисунок 42 иллюстрирует эти абстракции.

Необычной особенностью Mach и ключом к эффективности системы является смешение функций памяти и межпроцессной связи (IPC). В то время как некоторые другие распределенные системы (например, Solaris с его функциями NFS) имеют специальные расширения файловой системы, чтобы расширить ее по сети, Mach обеспечивает универсальное расширяемое объединение памяти и сообщений на сердце его ядра. Эта функция не только позволяет использовать Mach для распределения и параллельного программирования, но также помогает в реализации самого ядра.

Mach связывает управление памятью и IPC, позволяя каждому быть используемым при реализации другого. Управление памятью основано на использовании объектов памяти. Объект памяти представлен портом (или портами), и сообщения IPC отправляются на этот порт для запроса операций (например, pagein, pageout) на объекте. Поскольку используется IPC, память объектов может находиться в удаленных системах и доступна прозрачно. Ядро кэширует содержимое объектов памяти в локальной памяти. И наоборот, управление памятью методами используется при реализации передачи сообщений.

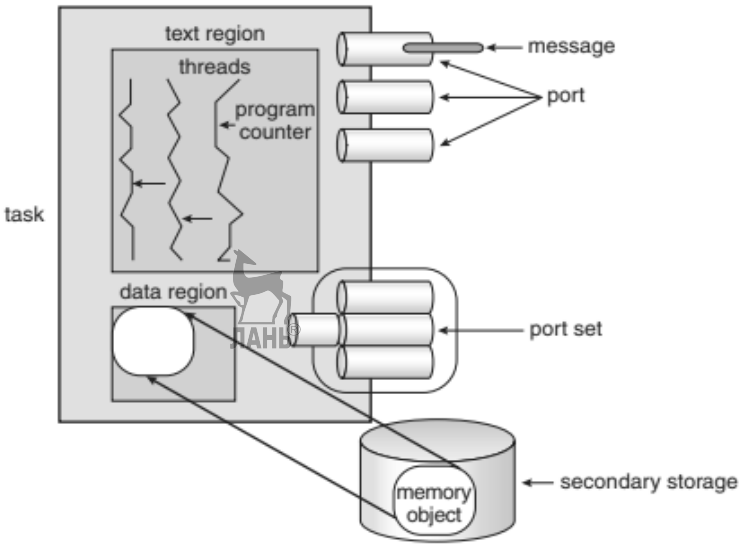


Рис. 42
Основные абстракции Mach

Где это возможно, Mach передает сообщения, перемещая указатели в общую память объектов, а не путем копирования самих объектов.

IPC, как правило, предполагает значительную нагрузку на систему. Для внутрисистемных сообщений он менее эффективен, чем связь через общую память. Поскольку Mach — это ядро на основе сообщений, обработка сообщений должна быть выполнена эффективно. Большая часть неэффективной обработки сообщений в традиционных операционных системах связано либо с копированием сообщений из одной задачи к другой (для внутрикомпьютерных сообщений), либо с низкой скоростью сети-передачи (для межкомпьютерных сообщений). Для решения этих проблем Mach использует виртуальное переназначение памяти для передачи содержимого больших сообщений. Другими словами, передача сообщения изменяет адресное пространство принимающей задачи, чтобы включить копию содержимого сообщения. Методы виртуального копирования (или копирования при записи) используются, чтобы избежать или задержать фактическое копирование данных. Этот подход имеет несколько преимуществ.

- Повышенная гибкость в управлении памятью для пользовательских программ.
- Большая общность, позволяющая использовать подход виртуального копирования в слабосвязанных компьютерах.
- Улучшена производительность по сравнению с передачей сообщений в UNIX.
- Более простая миграция задач (поскольку порты не зависят от местоположения, задача и все его порты могут быть перемещены с одной машины на другую. Все задачи, которые ранее общались с перемещенной задачей, можно продолжать делать, потому что они ссылаются на задачу только по ее портам и общаются через сообщения на эти порты.).

Задача может рассматриваться как традиционный процесс, который не имеет указателей инструкции или набора регистров. Задача содержит виртуальное адресное пространство, набор прав порта и учетную информацию. Задача — это пассивная сущность, которая ничего не делает, если в нем не выполняется один или несколько потоков.

Задача, содержащая один поток, похожа на процесс UNIX. Например, как fork(). Системный вызов создает новый процесс UNIX, Mach создает новую задачу с помощью fork(). Память новой задачи является дубликатом адресного пространства родителя, как продиктовано атрибутами наследования памяти родителя. Новая задача содержит один поток, который запускается в той же точке, где fork() обращается к родителю. Потоки и задачи также могут быть приостановлены и возобновлены.

Потоки особенно полезны в серверных приложениях, которые являются общими в UNIX, поскольку одна задача может иметь несколько потоков для обслуживания нескольких запросов к задаче. Потоки также позволяют эффективно использовать ресурсы параллельных вычислений. Вместо того, чтобы иметь один процесс на каждом процессоре, с соответствующим снижением производительности и издержек операционной системы, задача может иметь потоки, распределяющиеся между параллельными процессорами. Потоки также позволяют эффективно использовать параллельные вычислительные ресурсы. Например, в UNIX весь процесс должен ожидать, когда происходит сбой стра-

ницы или при выполнении системного вызова. В задаче с несколькими потоками, только тот поток, который вызывает ошибку страницы или выполняет системный вызов, задерживается, а все остальные потоки продолжают выполняться.

На уровне пользователя потоки могут находиться в одном из двух состояний:

• Запущенный. Поток либо выполняет, либо ожидает, что ему будет выделено процессором. Поток считается работающим, даже если он заблокирован внутри ядра (например, ожидание сбоя страницы).

• Приостановленный. Поток не выполняется на процессоре и не ожидает выделения процессором. Поток может возобновить свое выполнение, только если он вернется в рабочее состояние.

Эти два состояния также связаны с задачей. Операция над задачей влияет на все потоки в задаче, поэтому приостановка задачи включает в себя приостановку всех потоков в нем.

Mach предоставляет примитивы, из которых могут быть получены инструменты синхронизации встроенных потоков. Эта практика согласуется с философией Mach о предоставлении минимальной, но достаточной функциональности в ядре. Объект Mach IPC может быть использован для синхронизации с процессами, обменивающимися сообщениями в точках встречи. Синхронизация на уровне потоков обеспечивается вызовами для запуска и остановки потоков в соответствующее время. Количество приостановок сохраняется для каждого потока. Этот счетчик позволяет выполнять несколько вызовов suspend () в потоке, и только при равном количестве вызовов resume () поток возобновляется. К сожалению, эта функция имеет свои ограничения. Потому что это ошибка для вызова start() выполняется до вызова stop() (счетчик приостановок станет отрицательным), эти процедуры не могут быть использованы для синхронизации доступа к общим данным. Тем не менее, операции wait () и signal(), связанные с семафорами и используемые для синхронизации, могут быть реализованы через вызовы IPC.

Пакет потоков языка C

Mach предоставляет низкоуровневые, но более гибкие процедуры-шаблоны вместо элегантных, больших и ограниченных функций. Но вместо того, чтобы заставлять программистов работать на таком низком уровне, Mach предоставляет множество высокоуровневых интерфейсов для программирования на C и других языках. Например, пакет потоков C обеспечивает несколько потоков управления, общие переменные, взаимное исключение для критических секций и условные переменные для синхронизации. На самом деле, потоки C оказывают одно из основных влияний на стандарт POSIX Pthreads, которые поддерживают многие операционные системы. В результате между потоками C и Pthreads образуются программные интерфейсы. Процедуры контроля потока включают в себя вызовы для выполнения следующих задач.

• Создание нового потока в рамках задачи, учитывая функцию для выполнения и параметры в качестве ввода. Затем поток выполняется одновремен-

но с созданием нового потока, который получает идентификатор потока при возврате вызова.

• Уничтожение вызываемого потока и возвращение значения в созданный поток.

• Дождитесь завершения определенного потока, прежде чем разрешить вызванному потоку продолжать. Этот вызов является инструментом синхронизации, очень похожим на системные вызовы UNIX wait ().

• Использовать процессор, сигнализируя о том, что планировщик может запустить другой поток в этой точке. Тот вызов также полезен при наличии упреждающего планировщика, так как он может быть использован для добровольного освобождения ЦП, если поток не использует процессор.

Подпрограммы, связанные с взаимным исключением:

• Подпрограмма mutexalloc () динамически создает переменную mutex.

• Подпрограмма mutexfree () освобождает динамически создаваемую переменную mutex.

• Процедура блокировки mutex () блокирует переменную mutex. Исполняющий поток находится в цикле в спин-блокировке, пока блокировка не будет достигнута. В результате возникает безвыходное положение, если поток с блокировкой пытается заблокировать ту же переменную mutex. Ограниченное ожидание не гарантируется пакетом потоков C. Скорее, это зависит от аппаратных указаний, используемых для реализации процедур mutex. Подпрограмма mutexunlock () разблокирует переменную mutex.

Общая синхронизация без ожидания может быть достигнута благодаря использованию условных переменных, которые могут быть использованы для реализации условий критической области, как описано на стр 128. Условная переменная связана с переменной mutex и отражает логическое состояние этой переменной.

Процедуры, связанные с общей синхронизацией, таковы.

• Подпрограмма condition_alloc () динамически распределяет переменную условия.

• Подпрограмма condition_free () удаляет динамически созданную условную переменную, выделенную в результате выполнения условия alloc ().

• Подпрограмма condition_wait () разблокирует связанную переменную mutex и блокирует поток до тех пор, пока condition_signal() не будет выполнен на переменной условия, указывающей, что ожидаемое событие могло произойти.

Переменная mutex затем блокируется, и поток продолжается.

Планировщик ЦП

Планировщик ЦП для многопроцессорной операционной системы на основе потоков является более сложным, чем его родственные процессы.

В многопоточной системе намного больше тем, чем процессов в многозадачной системе. Отслеживание нескольких процессоров также сложно и является относительно новой областью исследования. Mach использует простую политику, чтобы сохранить планировщик управляемым. Только потоки запла-

нированы, поэтому знание планировщика не требуется. Все потоки в равной степени конкурируют за ресурсы, включая кванты времени.

Каждый поток имеет связанный номер приоритета в диапазоне от 0 до 127, который основан на экспоненциальном среднем использовании процессора. Тот поток, который недавно использовал процессор в течение большого количества времени, имеет самый низкий приоритет. Mach использует приоритет для размещения потока в одном из 32 глобальных прогонов очереди. Эти очереди ищутся в порядке приоритета для ожидающих потоков, когда процессор становится бездействующим. Mach также поддерживает очереди для каждого процессора или локальные очереди. Локальная очередь выполнения используется для потоков, которые связаны с отдельным процессором.

Например, драйвер устройства для устройства, подключенного к отдельному ЦП, должен работать только на этом процессоре.

```
do {
mutex lock(mutex);
while(empty)
condition wait(nonempty, mutex);
...
// remove an item from the buffe to nextc
...
condition signal(nonfull);
mutex unlock(mutex);
...
// consume the item in nextc
...
} until(FALSE);
```

Вместо центрального диспетчера, который назначает потоки процессорам, каждый процессор консультирует локальные и глобальные очереди выполнения, чтобы выбрать подходящий поток для запуска. Потоки в локальной очереди выполнения имеют абсолютный приоритет над теми, что в глобальных очередях, потому что предполагается, что они выполняют какие-то функции в ядре (inkernel). Очереди выполнения — как и большинство других объектов в Mach — заблокированы, пока они изменяются, чтобы избежать одновременных изменений несколькими процессами. Для ускорения диспетчеризации потоков в глобальной очереди выполнения Mach ведет список неработающих процессоров.

Дополнительные трудности планирования возникают из-за многопроцессорной природы Mach'а. Фиксированный квант времени не подходит, потому что, например, может быть меньше выполняемых потоков, чем доступно процессоров. Было бы расточительно прерывать поток с переключением контекста на ядро, когда квант этого потока заканчивается, только чтобы поток был помещен обратно в рабочее состояние. Таким образом, вместо использования кванта фиксированной длины, Mach меняет размер кванта времени обратно пропорционально общему количеству потоков в системе. Однако он сохраняет временной интервал по всей системе постоянным. Например, в системе с 10 процессорами, 11 потоками и 100 миллисекундами квантования, переключение

контекста должно происходить на каждом процессоре только один раз в секунду для поддержания желаемого кванта.

Конечно, сложности все еще существуют. Даже отказ от работы процессора во время ожидания ресурсов для работы сложнее, чем при работе традиционных операционных систем. Во-первых, поток должен выполнить вызов, чтобы предупредить планировщик о том, что поток вот-вот перекроют. Это предупреждение позволяет избежать условий скоростных перегрузок и тупиков, которые могут возникать, когда работа происходит в многопроцессорной среде. Второй вызов фактически приводит к перемещению потока из очереди выполнения до тех пор, пока происходит соответствующее событие. Планировщик использует много других состояний внутреннего потока для выполнения управления потока.

Обработка исключений

Mach был разработан для обеспечения единой, простой, последовательной системы обработки исключений, с поддержкой как стандартных, так и пользовательских исключений. Для того чтобы избегать избыточность в ядре, Mach использует примитивы ядра, когда это возможно. Например, обработчик исключений — это просто еще один поток в задаче, в которой возникает исключение. Сообщения удаленного вызова процедур (RPC) используются для синхронизации выполнения потока, вызывающего исключение, и для передачи информации об исключении. Исключения Mach также используются для эмуляции сигнала BSD-пакета.

Нарушения нормального выполнения программы бывают двух видов: внутренние генерируемые исключения и внешние прерывания. Прерывания — асинхронно созданные срывы потоков или задачи, в то время как исключения вызваны возникновением необычных условий во время выполнения потока. Универсальное средство исключения Mach используется для обнаружения ошибок и поддержки отладчика.

Это средство также полезно для других функций, таких как сброс ядра плохой задачи, позволяющих задачам обрабатывать собственные ошибки (в основном арифметические), а также эмулировать инструкции, не реализованные аппаратно. Mach поддерживает две разные детализации обработки исключений. Обработка ошибки поддерживается обработкой исключений для каждого потока, тогда как отладчики используют обработку каждой задачи. Нет смысла пытаться отлаживать только один поток или иметь исключения из нескольких потоков, вызывающих несколько отладчиков.

Помимо этого, единственное различие между двумя типами исключений заключается в их наследовании от родительской задачи. Общие для всех задач средства обработки исключений передаются от родительских к дочерним задачам, поэтому отладчики могут управлять целым деревом задач. Обработчики ошибок не наследуются и по умолчанию не обрабатываются во время создания потока и задачи. Наконец, обработчики ошибок имеют приоритет над отладчиками, если исключения происходят одновременно. Причина такого подхода за-

ключается в том, что обработчики ошибок обычно являются частью задачи и поэтому должны выполняться даже в присутствии отладчика.

Обработка исключений осуществляется следующим образом.

- Поток-жертва вызывает уведомление о возникновении исключения через сообщение raise () RPC, отправленное обработчику.
- Затем жертва вызывает процедуру, чтобы дождаться обработки исключения.
- Обработчик получает уведомление об исключении, обычно включающее сведения об исключении, потоке и задаче, вызывающей исключение.
- Обработчик выполняет свою функцию в соответствии с типом исключения. Действие обработчика включает в себя очистку исключения, в результате чего жертва возобновляет или прерывает поток.

Чтобы поддержать выполнение программ BSD, многое должно поддерживать сигналы стиля BSD. Сигналы обеспечивают программные прерывания и исключения. К сожалению, сигналы имеют ограниченную функциональность в многопоточной работе системы. Первая проблема заключается в том, что в UNIX обработчик сигнала должен быть шаблонным в процессе получения сигнала. Если сигнал вызван проблемой в самом процессе (например, деление на 0), то проблема не может быть исправлена, потому что процесс имеет ограниченный доступ к своему собственному контексту. Вторая проблема заключается в том, что более хлопотным аспектом сигналов является то, что они были разработаны только для однопоточных программ. Например, для всех потоков в задаче нет смысла в получении сигнала, но как сигнал может быть виден только одним потоком?

Поскольку сигнальная система должна правильно работать с многопоточными приложениями для Mach для запуска программ 4.3 BSD, сигналы не могут быть проигнорированы. Для создания функционально корректного пакета сигналов потребовалось несколько перезаписей кода. Последняя проблема с сигналами UNIX заключается в том, что они могут потеряться. Эта потеря происходит, когда другой сигнал того же типа попадается до того, как первый еще не обработался. Исключения Mach помещаются в очередь в результате их реализации RPC. Внешне генерируемые сигналы, в том числе и отправленные из одного BSD процесса другим, обрабатываются разделом сервера BSD ядра Mach 2.5. Их поведение такое же, как и в BSD. Другое дело — аппаратные исключения, потому что программы BSD принимают получение аппаратных исключений в качестве сигналов. Поэтому аппаратное исключение, вызванное потоком, должно прибыть в поток как сигнал.

Чтобы этот результат был получен, аппаратные исключения преобразованы в исключения RPC. Для задач и потоков, которые не используются средствами обработки исключений Mach, назначение этого RPC по умолчанию лежит в задаче «ядра». Эта задача имеет только одну цель: ее поток выполняется в непрерывном цикле, получая исключения RPC. Для каждого RPC, она преобразует исключение в соответствующий сигнал, который отправляется потоку, вызвавшему аппаратное исключение. Затем он завершает RPC, очищая исходное условие исключения. С завершением RPC инициирующий поток снова входит в

рабочее состояние. Он немедленно выполняет обнаружение сигнала и выполняет его обработку. Однако потоки, не предназначенные для обработки таких исключений, получают исключения, как и в стандартной системе BSD, — как сигналы. В Mach 3.0 код обработки сигналов перемещается полностью на сервер, но в целом структура и подача управления подобны Mach 2.5.

Большинство коммерческих операционных систем, таких как UNIX, обеспечивают связь между процессами и между хостами с фиксированными глобальными именами (или интернет-адресами). Нет зависимости от местоположения объектов, потому что любая удаленная система, нуждающаяся в использовании объекта, должна знать имя системы, предоставляющей этот объект. Обычно данные в сообщениях представляют собой нетипизированные потоки байтов. Mach упрощает эту картину, отправляя сообщения между независимыми от местоположения портами. Сообщения содержат типизированные данные для удобства интерпретации. Все BSD-методы связи могут быть реализованы с помощью этой упрощенной системы. Двумя компонентами Mach IPC являются порты и сообщения. Почти все в Mach является объектом, и все объекты адресуются через их коммуникационные порты. Сообщения отправляются в эти порты для инициирования операций над объектами по подпрограммам, реализующим объекты. Mах обеспечивает независимость расположения объектов и безопасность связи. Независимость данных обеспечивается задачей NetMsgServer. Mach обеспечивает безопасность, требуя, чтобы отправители и получатели сообщений имели права. Право состоит из имени порта и возможности отправить или получить что-либо на этот порт. Только одна задача может иметь права на получение любого заданного порта, но многие задачи могут отправлять права. При создании объекта его создатель также выделяет порт для представления объекта и получает права доступа к этому порту. Права могут быть предоставлены создателем объекта, включая ядро, и передаются в сообщениях. Если обладатель права на получение отправляет это право в сообщении, то получатель сообщения получает это право, а отправитель его теряет.

Порты

Порт реализуется как защищенная, ограниченная очередь в ядре системы, в которой находится объект. Если очередь заполнена, отправитель может отменить отправление или дождаться, пока слот станет доступным в очереди.

Несколько системных вызовов предоставляют порту следующие функциональные возможности.

- Выделить новый порт в указанной задаче и предоставить задаче вызывающего абонента доступ к правам на новый порт.
- Отменить права доступа задачи к порту. Если задача удерживает право на получение, то порт уничтожается, а все остальные задачи с правами на потенциальную отправку предупреждаются.
- Получить текущее состояние порта задачи.
- Создать резервный порт, которому предоставляется право получения, если задача, содержащая право получения, запрашивает его освобождение или уничтожение.

При создании задачи ядро создает для нее несколько портов. Функция taskself() возвращает имя порта, представляющего задачу в вызовах ядра. Например, чтобы выделить новый порт, задача вызывает portallocate() с помощью задачи self() в качестве имени задания, которым будет владеть порт. Эта схема аналогична стандартной концепции process-ID, найденной в UNIX. Другой порт возвращается с помощью функции tasknotify (); это порт, на который ядро будет отправлять уведомления о событиях (например, уведомления о прекращении работы порта).

Порты также могут быть собраны в комплекты. Этот механизм полезен, если поток должен обслуживать запросы, поступающие на несколько портов — например, для несколько объектов. Порт может быть членом не более одного комплекта портов, установленного в данный момент времени. Кроме того, если порт находится в наборе, он не может использоваться непосредственно для получения сообщения. Вместо этого сообщения будут перенаправляться в очередь набора портов. Порт-установщик не может передавать сообщения, в отличие от порта. Наборы портов — это те же объекты, обслуживающие цель аналогично системному вызову 4.3 BSD select (), но они более эффективны.

Сообщения

Сообщение состоит из заголовка фиксированной длины и переменной количества введенных объектов-данных. Заголовок содержит имя порта назначения, имя порта-ответчика, на который должны быть отправлены ответные сообщения, и длину сообщения. Данные в сообщении (или встроенные данные) были ограничены 8 Кб в системах Mach 2.5, но Mach 3.0 не имеет ограничений. Все данные раздела могут быть простого типа (цифры или символы), правами порта или указателями к устаревшим данным. Разделы связаны между собой так, что приемник сможет распаковать данные правильно, даже если он использует порядок байтов, отличный от того, который использует отправитель. Ядро также проверяет сообщение для определенных типов данных. Например, ядро должно обрабатывать информацию о порте в сообщении либо путем преобразования имени порта во внутренний адрес структуры данных порта, либо путем пересылки его для обработки в NetMsgServer.

Использование указателей в сообщении предоставляет возможность для передачи всего адресного пространства задачи в одном сообщении. Ядро также должно обрабатывать указатели на данные вне строки, так как указатель на данные в адресном пространстве отправителя будет недействительным у получателя, особенно если отправитель и получатель находятся в разных системах. Как правило, системы отправляют сообщения копирования данных от отправителя к получателю. Потому что этот метод может быть неэффективным, особенно для больших сообщений. Mach использует другой подход. Данные, на которые ссылается указатель в сообщении, отправляемом в порт в той же системе, не копируются между отправителем и получателем. Вместо этого адрес получающейся задачи изменяется для включения копии при записи страницы сообщения. Эта операция выполняется намного быстрее, чем копирование дан-

ных, и передача сообщений более эффективна. По сути, реализована передача сообщений через управление виртуальной памятью.

В версии 2.5 эта операция была реализована в два этапа. Указатель в области памяти заставляет ядро отображать эту область в своем собственном пространстве и временно установить карту памяти отправителя в режим копирования при записи, чтобы обеспечить неприкасаемость исходных версий данных. Когда сообщение получено в месте назначения, ядро перенесло адресное пространство получателя с использованием недавно выделенной области виртуальной памяти в рамках этой задачи.

В версии 3 этот процесс был упрощен. Ядро создает структуру данных, это была бы копия региона, если бы он был частью адресной карты. При получении эта структура данных добавляется на карту получателя и становится доступной для копирования получателю. Поэтому виртуальная память Mach называется разреженной, состоящей из областей данных, разделенных нераспределенными адресами. Полная передача сообщений показана на рисунке 43.

Чтобы сообщение было отправлено между компьютерами, его местоназначение должно быть обнаруженным, и сообщение должно быть передано в пункт назначения. UNIX традиционно оставляет эти механизмы сетевым протоколам низкого уровня, которые требуют использования статически назначенных конечных точек связи (например, номер порта для служб на основе TCP или UDP). Один из принципов Mach заключается в том, что все объекты в системе независимы от местоположения и что расположение прозрачно для пользователя. Этот принцип требует, чтобы Mach обеспечивал прозрачность имен и транспорта для передачи IPC на несколько компьютеров.

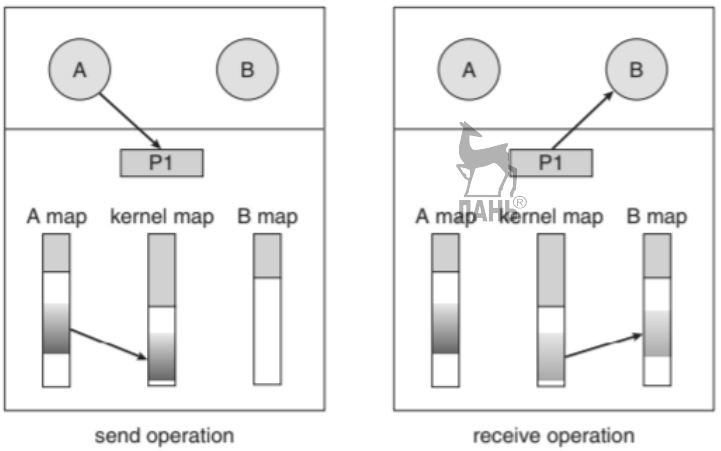


Рис. 43
Передача сообщений Mach

Это наименование и перераспределение выполняется сетевым сервером сообщения (NetMsgServer), пользовательской службой сети на основе возможностей, которые пересылают сообщения между хостами. Он также предоставляет примитивную службу широких сетевых имен, которая позволяет задачам регистрировать порты для поиска по задачам на любом другом компьютере в

сети. Netmsgserver поддерживают между собой распределенную базу данных прав портов, которые были переданы между компьютерами и портами, которым соответствуют эти права.

Ядро использует NetMsgServer, когда необходимо отправить сообщение на порт, которого нет на компьютере. Ядро Маха IPC используется для перенесения сообщений на локальный NetMsgServer. Затем NetMsgServer использует сетевые протоколы, подходящие для передачи сообщения его одноранговому узлу на другом компьютере. Понятие NetMsgServer не зависит от протокола, и NetMsgServer был построен для использования различных протоколов NetMsgServer, участвующий в передаче должен был быть согласован с протоколом использования. Возможность прозрачного расширения локального IPC между узлами поддерживается использованием прокси-портов. Сообщения, отправленные на этот прокси-сервер, получает NetMsgServer и перенаправляется прозрачно к исходному порту. Эта процедура является одним из примеров того, как Netmsgserver'ы сотрудничают, чтобы сделать прокси неотличимым от исходного порта.

Поскольку Mach предназначен для работы в сети разнородных систем, он должен предоставлять системам возможность отправлять данные в формате, понятном как отправителю, так и получателю. К сожалению, компьютеры различаются по форматам, которые они используют для хранения различных типов данных. В случае экземпляра целое число в одной системе может занять 2 байта для хранения, и самый старший байт может быть сохранен перед наименее значимым. Другая система может изменить этот порядок. Поэтому NetMsgServer использует информацию о типе, хранящуюся в сообщении, для перевода данных из формата отправителя в формат получателя. Таким образом, все данные отображаются правильно, когда они достигают своего места назначения.

NetMsgServer на данном компьютере принимает RPC, которые добавляют, ищут и удаляют сетевые порты из службы имен NetMsgServer. В качестве меры безопасности значение порта, указанное в запросе на добавление порта, должно совпадать со значением в запросе на удаление для потока, запрашивающего имя порта для удаления из базы данных. В качестве примера работы NetMsgServer рассмотрим поток на узле А, отправляющий сообщение в порт, который находится в заданном узле В. Программа просто отправляет сообщение в порт, на который у нее есть право отправки. Сообщение сначала передается ядру, которое доставляет его первому получателю, NetMsgServer на узле А. Затем NetMsgServer связывается (через информацию своей базы данных) с NetMsgServer на узле В и отправляет сообщение. NetMsgServer на узле В предоставляет ядру сообщение с соответствующим локальным портом для узла В. Наконец, ядро предоставляет сообщение принимающей задаче, когда поток в этой задаче выполняет вызов msg_receive (). Эта последовательность событий показана на рисунке 44. Mach 3.0 предоставляет альтернативу NetMsgServer как часть улучшенной поддержки мультипроцессоров NORMA. Подсистема NORMA IPC в Mach 3.0 реализует функциональность, аналогичную NetMsgServer, непосредственно в ядре Mach, обеспечивая гораздо более эффек-

тивный междоузличный IPC для ультикомпьютеров с аппаратным обеспечением для быстрого соединения. Например, устраняется трудоемкое копирование сообщений между NetMsgServer и ядром. Использование NORMA IPC не исключает использование NetMsgServer; NetMsgServer все еще можно использовать для предоставления службы Mach IPC по сетям, которые связывают ультипроцессор NORMA с другими компьютерами. В дополнение к NORMA IPC Mach 3.0 также поддерживает управление памятью в системе NORMA и позволяет задаче в такой системе создавать дочерние задачи на узлах, отличных от ее собственных. Эти функции поддерживают реализацию операционной системы с единым системным образом на мультипроцессоре NORMA. Мультипроцессор ведет себя как одна большая система, а не как совокупность небольших систем (для обоих пользователей и приложения).

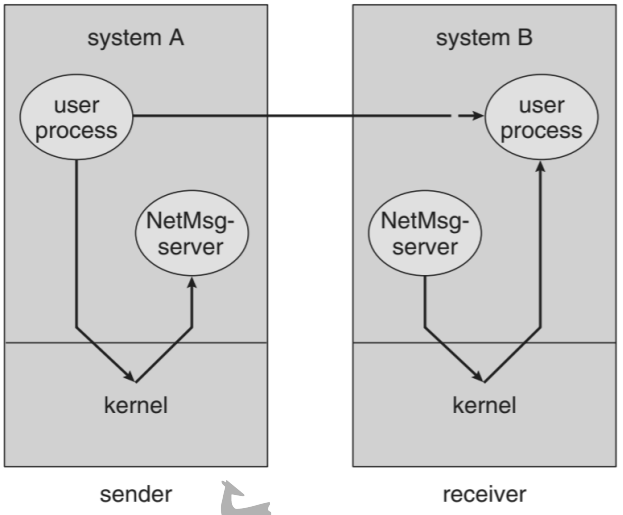


Рис. 44
Сетевая переадресация IPC NetMsgServer

Синхронизация через IPC

Механизм IPC чрезвычайно гибок и используется повсюду в Mach. Например, это может быть использовано для синхронизации потоков. Порт может использоваться как переменная синхронизации и может иметь n сообщений, отправленных ему для n ресурсов. Любой поток, желающий использовать ресурс, выполняет принимаемый вызов на этом порту. Поток получит сообщение, если ресурс доступен. В противном случае он будет ожидать порта, пока там не появится сообщение. Чтобы вернуть ресурс после использования, поток может отправить сообщение в порт. В этом отношении получение эквивалентно операции семафора wait (), а отправка эквивалентна signal(). Этот метод можно использовать для синхронизации операций семафора между потоками в одной и той же задаче, но его нельзя использовать для синхронизации между задачами, поскольку только одна задача может иметь права на порт. Для семафоров более общего назначения можно написать простую фоновую службу для реализации того же метода.

Управление памятью

Учитывая объектно-ориентированную природу Mach, неудивительно, что главной абстракцией в Mach является объект памяти. Объекты памяти используются для управления вторичным хранилищем и обычно представляют файлы, каналы или другие данные, которые отображаются в виртуальную память для чтения и записи (рис. 45). Объекты памяти могут поддерживаться менеджерами памяти на уровне пользователя, которые заменяют более традиционные встроенные в ядро пейджеры виртуальной памяти, используемые в других операционных системах. В отличие от традиционного подхода с ядром, управляя вторичным хранилищем, Mach обрабатывает объекты вторичного хранилища (обычно файлы) так же, как и все другие объекты в системе. Каждый объект имеет связанный с ним порт и может управляться сообщениями, отправляемыми на его порт. Объекты памяти — в отличие от процедур управления памятью в монолитных традиционных ядрах — позволяют легко экспериментировать с новыми алгоритмами манипулирования памятью.

Основная структура

Виртуальное адресное пространство задачи, как правило, невелико и состоит из множества дырок нераспределенного пространства. Например, отображенный в память файл помещается в некоторый набор адресов. Большие сообщения также передаются как сегменты общей памяти. Для каждого из этих примеров раздел адреса виртуальной памяти используется для предоставления потокам доступа к сообщению. Когда новые элементы отображаются или удаляются из адресного пространства, в адресном пространстве появляются дыры нераспределенной памяти. Mach не пытается сжать адресное пространство, хотя задача может завершиться неудачей (или аварийно завершиться), если в его адресном пространстве нет места для запрашиваемого региона. Учитывая, что адресные пространства составляют 4 ГБ или более, это ограничение в настоящее время не является проблемой.

Однако поддержание обычной таблицы страниц для адресного пространства объемом 4 ГБ для каждой задачи, особенно одной с дырами в ней, потребовало бы чрезмерного объема памяти (1 МБ или более). Ключ к разреженным адресным пространствам заключается в том, что пространство табличных страниц используется только для выделенных в настоящее время областей. Когда происходит сбой страницы, ядро должно проверить, находится ли страница в допустимой области, а не просто проиндексировать таблицу страниц и проверить запись. Хотя полученный в результате поиск является более сложным, преимущества сниженных требований к объему памяти и упрощенного обслуживания адресного пространства делают этот подход оправданным.

У Mach также есть системные вызовы для поддержки стандартных функций виртуальной памяти, включая выделение, освобождение и копирование виртуальной памяти. При выделении нового объекта виртуальной памяти поток может предоставить адрес для объекта или может позволить ядру выбрать адрес. Физическая память не выделяется до тех пор, пока страницы в этом объек-

те не будут доступны. Объекты виртуальной памяти также распределяются автоматически, когда задача получает сообщение, содержащее данные вне линии.

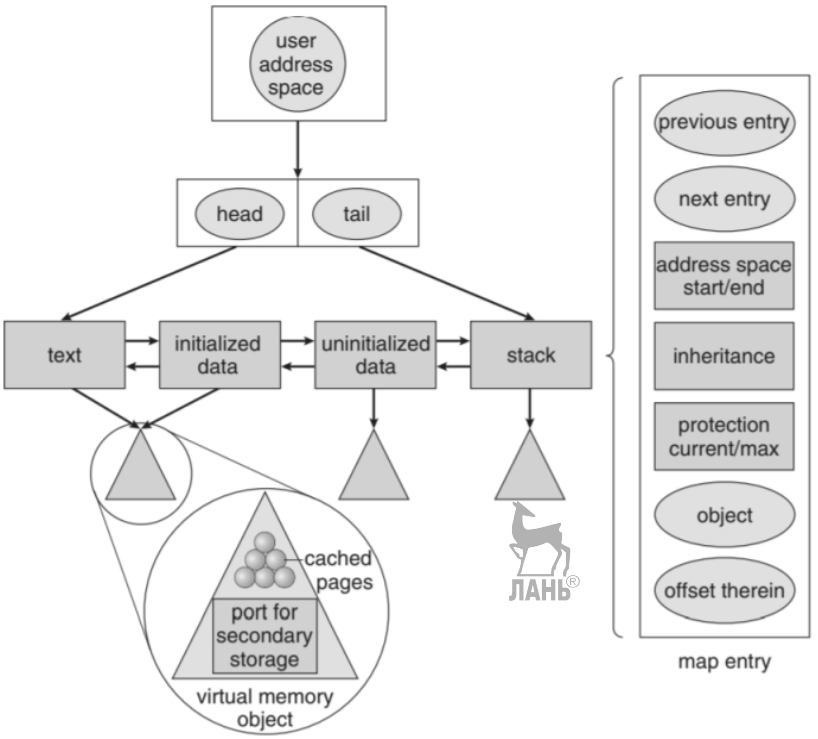


Рис. 45
Карта адресов задач виртуальной памяти Mach

Связанные системные вызовы возвращают информацию об объекте памяти в адресном пространстве задачи, изменяют защиту доступа к объекту и указывают, как объект должен передаваться дочерним задачам во время их создания (общий доступ, копирование при записи или нет).

Менеджеры памяти на уровне пользователя

Объект вторичного хранилища обычно отображается в виртуальном адресном пространстве задачи. Mach поддерживает кэш резидентных страниц памяти всех отображаемых объектов, как и в других реализациях виртуальной памяти. Однако ошибка страницы, возникающая, когда поток обращается к нерезидентной странице, выполняется как сообщение на порт объекта. Концепция, что объект памяти может быть создан и обслуживаться неядерными задачами (в отличие, например, от потоков, которые создаются и поддерживается только ядром), это важно. Конечным результатом является то, что в традиционном смысле память может быть выгружена с помощью пользовательских менеджеров памяти. Когда объект уничтожен, администратор памяти должен записать любые измененные страницы во вторичное хранилище. Mach не делает никаких предположений о содержании или важности объектов памяти, поэтому объекты памяти не зависят от ядра. В некоторых случаях менеджеров памяти на

уровне пользователя недостаточно. Например, задаче, выделяющей новую область виртуальной памяти, может не назначаться диспетчер памяти для этой области, поскольку он не представляет объект вторичного хранилища (но должен быть выгружен), или диспетчер памяти может не выполнить выгрузку страниц. Самому Mach также нужен менеджер памяти, чтобы позаботиться о его потребностях в памяти. Для этих случаев Mach предоставляет менеджер памяти по умолчанию. Диспетчер памяти Mach 2.5 по умолчанию использует стандартную файловую систему для хранения данных, которые должны быть записаны на диск, а не требует отдельного пространства подкачки, как в 4.3 BSD. В Mach 3.0 (и OSF/1) менеджер памяти по умолчанию способен использовать файлы в стандартной файловой системе или выделенные разделы диска. Диспетчер памяти по умолчанию имеет интерфейс, аналогичный интерфейсу пользовательского уровня, но с некоторыми расширениями, поддерживающими его роль диспетчера памяти, на которую можно положиться при выполнении выгрузки страниц, когда администраторы уровня пользователя этого не делают. Политика выгрузки реализована внутренним потоком ядра, фоновой службой выгрузки. Алгоритм подкачки на основе FIFO со вторым шансом используется для выбора страниц для замены. Выбранные страницы отправляются соответствующему менеджеру (пользовательского уровня или по умолчанию) для фактической выгрузки. Менеджер уровня пользователя может быть более интеллектуальным, чем менеджер по умолчанию, и он может реализовывать другой алгоритм разбиения на страницы, подходящий для объекта, который он поддерживает (то есть путем выбора какой-либо другой страницы и принудительного ее разбиения на страницы). Если администратору уровня пользователя не удается уменьшить резидентный набор страниц, когда ядро об этом просит, вызывается диспетчер памяти по умолчанию, и он выводит на экран менеджера уровня пользователя, чтобы уменьшить размер резидентного набора менеджера уровня пользователя. Если менеджер уровня пользователя восстановится после проблемы, из-за которой он не смог выполнить свои собственные постраничные выпуски, он коснется этих страниц (в результате чего ядро снова их загрузит), а затем может выгрузить их по своему усмотрению. Если потоку необходим доступ к данным в объекте памяти (например, в файле), он вызывает системный вызов vm_map (). В этот системный вызов включен порт, который идентифицирует объект и диспетчер памяти, который отвечает за регион. Ядро выполняет вызовы на этом порту, когда данные должны быть прочитаны или записаны в этом регионе. Дополнительная сложность заключается в том, что ядро выполняет эти вызовы синхронно, поскольку для ядра было бы неразумно ожидать в потоке пользовательского уровня. В отличие от ситуации с pageout, у ядра нет ресурсов, если его запрос не удовлетворен anager внешней памяти. Ядро ничего не знает о содержимом объекта или о том, как этим объектом следует манипулировать.

Менеджеры памяти отвечают за согласованность содержимого объекта памяти, отображаемого задачами на разных машинах. (Задачи на одном компьютере совместно используют одну копию сопоставленного объекта памяти.) Рассмотрим ситуацию, когда задачи на двух разных машинах пытаются изме-

нить одну и ту же страницу объекта одновременно. Менеджер должен решить, должны ли эти изменения быть упорядоченными. Консервативный менеджер, реализующий строгую согласованность памяти, вынудит упорядочить модификации путем предоставления доступа на запись только к одному ядру за раз. Более сложный менеджер мог бы разрешить одновременный доступ к обоим доступам (например, если менеджер знал, что две задачи модифицировали отдельные области на странице и что он мог бы успешно объединить изменения в будущем). Большинство менеджеров внешней памяти, написанных для Mach (например, те, которые реализуют сопоставленные файлы), не реализуют логику для работы с несколькими ядрами из-за сложности такой логики. Когда первый вызов vm_map () сделан для объекта памяти, ядро отправляет сообщение на порт диспетчера памяти, переданный в вызове, вызывая подпрограмму менеджера памяти init(), которую менеджер памяти должен предоставить как часть своей поддержки объекта памяти. Два порта, передаваемые диспетчеру памяти, — это порт управления и порт имени. Управляющий порт используется диспетчером памяти для предоставления данных ядру — например, страниц, которые должны быть резидентными. Порты имени используются повсюду в Mach. Они не получают сообщения, но используются просто как точки отсчета и сравнения. Наконец, объект памяти должен ответить на вызов init () диспетчера памяти с помощью вызова атрибута set () объекта памяти, чтобы указать, что он готов принимать запросы. Когда все задачи с правами на отправку объекта памяти отказываются от этих прав, ядро освобождает порты объекта, освобождая диспетчер памяти и объект памяти для уничтожения. Для поддержки менеджера внешней памяти требуется несколько вызовов ядра. Вызов vmmap() только что обсуждался. Кроме того, некоторые команды получают и устанавливают атрибуты и обеспечивают блокировку на уровне страницы, когда это необходимо (например, после сбоя страницы, но до того, как менеджер памяти вернул соответствующие данные). Другой вызов используется диспетчером памяти для передачи страницы (или нескольких страниц, если используется упреждающее чтение) ядру в ответ на ошибку страницы. Этот вызов необходим, поскольку ядро асинхронно вызывает диспетчер памяти. Наконец, несколько вызовов позволяют диспетчеру памяти сообщать ядру об ошибках. Сам диспетчер памяти должен обеспечивать поддержку нескольких вызовов, чтобы он мог поддерживать объект. Мы уже обсуждали объект памяти init() и другие. Когда поток вызывает ошибку страницы на странице объекта памяти, ядро отправляет запрос данных объекта памяти в порт объекта памяти от имени, вызывающего поток. Поток переводится в состояние ожидания до тех пор, пока менеджер памяти не вернет страницу в вызове data (), предоставленном объектом памяти, или вернет ядру соответствующую ошибку. Любые из страниц, которые были изменены, или любые «драгоценные страницы», которые ядру необходимо удалить из резидентной памяти (например, из-за устаревания страницы), отправляются в объект памяти через функцию datadatawrite(). Это страницы, которые, возможно, не были изменены, но не могут быть удалены, поскольку в противном случае диспетчер памяти больше не сохраняет копию. Менеджер emory объявляет эти страницы драгоценными и ожидает, что ядро вернет их, когда

они будут удалены из памяти. Драгоценные страницы сохраняют ненужное дублирование и копирование памяти. В текущей версии Mach не позволяет анонимам внешней памяти напрямую влиять на алгоритм замены страниц. Mach не экспортирует информацию о доступе к памяти, которая понадобилась бы внешней задаче, например, для выбора наименее недавно использованной страницы. Методы предоставления такой информации в настоящее время изучаются. Однако менеджер внешней памяти по-прежнему полезен по ряду причин:

- Он может отклонить замену ядра, если знает о лучшем претенденте (например, замена страницы MRU).
- Он может отслеживать объект памяти, который он поддерживает, и запрашивать страницы, которые будут выгружены до того, как использование памяти вызовет фоновую службу выгрузки страниц Mach.
- Это особенно важно для поддержания согласованности вторичного хранилища для потоков на нескольких процессорах.
- Он может контролировать порядок операций над вторичным хранилищем для обеспечения ограничений согласованности, требуемых системами управления базами данных. Например, в журнале транзакций транзакции должны быть записаны в файл журнала на диске, прежде чем они изменят данные базы данных.
- Он может контролировать доступ к файлам.

Общая память

Mach использует разделяемую память для уменьшения сложности различных системных функций, а также для эффективного предоставления этих функций. Общая память обычно обеспечивает чрезвычайно быструю связь между процессами, снижает накладные расходы на управление файлами и помогает поддерживать многопроцессорность и управление базами данных. Однако Mach не использует разделяемую память для всех этих традиционных ролей совместно используемой памяти. Например, все потоки в задаче совместно используют память этой задачи, поэтому в задаче не требуется формальное средство совместной памяти. Однако Mach по-прежнему должен предоставлять традиционную разделяемую память для поддержки других конструкций операционной системы, таких как системный вызов UNIXfork(). Очевидно, что для задач на нескольких машинах трудно совместно использовать память и поддерживать согласованность данных. Mach не пытается решить эту проблему напрямую; скорее, это обеспечивает возможности, позволяющие решить проблему. Mach поддерживает согласованную совместную память только тогда, когда память совместно используется задачами, выполняющимися на процессорах, которые совместно используют память. Родительская задача способна объявить, какие области памяти должны быть унаследованы ее дочерними, а какие должны быть доступны для чтения и записи. Эта схема отличается от наследования при копировании при записи, в котором каждая задача поддерживает свою собственную копию любых измененных страниц. Доступный для записи объект адресуется из карты адресов каждой задачи, и все изменения вносятся в

объект адресуется из карты адресов каждой задачи, и все изменения вносятся в одну и ту же копию. Потоки в задачах отвечают за координацию изменений в памяти, чтобы они не мешали друг другу (путем одновременной записи в одно и то же место). Эта координация может быть выполнена с помощью обычных методов синхронизации: критических секций или блокировок взаимоисключения. Для случая, когда память распределяется между отдельными машинами, Mach позволяет использовать менеджеры внешней памяти. Если набор несвязанных задач хочет разделить часть памяти, задачи могут использовать один и тот же диспетчер внешней памяти и получать к нему доступ к тем же областям вторичного хранилища. Разработчик этой системы должен был бы написать задачи и внешний пейджер. Этот пейджер может быть настолько простым или сложным, насколько это необходимо. Простая реализация не позволила бы читателям во время записи страницы. Любая попытка записи может привести к тому, что пейджер сделает недействительной страницу во всех задачах, которые в данный момент обращаются к ней. Затем пейджер разрешил запись и подтвердил читателям новую версию страницы. Читатели будут просто ждать ошибки страницы, пока страница снова не станет доступной. Mach обеспечивает такой менеджер памяти: сервер сетевой памяти (NetMemServer). Для мощных компьютеров конфигурация NORMA в Mach 3.0 обеспечивает поддержку, аналогичную стандартной части ядра. Эта подсистема XMM позволяет компьютерным системам использовать менеджеры внешней памяти, которые не включают логику для работы с несколькими ядрами. Подсистема XMM отвечает за поддержание согласованности данных между несколькими ядрами, которые совместно используют память, и делает эти ядра единым ядром для менеджера памяти. UM-система XMM также реализует логику виртуального копирования для сопоставленных объектов, которые она выводит из строя. Эта логика виртуального копирования включает в себя как копирование по ссылке между ядрами мультикомпьютера, так и сложные оптимизации копирования при записи.

Резюме

Операционная система Mach разработана с учетом последних инновационных исследований в области операционных систем с целью создания полнофункциональной, технически совершенной операционной системы.

Операционная система Mach была разработана с учетом трех важных целей:

– эмулирование 4.3 BSD UNIX, чтобы исполняемые файлы из системы UNIX могли корректно работать в среде Mach;

– наличие современной операционной системы, которая поддерживала бы множество моделей памяти, а также параллельные и распределенные вычисления;

– разработка ядра, которую проще и легче модифицировать, чем 4.3 BSD.

Как мы уже показали, Mach успешно продвигалась к достижению этих целей.

Mach 2.5 включает в свое ядро 4.3 BSD, которое обеспечивает эмуляцию необходимых служб. Код BSD 4.3 был модифицирован для обеспечения поддержки примитивов Mach. Это изменение позволяет запускать код поддержки BSD 4.3 в пользовательском пространстве в системе Mach 3.0. Mach использует легкие процессы в форме нескольких потоков выполнения в одной задаче (или адресном пространстве) для поддержки многопроцессорных и параллельных вычислений. Широкое использование сообщений в качестве единственного метода связи обеспечивает полноту и эффективность механизмов защиты. Интегрируя сообщения с системой виртуальной памяти, Mach также обеспечивает эффективную обработку сообщений. Наконец, благодаря тому, что система виртуальной памяти использует сообщения для связи с фоновыми службами, управляющими резервными хранилищами BibliographicNotes 25, Mach обеспечивает большую гибкость в разработке и реализации задач управления объектами памяти. Предоставляя низкоуровневые или примитивные системные вызовы, из которых могут быть построены более сложные функции, Mach уменьшает размер ядра, одновременно разрешая эмуляцию операционной системы на уровне пользователя, во многом подобно системам виртуальных машин IBM.

Сравнение

Исследования показали, что проблема производительности IPC не так страшна, как считается. Напоминаем, что односторонний вызов на BSD занимает 20 микросекунд, в то время как на Mach — 114, 11 из которых — это переключение контекста, идентичного BSD. Дополнительно 18 используется менеджером памяти для отображения сообщения между непривилегированной средой исполнения и привилегированной (user-space и kernel-space). Это добавляет 31 микросекунду, что дольше традиционного вызова, но не намного.

Оставшаяся часть проблемы — это проверки прав доступа к порту сообщений. В то время, как это выглядит очень важным, фактически, это требуется только на Unix-системах. К примеру, однопользовательская система, запущенная на мобильном телефоне, может не нуждаться в таких возможностях, и это тот тип систем, в которых Mach может быть использован. Однако Mach создает проблемы: когда память перемещается в ОС, другие задачи могут не нуждаться в этом. DOS и ранние Mac OS имели единое адресное пространство, разделяемое всеми процессами, поэтому в таких системах отображение памяти — пустая трата времени.

Эти реализации положили начало второму поколению микроядер, которое уменьшает сложность системы, размещая большую часть функциональности в непривилегированном режиме исполнения.

Ярким примером микроядер второго поколения, основанных на Mach, является микроядро L4, разработанное Йохеном Лидтке в 1993 г.

Архитектура микроядра L4 оказалась успешной. Было создано множество реализаций ABI и API микроядра L4. Все реализации стали называть семейством микроядер L4. Реализация Лидтке неофициально была названа «L4/x86».

Во время работы над микроядром L3 Йохен Лидтке обнаружил недостатки микроядра Mach. Желая повысить производительность, Лидтке стал состав-

лять код нового микроядра на языке ассемблера с использованием особенностей архитектуры процессоров Intel i386. Новое микроядро получило название «L4» (от «4-я работа Liedtke»).

В 1993 г. реализация микроядра L4 была закончена. Компонент IPC оказался в 20 раз быстрее IPC из микроядра Mach [1].

ОС, построенные на микроядрах первого поколения (в частности, на микроядре Mach), отличались низкой производительностью. Из-за этого в середине 1990-х гг. разработчики стали пересматривать концепцию микроядерной архитектуры. В частности, плохую производительность микроядра Mach объясняли переносом компонента, ответственного за IPC, в пространство пользователя.

Некоторые компоненты микроядра Mach были возвращены назад — внутрь микроядра. Это нарушало саму идею микроядер (минимальный размер, изоляция компонентов), но позволило увеличить производительность ОС.

Исследователи искали причины низкой производительности микроядра Mach и тщательно анализировали компоненты, важные для обеспечения хорошей производительности. Анализ показал, что ядро выделяло процессам слишком большой working set (много памяти), в результате чего при обращении ядра к памяти постоянно происходили кэш-промахи (*англ*. cache misses) [6]. Анализ позволил сформулировать новое правило для разработчиков микроядер — микроядро должно проектироваться так, чтобы компоненты, важные для обеспечения высокой производительности, помещались в кэш-процессора (желательно, первого уровня (*англ*. level 1, L1) и желательно, чтобы в кэше еще осталось немного места).

Из-за резкого скачка в производительности компонента IPC существующие ОС оказались неспособны обработать возросший поток сообщений IPC. Несколько университетов (например, технический университет Дрездена, университет Нового Южного Уэльса), институтов и организаций (например, IBM) начали создавать реализации L4 и строить на их основе новые ОС.

Аналитический обзор и сравнение возможностей операционных систем для мобильных устройств

В рамках курса «Операционные системы» студенты должны иметь возможность познакомиться, во-первых, с основными компонентами операционных систем (ОС), во-вторых, с основными разновидностями операционных систем. Одним из быстро развивающихся направлений в разработке операционных систем является направление ОС для мобильных устройств. Следует помнить, что ОС для мобильных устройств имеют свою специфику, связанную с различиями в требованиях к мобильному устройству в отличие от настольного или портативного компьютера, а также зависящую от их реализации. К числу таких особенностей можно отнести следующие.

- Ограничения по памяти и скорости процессора.
- Различные дизайнерские и конструкционные отличия в экранах и экранных навигаторах разных моделей мобильных устройств.
- Совместимость с основными форматами файлов.

• Мультимедийные возможности.

• Поддержка коммуникационных и сетевых технологий.

В данной статье мы рассматриваем развитие мобильных ОС, опираясь на информацию, имеющуюся в открытых источниках Интернета.

В настоящее время выбор мобильного устройства сводится, в основном, к выбору операционной системы. Ведущие фирмы-производители мобильных устройств поддерживают собственные операционные системы либо операционные системы (далее ОС), приобретенные вместе с их фирмами-разработчиками.

ОС Symbian была разработана консорциумом Symbian (Nokia, Ericsson, Psion, Motorola), который был основан в 1998 г. Объединение Symbian Foundation занимался разработкой и поддержкой единой мобильной платформы, подходящей для мобильных устройств различных компаний, на основе Simbian OS, финансировала данную разработку фирма Nokia.

На конец 2009 г. рынок мобильных OS распределялся следующим образом: Blackberry OS — 20%, Windows Mobile — около 9%, Google Android — около 5%, Symbian OS — 47% [1]. У Simbian OS было разработано несколько модификаций операционной системы, что было связанно с разными типами устройств, самые распространенные из них: Series 90, UIQ, Series 60 и FOMA в Японии.

Основной платформой для смартфонов компании Sony Ericsson являлась модификация UIQ. Отличительной особенностью данной модификации была возможность работы на устройствах с сенсорным экраном.

Для смартфонов финской компании Nokia была разработана модификация Symbian OS Series 60. Данная модификация была разработана для устройств с телефонной клавиатурой, имеющей сокращенный набор кнопок.

Для устройств с полноразмерной клавиатурой была разработана модификация Series 90.

В смартфонах одного из крупнейших сотовых операторов Японии NTT DoCoMo, использовалась модификация Symbian OS — FOMA. По заказам данного сотового оператора, смартфоны на FOMA производили Mitsubishi, Fujutsu и Motorola.

В августе 2011 г. компания Nokia представила обновленную версию Symbian, которая получила название Symbian Belle.

С точки зрения безопасности, данную версию ОС можно считать безопасной, так как вирусов для нее не существовало, была лишь вероятность, что на ней могут запуститься несколько вирусов для Symbian 9.

Интерфейс данной версии ОС радовал домашними экранами, для каждого из которых можно выбрать свои обои и масштабируемые в 5 различных размерах виджеты, в том числе новый виджет «переключатель». В операционной системе появились улучшенная многозадачность, выпадающие меню и панели задач, доступные на любом домашнем экране.

В Symbian Belle мы наконец-то можем увидеть первые плоды сотрудничества Nokia и Microsoft. В ее состав вошли приложения Lync, Sharepoint, OneNote, Exchange ActiveSync и PowerPoint Broadcaster.

Но одной из наиболее интересных особенностей Symbian Belle стала поддержка технологии NFC. С ее помощью можно быстро обмениваться данными с другими устройствами и использовать различные аксессуары, типа колонок и наушников.

Nokia вела заметную работу над Symbian, но, строго говоря, этого уже мало, чтобы поддержать ее популярность на былом уровне. В чем дело? А в том, что вокруг этой ОС ничего нет. Нет той самой экосистемы, которую строят Apple, Google и Microsoft.

В январе 2013 г. было сделано официальное заявление Nokia: «Устройство, показавшее наши возможности визуализации и вышедшее на рынок в середине 2012 г., было последним устройством Nokia на Symbian». После чего операционная система Symbian была переведена в режим поддержки.

Следующая операционная система, которую хочется рассмотреть — Samsung Bada.

Данная операционная система была разработана компанией Samsung Electronics. Первоначально за основу был взят опыт разработки и развития платформы SHP (Samsung Hand-Held Platform). Анонсирована данная операционная система была 10 ноября 2009 г., а выпущена в 2010 г.

Bada являлась платформой закрытого типа, для которой могут быть разработаны так называемые native-приложения, то есть приложения, разрабатываемые непосредственно под платформу, с использованием SDK от производителя. В результате было возможным использовать неограниченное количество вариантов аппаратных решений, и, как следствие, ОС (Linux, RTOS, Nucleus).

В bada использовался интерфейс, основанный на популярном интерфейсе TouchWiz. Также он поддерживал сенсорные приложения с привязкой к контенту. Благодаря этому разработчики могли создавать различные приложения, которые использовали акселерометр, датчик высоты, движения, активности и т. д. для создания приложений нового поколения.

Гибкость данной платформы позволило использовать ее на огромном количестве устройств, что было невозможно в случае с другими платформами.

В конце 2011 г. была официально представлена новая версия bada, получившая номер 2.0. Рассмотрим ее поподробнее.

Несмотря на то что под bada не было замечено ни одного вируса или вредоносной программы, компания Sumsung все же собственноручно создала для своей ОС утилиту Mobile Scan, которую можно скачать в SamsungApps.

Интерфейс представляет несколько рабочих столов, Живую Панель (Live Panel), статусную строку и возможность создавать папки и складывать в них иконки. Но что огорчает — это плохая продуманность навигации, которая связана с тем, что TouchWiz создавался в первую очередь под Android с его тремя функциональными клавишами. Контекстное меню приложений отсутствует, поэтому их настройки попадают в общие настройки телефона. Из-за этого bada 2.0 производит впечатление странного и не всегда удобного гибрида между Android и iOS.

Единственными устройствами на ОС bada стала линейка Sumsung Wave.

Данная операционная система способна работать с большим числом различных сервисов, поддерживает протокол Exchange ActiveSync. Приложение Social Hub позволяет следить за всей социальной активностью в одном месте. Аппараты на базе bada 2.0 стандартно поставляются с предустановленным офисным пакетом Polaris Office, который можно использовать для чтения и редактирования офисных документов, в том числе и PDF. Есть и встроенный файловый менеджер. Bada-телефоны поставляются с пакетом приложений «Яндекса», в числе которых карты, Яндекс. Метро и т. п. Главный недостаток операционной системы bada — практически полное отсутствие сторонних приложений, причем компания Samsung усугубила ситуацию собственными руками, сделав bada 2.0 несовместимой с программами для bada 1.x.

25 февраля 2013 г. Samsung официально заявил о слиянии bada с Tizen — другой мобильной платформой, разрабатываемой совместно с Intel, Asus и Acer. Разработки в рамках bada прекращаются, а все наследие проекта будет интегрировано в Tizen.

BlackBerry OS — это операционная система, работающая на популярных в США коммуникаторах BlackBerry фирмы RIM. Современная версия — BlackBerry 10 OS — базируется на *QNX*.

К особенностям BlackBerry 10 можно отнести: возможность с помощью жестов переходить от одного приложения к другому, плиточный интерфейс, интеллектуальную клавиатуру, которая подбирает нужное слово в зависимости от стиля беседы. Уникальное приложение для камеры автоматически записывает кадры до того, как пользователь начнет съемку, то есть ни один кадр не будет пропущен. Вывод предупреждений о приближении к порогу использования трафика, заданного пользователем. IntelligentAssistant — помощник, которому можно вводить запросы с клавиатуры или отдавать команды голосом. BlackBerry Blend — новое ПО, позволяющее получить доступ к функциям телефона с компьютера, ноутбука и планшета, в том числе и с iPad.

Первый телефон, работающий на BlackBerry 10 был BlackBerry Z10.

Так как компания RIM, прежде всего, ориентирована на корпоративных клиентов, то и уровень безопасности во всех версиях операционных систем приближен к идеалу.

По сравнению с iOS и Android интерфейс данной ОС довольно скучен, но иконки приложений не имеют полей вокруг себя и существует интересное новшество — это возможность отключать «домашний» экран, то есть делать так, чтобы на нем не отображались иконки приложений. Также имеется функция восстановления приложений после перезагрузки, что является весьма полезным. В каждом приложении есть «основная кнопка», которая отвечает за самую востребованную функцию.

BlackBerry OS доступна для установки исключительно на устройства линейки Blackberry. По оценке исследовательской компании IDC сейчас на ее долю приходится 0,4% всемирного рынка данной отрасли (по итогам 2014 г.).

Основные приложения, которые можно купить или скачать в специальном магазине App World, нацелены на решение бизнес задач (курсы валют, рейтинги, статистика и проч.). В телефонах BlackBerry идет предустановленный

магазин Android-приложений Amazon, который призван скомпенсировать отсутствие некоторых очень важных приложений в BlackBerry World. Доступен Skype — приложение, написанное на Android NDK.

Устройства на этой системе широко используются в основном в США, так как спецслужбы некоторых стран не заинтересованы в использовании этих смартфонов в своей стране из-за того, что все входящие/исходящие данные шифруются с помощью AES.

А теперь проанализируем три самые популярные на сегодняшний день операционные системы — iOS, Android, Windows — в отношении следующих характеристик.

- Надежность.
- Безопасность.
- Интерфейс.
- Многоплатформенность.
- Приложения (в особенности, возможность разработки).

Надежность: iOS является закрытой. Мобильное устройство поставляется со всеми необходимыми заводскими настройками и избавляет пользователя от необходимости проводить настройку своего смартфона. Однако, с другой стороны, это и является минусом, так как нет возможности расширить память смартфона, нет дополнительных структурных компонентов, нет сменных кадров для «домашнего» экрана. Конечно, не стоит забывать о существовании джейлбрейка — процедуры для открытия файловой системы iOS, которая позволяет пользователю производить модификации iPhone. Платформа Android является открытой. Мобильное устройство перед использованием требует тщательной настройки, что, с одной стороны, требует времени, а с другой стороны, позволяет пользователю учесть все свои требования к смартфону и настроить его «под себя». Windows занимает место между iOS и Android — настройки минимальны, но все же существуют. Например, можно изменять размер «плиток» домашнего экрана. Одна из положительных особенностей — это возможность читать и редактировать файлы Word, Excel, PowerPoint и записи OneNote прямо в телефоне. Последние модели смартфонов на Android и Windows предусматривают чтение карт памяти объемом до 32 Гб.

С точки зрения безопасности, iOS и Windows имеют встроенную защиту от вирусов, так что автономные приложения этого действия ей не нужны. А вот Android за открытость платформы приходится расплачиваться огромным количеством вирусов, так что без установки дополнительных приложений, отвечающих за безопасность, пользователю не обойтись.

У iOS простой пользовательский интерфейс и однокадровый «домашний» экран. По простоте и легкости использования iOS от Apple — явный и несомненный лидер. Интерфейс Android разработан с использованием двумерной и трехмерной графики (библиотеки OpenGL). ОС располагает многокадровым экраном и возможностью выносить компоненты приложений на «домашние» экраны для быстрого доступа. Windows обеспечивает пользователю весьма своеобразную рабочую среду, используются «плитки», которые прокручиваются по вертикали и могут быть настроены для быстрого запуска. Все статичные

иконки заменены на так называемые «живые элементы» (Live Tiles), которые отражают информацию в режиме реального времени без участия пользователя. Но пользователям требуется время, чтобы привыкнуть к этому совершенно новому интерфейсу.

Если анализировать многоплатформенность, то iOS доступна для установки исключительно на Apple-устройства, что ставит Apple в особое положение. Android не имеет эксплуатационных ограничений, и на ней работают изделия различных марок, например, HTC, Samsung, Motorola, LG и даже OMobile. Google Android устанавливается не только на смартфоны, данная платформа подходит и для нетбуков. Так, например, Android уже стоит на ряде моделей Asus EE PC, а также портирован на нетбуки компаний MSI, Dell и Acer. Windows также не имеет ограничений по установке на различные устройства, за исключением лишь того, что предъявляются некоторые системные требования. Производители могут производить модификации при соблюдении условия, что операционная система будет узнаваемой на любой платформе. Компания IDC в своем последнем докладе, отражающем расстановку сил на глобальном рынке операционных систем по итогам 2014 г., опубликовала следующие данные: iOS — 14,8%, Android — 81,5%, Windows — 2,7%.

По состоянию на 1 мая 2014 г. магазин приложений *AppStore* содержит более 1,4 млн приложений, которые все вместе были загружены более 50 млрд раз. Другие приложения могут быть разработаны с помощью *Xcode* для Mac и iPhone, iPod Touch и iPad, Codea для *iPad* и могут распространяться только через App Store. При этом Apple оставляет за собой право отказать разработчику в публикации приложения, если сочтет его содержание оскорбительным или непристойным, что практически гарантирует отсутствие в App Store вредоносных программ или приложений сомнительного содержания. Интернет-магазин Google Play работает в 190 странах и насчитывает более 700 тысяч приложений, а за время работы сервиса набралось около 25 млрд скачиваний. Для Android нет разницы между основными приложениями телефона и сторонним программным обеспечением — можно изменить даже программу для набора номера или заставку экрана. Система имеет свою собственную интегрированную среду для разработки приложений — Android SDK, включающий эмулятор мобильных устройств, средства отладки, профилирования, а также plug-in к популярной среде Eclipse для разработки Java-приложений. Для устройств на Windows Phone предусмотрен интернет-магазин программ и игр Windows Phone Store, доступный в 191 стране. На сегодняшний день (август 2014) количество приложений в магазине составляет 300 тысяч. В августе этого же года количество загрузок достигло 2 млрд. Позже появилась возможность устанавливать приложения и игры с *SD*-карты вручную. Для разработки приложений и игр используется *Silverlight* или *XNA*. Microsoft выпустила инструментарий разработчика Windows Phone SDK, для которого необходимы *Visual Studio* 2010 Express for Windows Phone, Expression Blend 4 for Windows Phone, XNA Game Studio 4.0.

Итак, сегодня рынок мобильных ОС поделен между тремя игроками: Apple (iOS), Google (Android) и Microsoft (Windows). Хотя доля последней ОС и очень мала, она все же внушает надежды, ведь популярная раньше BlackBerry

уже осталась позади. Но на самом деле количество альтернативных ОС для мобильных устройств больше и каждый год они демонстрируют свои наработки на Mobile World Congress. Это Jolla (Sailfish OS), Mozilla (Firefox OS) и Canonical (Ubuntu Touch). Jolla и Canonical делают ставку на Linux-энтузиастов и гиков, Mozilla пытается зарекомендовать себя на рынках развивающихся стран и налаживает связи с операторами. Конечно, уже сейчас понятно, что ни одна из них не сможет тягаться с Apple, Google, Microsoft и на передел рынка в будущем вряд ли можно надеяться, но альтернативные операционные системы все же нужны по одной простой причине: их большая открытость дает более широкие возможности для самых безумных экспериментов. А ведь именно такие эксперименты двигают индустрию вперед.

ПРИЛОЖЕНИЕ

Примеры лабораторных работ по изучаемым темам

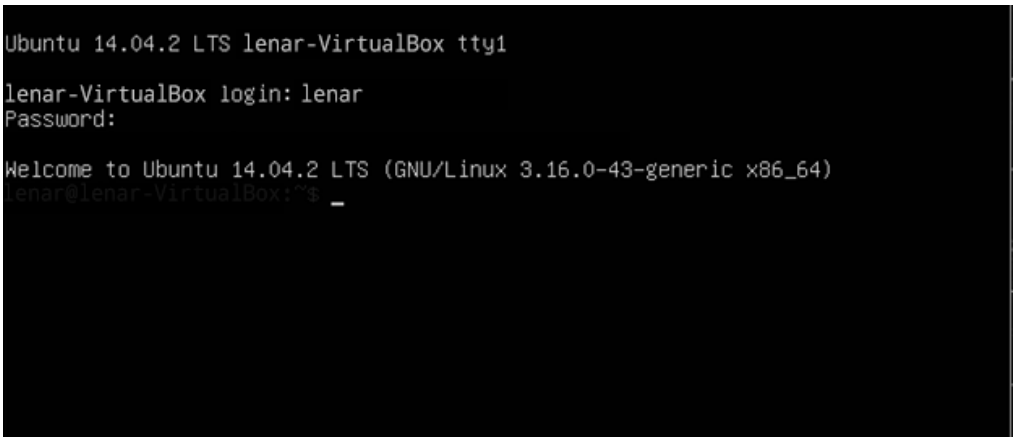
Лабораторная работа № 1

Цели и задачи

Начальное знакомство с системой, вход в систему, работа в терминальном режиме, изучение основных команд UNIX, получение начальных сведений о структуре каталогов в UNIX. Работа со справочной системой. Удаленный вход в систему. Вход в систему, работа в терминальном режиме, изучение основных команд UNIX.

1. По умолчанию Runtu запускает шесть символьных и один графический виртуальный терминал. По умолчанию система загружается в графическом терминале. Переключение в символьный терминал осуществляется клавишами **<Ctrl> - <Alt> - <F#>**, где # — номер символьного терминала, обратный переход в графический осуществляется — **<Ctrl> - <Alt> - <F7>**.

При входе в систему выводится: **имя машины login**, пользователем вводится логин и пароль. В случае успешного входа, система готова к вводу команды.



Доступ к удаленным терминалам осуществляется с помощью команды $ **ssh <Имя пользователя> <IP адрес удаленной машины>**. Для доступа к своей машине вместо IP-адреса — **localhost**.

Утилита **mc** — это удобная оболочка для работы с файлами. Запуск производится с помощью команды **$ mc**.

Для получения информации о зарегистрированных пользователях существует несколько команд:

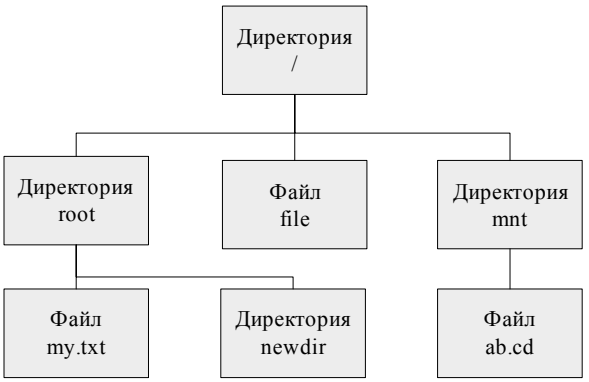
– **who** — кто из пользователей работает на машине;

– **w** — показывает, кто на данный момент вошел в систему, наряду с другой полезной информацией такой, как время работы или нагрузкой процессора;

– **finger —** системная информация о пользователе.

```
lenar@lenar-VirtualBox:~$ who
lenar    :0           2016-02-17 21:37 (:0)
lenar    pts/7        2016-02-17 21:38 (:0.0)
lenar@lenar-VirtualBox:~$ w
 21:54:26 up 16 min,  2 users,  load average: 0,01, 0,05, 0,08
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
lenar    :0       :0               21:37   ?xdm?   7.77s  0.04s init --user
lenar    pts/7    :0.0             21:38    0.00s  0.06s  0.00s w
lenar@lenar-VirtualBox:~$ finger lenar
Login: lenar                            Name: Lenar
Directory: /home/lenar                  Shell: /bin/bash
On since Wed Feb 17 21:37 (SAMT) on :0 from :0 (messages off)
On since Wed Feb 17 21:38 (SAMT) on pts/7 from :0.0
   2 seconds idle
No mail.
No Plan.
```

2. Все файлы, доступные в операционной системе UNIX, объединяются в древовидную логическую структуру. Файлы могут объединяться в каталоги или в директории. Все файлы находятся в какой-либо директории. Директории, в свою очередь, могут входить в состав других директорий. Только корневая папка не входит ни в какие директории. Название корневой папки «/». В наименовании других директорий данный символ не допускается. Нельзя создавать имена большей длины, чем это предусмотрено операционной системой (255 символов для Linux) и использовать символ NUL. Для обращения к файлам используется путь, который начинается корневым каталогом, далее промежуточные каталоги (разделяются **«/»**) и имя файла с расширением. Этот путь называется полным именем файла.



Каталоги /etc, /bin, /usr, /proc находятся в корневом каталоге системы. В каталоге /etc находятся конфигурационные файлы, которые обычно можно отредактировать вручную в текстовом редакторе. В каталоге /bin находятся основные двоичные пользовательские модули (программы), которые должны присутствовать, если система монтируется в однопользовательском режиме. Приложения, например, Firefox, хранятся в /usr/bin, в то время как важные системные программы и утилиты, такие как командная оболочка bash, расположены в каталоге /bin. В каталоге /usr находятся приложения и файлы, используемые пользователями, в отличие от приложений и файлов, используемых систе-

мой. Например, не очень важные приложения находятся в каталоге /usr/bin, а не в каталоге /bin, а не очень важные двоичные файлы, предназначенные для системного администрирования, находятся в каталоге /usr/sbin, а не в каталоге /sbin. Библиотеки для них находятся внутри каталога /usr/lib. Каталог /proc похож на каталог /dev, в котором хранятся файлы устройств, он не содержит стандартных файлов. В нем находятся специальные файлы, в которых представлена информация о системе и о процессах.

3. Основные информационные команды и команды управления процессами.

Команды	Описание
hostname	Вывести или изменить сетевое имя машины
whoami	Вывести имя, под которым вы зарегистрированы
date	Вывести или изменить дату и время
time	Получить информацию о времени, нужном для выполнения процесса
who	Определить, кто из пользователей работает на машине
rwho -a	Определение всех пользователей, подключившихся к вашей сети. Для выполнения этой команды требуется, чтобы был запущен процесс rwho. Если такого нет — запустите «setup» под суперпользователем
finger [имя_пользователя]	Системная информация о зарегистрированном пользователе. Попробуйте: finger root
ps -a	Список текущих процессов
df -h	(=место на диске) Вывести информацию о свободном и используемом месте на дисках (в читабельном виде)
Arch или uname -m uname -r	Отобразить архитектуру компьютера. Отобразить используемую версию ядра
find / -name file1 find / -user user1	Найти файлы и директории с именем file1. Поиск начать с корня (/). Найти файл и директорию, принадлежащие пользователю user1. Поиск начать с корня (/)
top	Отобразить запущенные процессы, используемые ими ресурсы и другую полезную информацию (с автоматическим обновлением данных)
Kill -9 98989 или kill -KILL 98989	«Убить» процесс с PID 98989 «на смерть» (без соблюдения целостности данных)

```
lenar@lenar-VirtualBox:~$ hostname
lenar-VirtualBox
lenar@lenar-VirtualBox:~$ whoami
lenar
lenar@lenar-VirtualBox:~$ date
Ср. февр. 17 18:20:05 SAMT 2016
lenar@lenar-VirtualBox:~$ time

real    0m0.000s
user    0m0.000s
sys     0m0.000s
```

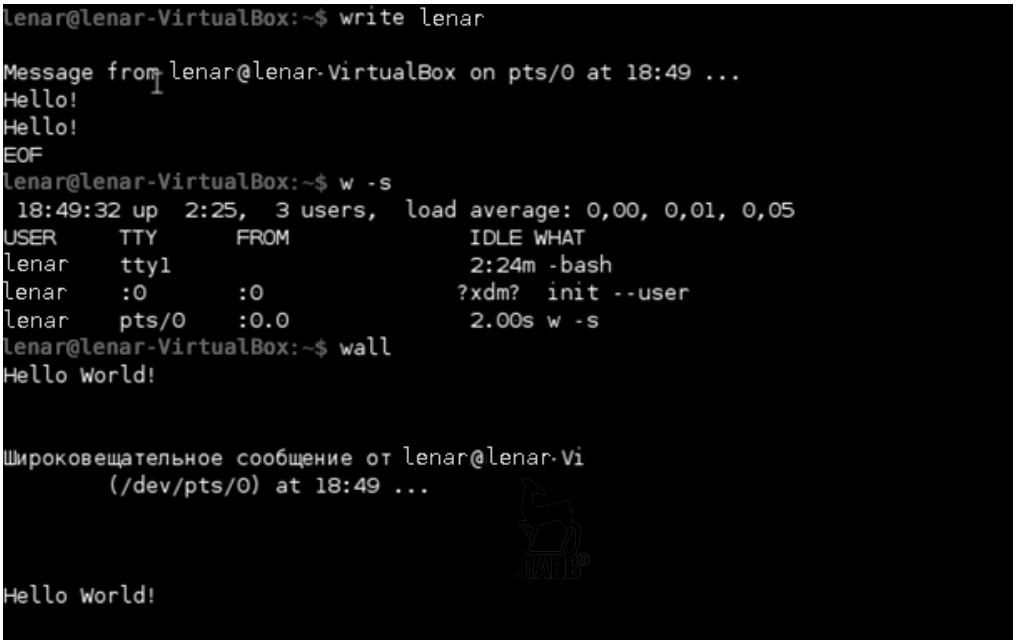
```
top - 22:39:37 up  1:04,  2 users,  load average: 0,01, 0,07, 0,12
Tasks: 178 total,   1 running, 176 sleeping,   0 stopped,   1 zombie
%Cpu(s):  0,0 us,  0,3 sy,  0,0 ni, 99,7 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 s
КиБ Mem:    759500 total,   694300 used,    65200 free,    60780 buffers
КиБ Swap:   784380 total,    60080 used,   724300 free.   150744 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 1183 root      20   0  226712  37612  10500 S  0,3  5,0   0:44.05 Xorg
 4456 lenar     20   0  478056  34084  21944 S  0,3  4,5   0:00.18 xfce4-ter+
    1 root      20   0   33652   2516   1308 S  0,0  0,3   0:01.74 init
    2 root      20   0       0      0      0 S  0,0  0,0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S  0,0  0,0   0:00.05 ksoftirqd+
    5 root       0 -20       0      0      0 S  0,0  0,0   0:00.00 kworker/0+
    6 root      20   0       0      0      0 S  0,0  0,0   0:00.28 kworker/u+
    7 root      20   0       0      0      0 S  0,0  0,0   0:01.98 rcu_sched
    8 root      20   0       0      0      0 S  0,0  0,0   0:00.39 rcuos/0
    9 root      20   0       0      0      0 S  0,0  0,0   0:00.00 rcu_bh
   10 root      20   0       0      0      0 S  0,0  0,0   0:00.00 rcuob/0
   11 root      rt   0       0      0      0 S  0,0  0,0   0:00.00 migration+
   12 root      rt   0       0      0      0 S  0,0  0,0   0:00.02 watchdog/0
   13 root       0 -20       0      0      0 S  0,0  0,0   0:00.00 khelper
   14 root      20   0       0      0      0 S  0,0  0,0   0:00.00 kdevtmpfs
   15 root       0 -20       0      0      0 S  0,0  0,0   0:00.00 netns
   16 root      20   0       0      0      0 S  0,0  0,0   0:00.00 khungtaskd
```

```
lenar@lenar-VirtualBox:~$ who -a
          загрузка системы 2016-02-16 21:35
          уровень выполнения 2 2016-02-16 21:35
ВХОД     tty4         2016-02-16 21:35               636 id=4
ВХОД     tty5         2016-02-16 21:35               637 id=5
ВХОД     tty2         2016-02-16 21:35               642 id=2
ВХОД     tty3         2016-02-16 21:35               643 id=3
ВХОД     tty6         2016-02-16 21:35               646 id=6
ВХОД     tty1         2016-02-16 21:35              1017 id=1
lenar    ? :0         2016-02-16 21:35   ?          1275 (:0)
lenar    + pts/0      2016-02-16 22:21   .          4113 (:0.0)
          pts/6        2016-02-16 21:59                0 id=/6    терминал=0
 выход=0
lenar@lenar-VirtualBox:~$
```

```
lenar@lenar-VirtualBox:~$ ps -a
  PID TTY          TIME CMD
 1645 tty1     00:00:00 bash
 1979 pts/0    00:00:00 ps
lenar@lenar-VirtualBox:~$ df -h
Файл.система   Размер Использовано  Дост Использовано% Смонтировано в
/dev/sda1        7,3G         2,9G  4,1G           42% /
none             4,0K            0  4,0K            0% /sys/fs/cgroup
udev             236M         4,0K  236M            1% /dev
tmpfs             49M         896K   49M            2% /run
none             5,0M            0  5,0M            0% /run/lock
none             245M          76K  245M            1% /run/shm
none             100M          24K  100M            1% /run/user
lenar@lenar-VirtualBox:~$ uname -m
x86_64
lenar@lenar-VirtualBox:~$ uname -r
3.16.0-43-generic
```

```
lenar@lenar-VirtualBox:~$ kill -KILL 4947
lenar@lenar-VirtualBox:~$ top

top - 22:49:14 up  1:13,  5 users,  load average: 0,11, 0,20, 0,17
Tasks: 137 total,   3 running, 134 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0,7 us,  0,3 sy,  0,0 ni, 99,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 s
КиБ Mem:    759500 total,   569744 used,   189756 free,    29684 buffers
КиБ Swap:   784380 total,    18984 used,   765396 free.   140456 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 4572 root      20   0  215916  42672  17604 S   1,0  5,6   0:01.98 Xorg
 5000 lenar     20   0  478800  32972  21956 R   0,7  4,3   0:00.42 xfce4-ter+
 5080 lenar     20   0   22812   2944   2488 R   0,3  0,4   0:00.01 top
    1 root      20   0   33652   2512   1308 S   0,0  0,3   0:01.75 init
    2 root      20   0       0      0      0 S   0,0  0,0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S   0,0  0,0   0:00.06 ksoftirqd+
    5 root       0 -20       0      0      0 S   0,0  0,0   0:00.00 kworker/0+
    6 root      20   0       0      0      0 S   0,0  0,0   0:00.33 kworker/u+
    7 root      20   0       0      0      0 S   0,0  0,0   0:02.01 rcu_sched
    8 root      20   0       0      0      0 R   0,0  0,0   0:00.44 rcuos/0
    9 root      20   0       0      0      0 S   0,0  0,0   0:00.00 rcu_bh
   10 root      20   0       0      0      0 S   0,0  0,0   0:00.00 rcuob/0
   11 root      rt   0       0      0      0 S   0,0  0,0   0:00.00 migration+
   12 root      rt   0       0      0      0 S   0,0  0,0   0:00.02 watchdog/0
```

```
lenar@lenar-VirtualBox:~$ cat /home/lenar/file
text text text
lenar@lenar-VirtualBox:~$ chown test_user /home/lenar/file
chown: изменение владельца «/home/lenar/file»: Операция не позволена
lenar@lenar-VirtualBox:~$ sudo chown test_user /home/lenar/file
[sudo] password for lenar:
lenar@lenar-VirtualBox:~$ rm /home/lenar/file
lenar@lenar-VirtualBox:~$
```



4. Командный интерпретатор в среде UNIX выполняет две основные функции:

– представляет интерактивный интерфейс с пользователем, т. е. выдает приглашение и обрабатывает вводимые пользователем команды;

– обрабатывает и исполняет командные файлы.

Существует два основных типа оболочек UNIX: «Bourne shell» и «C shell». Bourne shell (или просто shell) использует командный синтаксис, похожий на первоначальный для UNIX. C shell используется иной синтаксис, чем-то напоминающий синтаксис языка программирования Си. В большинстве UNIX-систем он имеет имя /bin/csh.

В Linux используются две популярные вариации этих оболочек: Bourne shell (Bourne Again Shell) или «Bash» (/bin/bash) и Tcsh (/bin/tcsh). Bash — это развитие прежнего shell с добавлением многих полезных возможностей, частично содержащихся в C shell.

Поскольку Bash можно рассматривать как надмножество синтаксиса прежнего shell, любая программа, написанная на sh shell должна работать и в Bash. Tcsh является расширенной версией C shell.

Имя shell-переменной — это начинающаяся с буквы последовательность букв, цифр и подчеркиваний. Значение shell-переменной — строка символов.

Например: Var = «String» или Var = String

Команда echo $Var выведет на экран содержимое переменной Var, т. е. строку «String»; на то, что мы выводим содержимое переменной, указывает символ «$».

В UNIX-системах можно создавать командный файл, который содержит в себе набор команд интерпретатора shell. Командный файл в UNIX представляет собой обычный текстовый файл. Для создания командного файла используются

следующие команды: $ echo « « > commandfile — создание файла с названием commandfile, $ chmod +x commandfile — установка атрибута на исполнение данного файла.

5. Напишите свой собственный сценарий на языке shell с использованием изученных команд.

```
lenar@lenar-VirtualBox:~$ Today=`date`;
lenar@lenar-VirtualBox:~$ echo `date`;
Чт. февр. 18 15:49:42 SAMT 2016
lenar@lenar-VirtualBox:~$ echo "enter number:"; read x1; echo "you'r number:
$x1;
> "
enter number:
7423892
you'r number: 7423892;
```

6. Получите подробную информацию о файлах домашней директории.

```
lenar@lenar-VirtualBox:~$ ls -l
итого 44
-rw-rw-r-- 1 lenar lenar   69 февр. 17 22:13 1
-rw-rw-r-- 1 lenar lenar   69 февр. 17 22:17 file
-rw-rw-r-- 1 lenar lenar   69 февр. 17 22:24 txt
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Видео
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Документы
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Загрузки
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Изображения
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Музыка
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Общедоступные
drwxr-xr-x 2 lenar lenar 4096 февр. 17 17:47 Рабочий стол
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Шаблоны
```

7. Создайте новый файл и посмотрите на права доступа к нему, установленные системой при его создании.

```
lenar@lenar-VirtualBox:~$ cat>textfile
It is text file.
lenar@lenar-VirtualBox:~$ ls -l
итого 56
-rw-rw-r-- 1 lenar lenar   69 февр. 17 22:13 1
-rwxrwxr-x 1 lenar lenar   25 февр. 17 22:33 commandfile
-rw-rw-r-- 1 lenar lenar    7 февр. 17 22:41 f
-rw-rw-r-- 1 lenar lenar   69 февр. 17 22:17 file
-rw-rw-r-- 1 lenar lenar   17 февр. 17 22:52 textfile
-rw-rw-r-- 1 lenar lenar   69 февр. 17 22:24 txt
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Видео
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Документы
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Загрузки
drwxr-xr-x 2 lenar lenar 4096 февр. 17 15:33 Изображения
```

8. Убедитесь, что вы находитесь в своей домашней директории, и создайте новый текстовый файл. Введите туда информацию: ФИО студента, № группы. Скопируйте этот файл в другую директорию.

```
lenar@lenar-VirtualBox:~$ pwd;
/home/lenar
lenar@lenar-VirtualBox:~$ cat>Khaliullin
Халиуллин Ленар Рашитович
группа 8141-21
lenar@lenar-VirtualBox:~$ sudo cp /home/lenar/Khaliullin /Khaliullin
lenar@lenar-VirtualBox:~$ cat /Khaliullin
Халиуллин Ленар Рашитович
группа 8141-21
```

9. Реализуйте командный файл, который выводит: дату, системную информацию и текущего пользователя.

```
lenar@lenar-VirtualBox:~$ echo "date; finger lenar; who;">commandfile;
lenar@lenar-VirtualBox:~$ chmod +x commandfile;
lenar@lenar-VirtualBox:~$ ./commandfile
Чт. февр. 18 16:21:11 SAMT 2016
Login: lenar                            Name: Lenar
Directory: /home/lenar                  Shell: /bin/bash
On since Thu Feb 18 15:03 (SAMT) on :0 from :0 (messages off)
On since Thu Feb 18 15:04 (SAMT) on pts/9 from :0.0
   6 seconds idle
No mail.
No Plan.
lenar    :0           2016-02-18 15:03 (:0)
lenar    pts/9        2016-02-18 15:04 (:0.0)
```

Индивидуальное задание

Написать командный файл, который в цикле по нажатию клавиши выводит информацию о системе, активных пользователях в системе, а для введенного имени пользователя выводит список активных процессов данного пользователя.

```
script1.sh          [-M--] 11 L:[  1+ 0   1/ 23] *(11  / 477b) 0010 0x00A
# !bin/bash_
# script1.sh
while [ "$var1" != "0" ]
do
    echo "
=========================================================
Type 'sinfo' for display: System info, active users
Type user name or UID for display a list of processes that user
Type 'exit' for exit from program
========================================================="
    read var1

    if [ "$var1" == "sinfo" ]
    then
        uname -a
        who
    elif [ "$var1" == "exit" ]
    then
        exit
    else
        ps -U $var1
    fi
done
 1Помощь 2Сох~ть 3Блок  4Замена 5Копия  6Пер~ть 7Поиск  8Уда~ть 9МенюMC10Выход
```

```
[root@localhost newdir]# chmod +x ./script1.sh
[root@localhost newdir]# ./script1.sh

=====================================================
Type 'sinfo' for display: System info, active users
Type user name or UID for display a list of processes that user
Type 'exit' for exit from program
=====================================================
sinfo
Linux localhost.localdomain 2.6.32-642.6.1.el6.x86_64 #1 SMP Wed Oct 5 00:36:12
UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
root     tty1         2016-10-26 21:29
root     tty2         2016-10-26 21:31

=====================================================
Type 'sinfo' for display: System info, active users
Type user name or UID for display a list of processes that user
Type 'exit' for exit from program
=====================================================
_
```

```
=====================================================
sinfo
Linux localhost.localdomain 2.6.32-642.6.1.el6.x86_64 #1 SMP Wed Oct 5 00:36:12
UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
root     tty1         2016-10-26 21:29
root     tty2         2016-10-26 21:31

=====================================================
Type 'sinfo' for display: System info, active users
Type user name or UID for display a list of processes that user
Type 'exit' for exit from program
=====================================================
root
  PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 migration/0
    4 ?        00:00:00 ksoftirqd/0
    5 ?        00:00:00 stopper/0
    6 ?        00:00:00 watchdog/0
    7 ?        00:00:01 events/0
    8 ?        00:00:00 events/0
    9 ?        00:00:00 events_long/0
   10 ?        00:00:00 events_power_ef
   11 ?        00:00:00 cgroup
```

Лабораторная работа № 2

Цели и задачи

Знакомство с процессной организацией Unix-подобных систем. Изучение информационных команд отслеживания информации о процессах. Изучение различных типов процессов. Изучение информации о первичном процессе init и уровнях загрузки системы. Создание программы на языке Си, реализующей порождение и замещение процессов с использованием системных вызовов Unix, запуск команд Unix из пользовательской программы.

1. ps [опции].

Опции, отбирающие процессы для отчета:

-A : все процессы;

-a : связанные с конкретным терминалом, кроме главных системных процессов сеанса, часто используемая опция;

-N : отрицание выбора;

-d : все процессы, кроме главных системных процессов сеанса;

-e : все процессы;

g : процессы, связанные с данными идентификаторами групп;

-f : расширение информации;

T : все процессы на конкретном терминале;

a : процессы, связанные с текущим терминалом, а также процессы других пользователей;

r : информация только о работающих процессах;

x : процессы, отсоединенные от терминала.

```
lenar@lenar-VirtualBox:~$ ps g
  PID TTY      STAT   TIME COMMAND
 1801 pts/0    Ss     0:00 bash
 1822 pts/0    R+     0:00 ps g
lenar@lenar-VirtualBox:~$ ps -N
  PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    5 ?        00:00:00 kworker/0:0H
    6 ?        00:00:00 kworker/u2:0
    7 ?        00:00:01 rcu_sched
    8 ?        00:00:00 rcuos/0
    9 ?        00:00:00 rcu_bh
   10 ?        00:00:00 rcuob/0
```

```
lenar@lenar-VirtualBox:~$ ps -A
  PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    5 ?        00:00:00 kworker/0:0H
    6 ?        00:00:00 kworker/u2:0
    7 ?        00:00:01 rcu_sched
    8 ?        00:00:00 rcuos/0
    9 ?        00:00:00 rcu_bh
   10 ?        00:00:00 rcuob/0
   11 ?        00:00:00 migration/0
   12 ?        00:00:00 watchdog/0
   13 ?        00:00:00 khelper
   14 ?        00:00:00 kdevtmpfs
   15 ?        00:00:00 netns
   16 ?        00:00:00 khungtaskd
   17 ?        00:00:00 writeback
   18 ?        00:00:00 ksmd
   19 ?        00:00:00 crypto
   20 ?        00:00:00 kintegrityd
   21 ?        00:00:00 bioset
   22 ?        00:00:00 kblockd
```

```
lenar@lenar-VirtualBox:~$ ps -a
  PID TTY          TIME CMD
 1770 pts/0    00:00:00 ps
lenar@lenar-VirtualBox:~$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lenar     1756  1751  0 22:33 pts/0    00:00:00 bash
lenar     1771  1756  0 22:33 pts/0    00:00:00 ps -f
lenar@lenar-VirtualBox:~$ ps r
  PID TTY      STAT   TIME COMMAND
 1772 pts/0    R+     0:00 ps r
lenar@lenar-VirtualBox:~$ ps T
  PID TTY      STAT   TIME COMMAND
 1756 pts/0    Ss     0:00 bash
 1773 pts/0    R+     0:00 ps T
lenar@lenar-VirtualBox:~$ ps x
  PID TTY      STAT   TIME COMMAND
 1333 ?        Ss     0:00 init --user
 1386 ?        Ss     0:00 dbus-daemon --fork --session --address=unix:abstra
 1394 ?        Ss     0:00 upstart-event-bridge
 1400 ?        Sl     0:00 gnome-keyring-daemon --start --components ssh
 1404 ?        S      0:00 upstart-dbus-bridge --daemon --session --user --bu
 1406 ?        S      0:00 upstart-dbus-bridge --daemon --system --user --bus
 1408 ?        S      0:00 upstart-file-bridge --daemon --user
```

```
lenar@lenar-VirtualBox:~$ ps --deselect
  PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    5 ?        00:00:00 kworker/0:0H
    6 ?        00:00:00 kworker/u2:0
    7 ?        00:00:01 rcu_sched
    8 ?        00:00:00 rcuos/0
    9 ?        00:00:00 rcu_bh
   10 ?        00:00:00 rcuob/0
   11 ?        00:00:00 migration/0
   12 ?        00:00:00 watchdog/0
   13 ?        00:00:00 khelper
   14 ?        00:00:00 kdevtmpfs
   15 ?        00:00:00 netns
   16 ?        00:00:00 khungtaskd
   17 ?        00:00:00 writeback
   18 ?        00:00:00 ksmd
   19 ?        00:00:00 crypto
   20 ?        00:00:00 kintegrityd
   21 ?        00:00:00 bioset
```

/sbin/init 1 — Включает режим суперпользователя.

/sbin/init 3 — Перезагружает систему, выход из режима суперпользователя.

/sbin/init 5 — Возвращение в многопользовательский режим.

/sbin/init 6 — Перезагружает систему.

/sbin/init 0 — Выход из системы.

2. Для компиляции программ в Linux применяются компиляторы gcc и g++.

Исходные файлы должны иметь расширение *.c. Компилирование запускается командой **g++ имя_исходного_файла**.

После удачной компиляции будет создан исполняемый файл **a.out.** Запустить программу на исполнение можно, набрав имя исполняемого файла и нажав клавишу <Enter>.

3. Основные информационные команды и команды управления процессами.

Команда **ps** — предназначена для вывода информации о выполняемых процессах.

Команда **top** — предназначена для вывода информации о процессах в реальном времени. Процессы сортируются по максимальному занимаемому процессорному времени.

Команда **nice** — изменение приоритета процесса. Выполняет указанную команду с пониженным приоритетом, коэффициент понижения указывается в диапазоне 1–19 (по умолчанию он равен 10). Суперпользователь может повышать приоритет команды, для этого нужно указать отрицательный коэффициент, например –10.

Команда **kill** — принудительное завершение процесса. **kill** [-номер сигнала] PID, где PID — идентификатор процесса, который можно узнать с помощью команды **ps**.

4. PID и PPID для текущего процесса.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    pid_t pid, ppid;
    pid = getpid();
    ppid = getppid();
    printf("My pid = %d, my ppid = %d", (int)pid, (int)ppid);
    return 0;
}
```

Функция **wait** приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый «зомби» («zombie»)), то функция немедленно возвращается.

```
#include <iostream>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
#include <cstdlib>
using namespace std;
int main()
{
    pid_tpid; // прототип системного вызова
    pid = fork(); // создание дочернего процесса
    execlp("/bin/ps", "ps", NULL); // дочерний процесс выводит запущеные процессы
    wait(0); // процесс-родитель ждет завершения процесса-потомка
    execlp("who", "who" ,NULL); // процесс-родитель выводит имя пользователя
    exit(0);
}
return 0;
```

Прототипы функций

```
#include <unistd.h>
int execlp(const char *file, const char *arg0, ... const char *argN, (char *)NULL)
int execvp(const char *file, char *argv[])
int execl(const char *path, const char *arg0, ... const char *argN, (char *)NULL)
int execv(const char *path, char *argv[])
int execle(const char *path, const char *arg0, ... const char *argN,(char *)NULL, char * envp[])
int execve(const char *path, char *argv[], char *envp[])
```

Описание функций

Аргумент **file** является указателем на имя файла, который должен быть загружен. Аргумент **path —** это указатель на полный путь к файлу, который должен быть загружен.

Аргументы **arg0, ..., argN** представляют собой указатели на аргументы командной строки.

Заметим, что аргумент **arg0** должен указывать на имя загружаемого файла. Аргумент **argv** представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель **NULL.**

Аргумент **envp** является массивом указателей на параметры окружающей среды, заданные в виде строк «переменная=строка». Последний элемент этого массива должен содержать указатель **NULL.**

Аргумент **envp** является массивом указателей на параметры окружающей среды, заданные в виде строк «переменная=строка». Последний элемент этого массива должен содержать указатель **NULL.**

Индивидуальное задание

Написать командный файл, который в цикле по нажатию клавиши выводит информацию о системе, активных пользователях в системе, а для введенного имени пользователя выводит список активных процессов данного пользователя.

Данное задание совпадает с индивидуальным заданием из лабораторной работы № 1.

Лабораторная работа № 3

Цели и задачи

Изучить переназначение операций ввода-вывода. Научиться использовать каналы для организации взаимодействия процессов и их синхронизации.

1. Неименованный канал является средством взаимодействия между связанными процессами — родительским и дочерним. Родительский процесс создает канал при помощи системного вызова.

Прототип системного вызова

```
#include <unistd.h>
int pipe(int *fd);
```

Параметр fd является указателем на массив из двух целых переменных. При нормальном завершении вызова в первый элемент массива — fd[0] — будет занесен файловый дескриптор, соответствующий выходному потоку данных pipe и позволяющий выполнять только операцию чтения, а во второй элемент массива — fd[1] — будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи.

В процессе работы системный вызов организует выделение области памяти под буфер и указатели и заносит информацию, соответствующую входному и выходному потокам данных, в два элемента таблицы открытых файлов, связывая тем самым с каждым pip'ом два файловых дескриптора. Для одного из них разрешена только операция чтения из pip'а, а для другого — только опера-

ция записи в pipe. Для выполнения этих операций мы можем использовать те же самые системные вызовы read() и write(), что и при работе с файлами.

```
lenar@lenar-VirtualBox:~$ g++ /home/lenar/3.1.c
lenar@lenar-VirtualBox:~$ ./a.out
Hello, world!
lenar@lenar-VirtualBox:~$ g++ /home/lenar/3.2.c
lenar@lenar-VirtualBox:~$ ./a.out
I am child, MyPid--1942

 I am Parent, I read your pid- 'CHILD_Pid=1942'
lenar@lenar-VirtualBox:~$ g++ /home/lenar/3.3.c
lenar@lenar-VirtualBox:~$ ./a.out
 Can't fork child
```

Результаты выполнения программ 3.1, 3.2, 3.3.

2. Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи — FIFO или именованный pipe. FIFO подобен pip'у, за одним исключением того, что данные о расположении FIFO в адресном пространстве ядра и его состоянии процессы можно получать не через родственные связи, а через файловую систему. Для создания FIFO используется системный вызов mknod() или существующая в некоторых версиях UNIX функция mkfifo().

Прототип системного вызова

```
#include <sys/stat.h>
#include <unistd.h>
int mknod(char *path, int mode, int dev);
```

Параметр dev является несущественным, и мы задаем его равным 0. Параметр path является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом существовать не должно. Параметр mode устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как результат побитовой операции «или» значения S_IFIFO, указывающего, что системный вызов должен создать FIFO, и некоторой суммы следующих восьмеричных значений:

0400 — разрешено чтение для пользователя, создавшего FIFO;

0200 — разрешена запись для пользователя, создавшего FIFO;

0040 — разрешено чтение для группы пользователя, создавшего FIFO;

0020 — разрешена запись для группы пользователя, создавшего FIFO;

0004 — разрешено чтение для всех остальных пользователей;

0002 — разрешена запись для всех остальных пользователей.

При успешном создании FIFO системный вызов возвращает значение 0, при неуспешном — отрицательное значение.

Функция mkfifo

Прототип функции

```
#include <sys/stat.h>
#include <unistd.h>
int mkfifo(char *path, int mode);
```

Функция mkfifo предназначена для создания FIFO в операционной системе. Параметр path является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом функции не должно существовать. Параметр mode устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как некоторая сумма следующих восьмеричных значений:

0400 — разрешено чтение для пользователя, создавшего FIFO;

0200 — разрешена запись для пользователя, создавшего FIFO;

0040 — разрешено чтение для группы пользователя, создавшего FIFO;

0020 — разрешена запись для группы пользователя, создавшего FIFO;

0004 — разрешено чтение для всех остальных пользователей;

0002 — разрешена запись для всех остальных пользователей.

При успешном создании FIFO функция возвращает значение 0, при неуспешном — отрицательное значение.

3. Для дальнейшей работы с FIFO (pipe) применяются системные вызовы read(), write(), open() и close().

Системный вызов open(). Прежде чем совершать операции чтения данных из файла и записи их в файл, необходимо поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом open().

Прототип системного вызова

```
#include <fcntl.h>
int open(char *path, int flags);
int open(char *path, int flags, int mode);
```

Системный вызов open предназначен для выполнения операции открытия файла и, в случае ее удачного осуществления, возвращает файловый дескриптор открытого файла. Параметр path является указателем на строку, содержащую полное или относительное имя файла. Параметр flags может принимать одно из следующих трех значений:

O_RDONLY — если над файлом в дальнейшем будут совершаться только операции чтения;

O_WRONLY — если над файлом в дальнейшем будут осуществляться только операции записи;

O_RDWR — если над файлом будут осуществляться и операции чтения, и операции записи.

Системные вызовы read() и write(). Для совершения потоковых операций чтения информации из файла и ее записи в файл применяются системные вызовы read() и write().

Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
size_t read(int fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr, size_t nbytes);
```

Системные вызовы read и write предназначены для осуществления потоковых операций ввода и вывода информации над каналами связи, описываемыми файловыми дескрипторами, т. е. для файлов, pipe и FIFO. Параметр fd является файловым дескриптором созданного ранее потокового канала связи, через который будет отсылаться или получаться информация, т. е. значением, которое вернул один из системных вызовов open() или pipe(). Параметр addr представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация. Параметр nbytes для системного вызова write определяет количество байт, которое должно быть передано, начиная с адреса памяти addr. Параметр nbytes для системного вызова read определяет количество байт, которое мы хотим получить из канала связи и разместить в памяти, начиная с адреса addr.

В случае успешного завершения системный вызов возвращает количество реально посланных или принятых байт. Это значение может не совпадать с заданным значением параметра nbytes, а быть меньше, чем оно. При возникновении какой-либо ошибки возвращается отрицательное значение.

Системный вызов close()

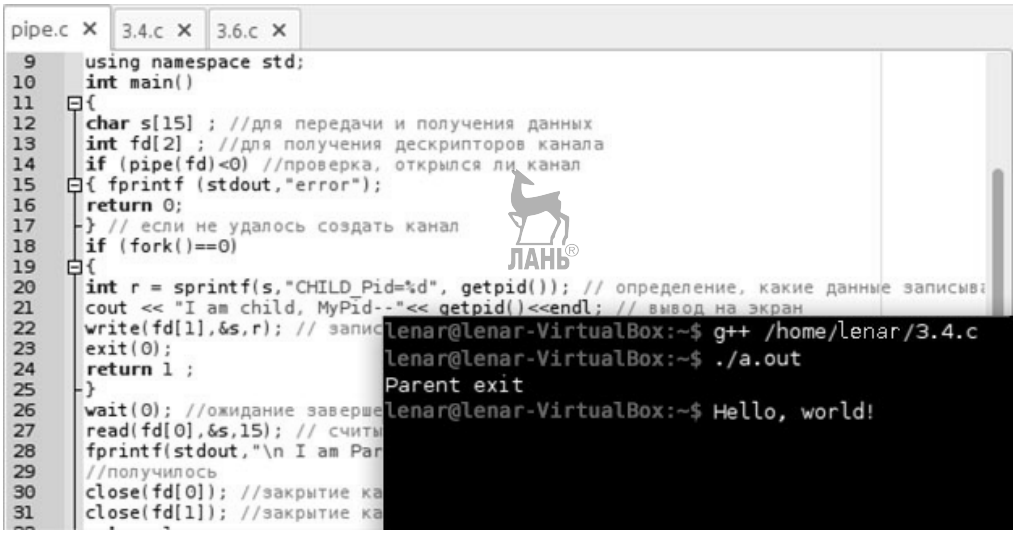
Прототип системного вызова

```
#include <unistd.h>
int close(int fd);
```

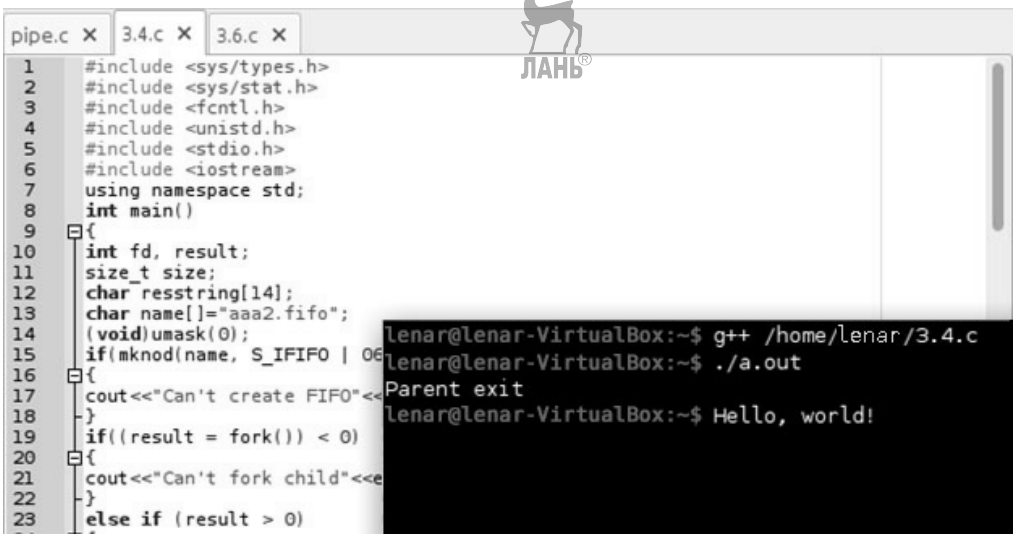
Системный вызов close предназначен для корректного завершения работы с файлами и другими объектами ввода-вывода, которые описываются в операционной системе через файловые дескрипторы: pipe и FIFO. Параметр fd является дескриптором соответствующего объекта, т. е. значением, которое вернул один из системных вызовов open() или pipe().

Системный вызов возвращает значение 0 при нормальном завершении и значение –1 при возникновении ошибки.

4. Системный вызов pipe.



5. Именованные каналы.



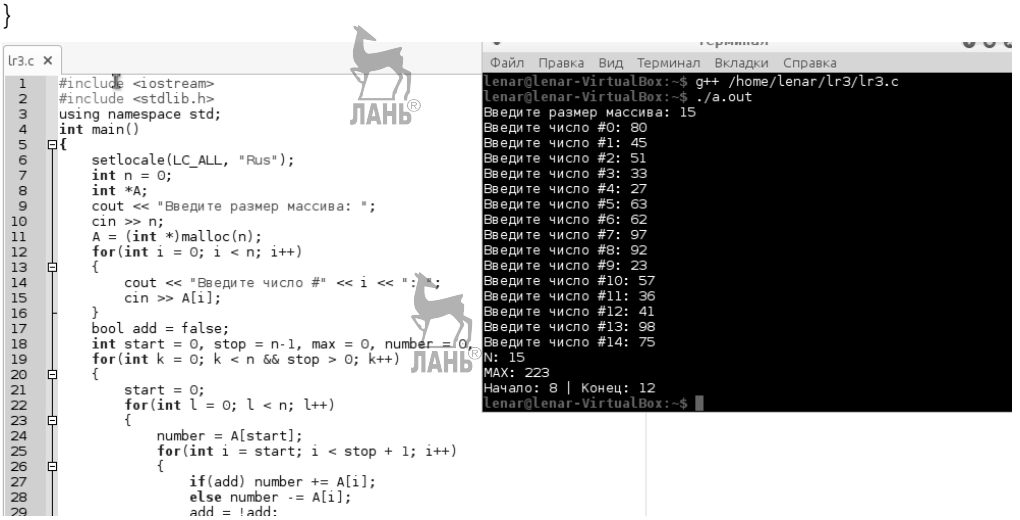
Индивидуальное задание

А. Найти индексы i и j, для которых существует наибольшая последовательность a[i] – a[i+1] + a[i+2] – a[i+3] … +/– a[j]. Входные данные: целое положительное число n, массив чисел A размерности n.

Исходный код программы

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{
setlocale(LC_ALL, «Rus»);
     int n = 0;
     int *A;
     cout << «Введите размер массива: «;
     cin >> n;
     A = (int *)malloc(n);
     for(int i = 0; i < n; i++)
     {
          cout << «Введите число #» << i << «: «;
          cin >> A[i];
     }
     bool add = false;
     int start = 0, stop = n-1, max = 0, number = 0, f_start = 0,
     f_stop = 0;
     for(int k = 0; k < n && stop > 0; k++)
     {
          start = 0;
          for(int l = 0; l < n; l++)
          {
               number = A[start];
               for(int i = start; i < stop + 1; i++)
               {
```

```
                    if(add) number += A[i];
                    else number -= A[i];
                    add = !add;
               }
               if(number > max)
               {
                    max = number;
                    f_start = start;
                    f_stop = stop;
               }
               start++;
          }
          stop--;
     }
     cout << «N: « << n << endl;
     cout << «MAX: « << max << endl;
     cout << «Начало: « << f_start << « | Конец: « << f_stop <<
endl;
     return 0;
}
```



Результат выполнения программы.

В. Первое приложение ожидает ввода чисел a, b, c и отсылает их второму приложению, которое находит решение уравнения ax^2+bx+c=0, и отсылает результат первому приложению.

Исходный код 1-й программы

```
#include <iostream>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
using namespace std;
int main()
{
     int kx[3] = { 0, 0, 0 }, result = 0, count = 0, fifo;
     setlocale(LC_ALL, «Rus»);
```

```
    if(mkfifo(«outchange.fifo», S_IFIFO | 0666) < 0)
    {
        cout << «Ошибка при создании FIFO» << endl;
        return 0;
    }
    if((fifo = open(«outchange.fifo», O_WRONLY)) < 0)
    {
        cout << «Ошибка открытия FIFO» << endl;
        return 0;
    }
    for(int i = 0; i < 3; i++)
    {
        cout << «Введите « << i+1 << «-й коэффициент: «;
        cin >> kx[i];
        if(write(fifo, &kx[i], 1) != 1)
        {
            cout << «Ошибка записи в FIFO» << endl;
            return 0;
        }
    }
    close(fifo);
    while((fifo = open(«inchange.fifo», O_RDONLY)) < 0) {}
    // ожидание получения ответа
    cout << «Результат решения ax^2+bx+c=0» << endl;
    while(count < 3)
    {
        while(read(fifo, &result, 1) < 0) {}
        if(count == 0)
        {
            cout << «D = « << result << endl;
            if(result < 0)
            {
                cout << «Уравнение ax^2+bx+c=0 не имеет
                корней» << endl;
                break;
            }
        }
        else cout << «X» << count << « = « << result << endl;
        count++;
    }
    close(fifo);
    return 0;
}
```

Исходный код 2-й программы

```
#include <iostream>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
using namespace std;
int main()
{
```

```
    int kx[3] = { 0, 0, 0 }, x[2] = { 0, 0 }, d = 0, fifo,
    count = 0, wrs;
    setlocale(LC_ALL, «Rus»);
    while ((fifo = open(«outchange.fifo», O_RDONLY)) < 0) {}
    while(count < 3)
    {
        read(fifo, &kx[count], 1);
        cout << «Получен « << count+1 << «-й коэффициент:
        « << kx[count] << endl;
        count++;
    }
    close(fifo);
    d = pow(kx[1], 2) - 4 * kx[0] * kx[2]; // D
    x[0] = (-kx[1] + sqrt(d)) / 2; // X1
    x[1] = (-kx[1] + sqrt(d)) / 2; // X2
    if(mkfifo(«inchange.fifo», S_IFIFO | 0666) < 0)
    {
        cout << «Ошибка при создании FIFO» << endl;
        return 0;
    }
    if((fifo = open(«inchange.fifo», O_WRONLY)) < 0)
    {
        cout << «Ошибка открытия FIFO» << endl;
        return 0;
    }
    for(int i = 0; i < 3; i++)
    {
        if(i == 0) wrs = write(fifo, &d, 1);
        else if(i == 1) wrs = write(fifo, &x[0], 2);
        else if(i == 2) wrs = write(fifo, &x[1], 2);
        if(wrs < 1)
        {
            cout << «Ошибка записи в FIFO» << endl;
            return 0;
        }
    }
    close(fifo);
    return 0;
}
```

Результат выполнения

3b1.c
```
#include <iostream>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

using namespace std;

int main()
{
    int kx[3] = { 0, 0, 0 }, result = 0, count = 0, fifo;
    setlocale(LC_ALL, "Rus");
    if(mkfifo("outchange.fifo", S_IFIFO | 0666) < 0)
    {
        cout << "Ошибка при создании FIFO" << endl;
        return 0;
    }
    if((fifo = open("outchange.fifo", O_WRONLY)) < 0)
    {
        cout << "Ошибка открытия FIFO" << endl;
        return 0;
    }

    for(int i = 0; i < 3; i++)
    {
        cout << "Введите " << i+1 << "-й коэффициент: ";
        cin >> kx[i];
        if(write(fifo, &kx[i], 1) != 1)
        {
            cout << "Ошибка записи в FIFO" << endl;
            return 0;
        }
    }
    close(fifo);
    while((fifo = open("inchange.fifo", O_RDONLY)) < 0) {} // o
    cout << "Результат решения ax^2+bx+c=0" << endl;

    while(count < 3)
    {
        while(read(fifo, &result, 1) < 0) {}
        if(count == 0)
        {
            cout << "D = " << result << endl;
            if(result < 0)
            {
                cout << "Уравнение ax^2+bx+c=0 не имеет корней"
                break;
            }
        }
        else cout << "X" << count << " = " << result << endl;
        count++;
    }
    close(fifo);
    return 0;
}
```

Терминал

```
Файл  Правка  Вид  Терминал  Вкладки  Справка
lenar@lenar-VirtualBox:~$ g++ 3b2.c -o 3b2.out
lenar@lenar-VirtualBox:~$ ./3b2.out
Получен 1-й коэффициент: 1
Получен 2-й коэффициент: -2
Получен 3-й коэффициент: -3
lenar@lenar-VirtualBox:~$
```

```
lenar@lenar-VirtualBox:~$ g++ 3b1.c -o 3b1.out
lenar@lenar-VirtualBox:~$ ./3b1.out
Введите 1-й коэффициент: 1
Введите 1-й коэффициент: -2
Введите 1-й коэффициент: -3
Результат решения ax^2+bx+c=0
D = 16
X1 = 3
X2 = -1
lenar@lenar-VirtualBox:~$
```

3b2.c
```
#include <iostream>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>

using namespace std;

int main()
{
    int kx[3] = { 0, 0, 0 }, x[2] = { 0, 0 }, d = 0, fifo, count
    setlocale(LC_ALL, "Rus");
    while ((fifo = open("outchange.fifo", O_RDONLY)) < 0) {}

    while(count < 3)
    {
        read(fifo, &kx[count], 1);
        cout << "Получен " << count+1 << "-й коэффициент: " << k
        count++;
    }
    close(fifo);

    d = pow(kx[1], 2) - 4 * kx[0] * kx[2]; // D
    x[0] = (-kx[1] + sqrt(d)) / 2; // X1
    x[1] = (-kx[1] + sqrt(d)) / 2; // X2

    if(mkfifo("inchange.fifo", S_IFIFO | 0666) < 0)
    {
        cout << "Ошибка при создании FIFO" << endl;
        return 0;
    }
    if((fifo = open("inchange.fifo", O_WRONLY)) < 0)
    {
        cout << "Ошибка открытия FIFO" << endl;
        return 0;
    }
    for(int i = 0; i < 3; i++)
    {
        if(i == 0) wrs = write(fifo, &d, 1);
        else if(i == 1) wrs = write(fifo, &x[0], 2);
        else if(i == 2) wrs = write(fifo, &x[1], 2);
        if(wrs < 1)
        {
            cout << "Ошибка записи в FIFO" << endl;
            return 0;
        }
    }
    close(fifo);
    return 0;
}
```

Терминал

```
Файл  Правка  Вид  Терминал  Вкладки  Справка
lenar@lenar-VirtualBox:~$ g++ 3b2.c -o 3b2.out
lenar@lenar-VirtualBox:~$ ./3b2.out
Получен 1-й коэффициент: 1
Получен 2-й коэффициент: -2
Получен 3-й коэффициент: -3
lenar@lenar-VirtualBox:~$
```

```
lenar@lenar-VirtualBox:~$ g++ 3b1.c -o 3b1.out
lenar@lenar-VirtualBox:~$ ./3b1.out
Введите 1-й коэффициент: 1
Введите 1-й коэффициент: -2
Введите 1-й коэффициент: -3
Результат решения ax^2+bx+c=0
D = 16
X1 = 3
X2 = -1
lenar@lenar-VirtualBox:~$
```

Лабораторная работа № 4

Цели и задачи

Изучить общие принципы работы с основными средствами межпроцессной коммуникации. Научиться создавать многопроцессные алгоритмы с общей разделяемой памятью. Познакомиться с легковесными процессами — потоками.

Средствами межпроцессной коммуникации (IPC — Inter-Process Communication) являются сигналы, каналы, сообщения, семафоры, разделяемая память и сокеты. Процессы выполняются в собственном адресном пространстве, они изолированы друг от друга, поэтому необходимы механизмы для взаимодействия процессов, предоставляемые самой операционной системой и, как правило, расположенные в адресном пространстве системы.

Коммуникация между процессами необходима для решения следующих задач.

1. Передача данных от одного процесса к другому.
2. Совместное использование общих данных несколькими процессами.
3. Синхронизация работы процессов.

Функция для генерации ключа System V IPC

Прототип функции

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj);
```

Описание функции

Функция **ftok** служит для преобразования имени существующего файла и небольшого целого числа, например, порядкового номера экземпляра средств связи, в ключ System V IPC.

Параметр **pathname** должен являться указателем на имя существующего файла, доступного для процесса, вызывающего функцию.

Параметр **proj** — это небольшое целое число, характеризующее экземпляр средства связи.

В случае невозможности генерации ключа функция возвращает отрицательное значение, в противном случае она возвращает значение сгенерированного ключа. Тип данных **key_t** обычно представляет собой 32-битовое целое.

Системный вызов shmget()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

Описание системного вызова

Системный вызов shmget предназначен для выполнения операции доступа к сегменту разделяемой памяти и, в случае его успешного завершения, возвращает дескриптор System V IPC для этого сегмента (целое неотрицательное

число, однозначно характеризующее сегмент внутри вычислительной системы и использующееся в дальнейшем для других операций с ним).

Параметр key является ключом System V IPC для сегмента, т. е. фактически его именем из пространства имен System V IPC. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции ftok(), или специальное значение IPC_PRIVATE. Использование значения IPC_PRIVATE всегда приводит к попытке создания нового сегмента разделяемой памяти с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции ftok() ни при одной комбинации ее параметров.

Параметр size определяет размер создаваемого или уже существующего сегмента в байтах. Если сегмент с указанным ключом уже существует, но его размер не совпадает с указанным в параметре size, констатируется возникновение ошибки.

Параметр shmflg — флаги — играет роль только при создании нового сегмента разделяемой памяти и определяет права различных пользователей при доступе к сегменту, а также необходимость создания нового сегмента и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или — « | ») следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT — если сегмента для указанного ключа не существует, он должен быть создан;

IPC_EXCL — применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании сегмента с указанным ключом, доступ к сегменту не производится и констатируется ошибочная ситуация, при этом переменная errno, описанная в файле <errno.h>, примет значение EEXIST;

0400 — разрешено чтение для пользователя, создавшего сегмент;

0200 — разрешена запись для пользователя, создавшего сегмент;

0040 — разрешено чтение для группы пользователя, создавшего сегмент;

0020 — разрешена запись для группы пользователя, создавшего сегмент;

0004 — разрешено чтение для всех остальных пользователей;

0002 — разрешена запись для всех остальных пользователей.

Возвращаемое значение

Системный вызов возвращает значение дескриптора System V IPC для сегмента разделяемой памяти при нормальном завершении и значение –1 при возникновении ошибки.

Доступ к созданной области разделяемой памяти в дальнейшем обеспечивается ее дескриптором, который вернет системный вызов shmget(). Доступ к уже существующей области также может осуществляться двумя способами: если знать ее ключ, то, используя вызов shmget(), можно получить ее дескриптор. В этом случае нельзя указывать в качестве составной части флагов флаг IPC_EXCL, а значение ключа, естественно, не может быть IPC_PRIVATE. Права доступа игнорируются, а размер области должен совпадать с размером, указанным при ее создании; либо можно воспользоваться тем, что дескриптор System V IPC действителен в рамках всей операционной системы, и передать

его значение от процесса, создавшего разделяемую память, текущему процессу. При создании разделяемой памяти с помощью значения IPC_PRIVATE — это единственно возможный способ.

После получения дескриптора необходимо включить область разделяемой памяти в адресное пространство текущего процесса. Это осуществляется с помощью системного вызова shmat().

При нормальном завершении он вернет адрес разделяемой памяти в адресном пространстве текущего процесса. Дальнейший доступ к этой памяти осуществляется с помощью обычных средств языка программирования.

Системный вызов shmat()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int shmflg);
```

Описание системного вызова

Системный вызов shmat предназначен для включения области разделяемой памяти в адресное пространство текущего процесса. Данное описание не является полным описанием системного вызова. Для полного описания обращайтесь к UNIX Manual.

Параметр shmid является дескриптором System V IPC для сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов shmget() при создании сегмента или при его поиске по ключу.

В качестве параметра shmaddr в рамках нашего курса мы всегда будем передавать значение NULL, позволяя операционной системе самой разместить разделяемую память в адресном пространстве нашего процесса.

Параметр shmflg в нашем курсе может принимать только два значения: 0 — для осуществления операций чтения и записи над сегментом и SHM_RDONLY — если мы хотим только читать из него. При этом процесс должен иметь соответствующие права доступа к сегменту.

Возвращаемое значение

Системный вызов возвращает адрес сегмента разделяемой памяти в адресном пространстве процесса при нормальном завершении и значение –1 при возникновении ошибки.

После окончания использования разделяемой памяти процесс может уменьшить размер своего адресного пространства, исключив из него эту область с помощью системного вызова shmdt().

В качестве параметра системный вызов shmdt() требует адрес начала области разделяемой памяти в адресном пространстве процесса, т. е. значение, которое вернул системный вызов shmat(), поэтому данное значение следует сохранять на протяжении всего времени использования разделяемой памяти.

Системный вызов shmdt()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

Описание системного вызова

Системный вызов shmdt предназначен для исключения области разделяемой памяти из адресного пространства текущего процесса.

Параметр shmaddr является адресом сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов shmat().

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение –1 при возникновении ошибки.

Системный вызов shmctl()

Прототип системного вызова

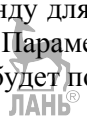

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Описание системного вызова

Системный вызов shmctl предназначен для получения информации об области разделяемой памяти, изменения ее атрибутов и удаления из системы. Данное описание не является полным описанием системного вызова. Для изучения полного описания обращайтесь к UNIX Manual.

В данном курсе системный вызов shmctl будет использован только для удаления области разделяемой памяти из системы. Параметр shmid является дескриптором System V IPC для сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов shmget() при создании сегмента или при его поиске по ключу.

В качестве параметра cmd в рамках данного курса всегда будет передаваться значение IPC_RMID — команду для удаления сегмента разделяемой памяти с заданным идентификатором. Параметр buf для этой команды не используется, поэтому всегда необходимо будет подставлять туда значение NULL.



Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение –1 при возникновении ошибки.

Функция pthread_self()

Прототип функции

```
#include <pthread.h>
pthread t pthread self(void);
```

Описание функции

Функция pthread_self возвращает идентификатор текущей нити исполнения.

Тип данных pthread_t является синонимом для одного из целочисленных типов языка Си.

Функция для создания нити исполнения

Прототип функции

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
void * (*start_routine)(void *), void *arg);
```

Описание функции

Функция pthread_create служит для создания новой нити исполнения (thread'a) внутри текущего процесса. Настоящее описание не является полным описанием функции. Для изучения полного описания обращайтесь к UNIX Manual.

Новый thread будет выполнять функцию start_routine с прототипом void *start_routine(void *), передавая ей в качестве аргумента параметр arg. Если требуется передать более одного параметра, они собираются в структуру; передается адрес этой структуры. Значение, возвращаемое функцией start_routine, не должно указывать на динамический объект данного thread'a.

Параметр attr служит для задания различных атрибутов создаваемого thread'a.

Функция для завершения нити исполнения

Прототип функции

```
#include <pthread.h>
void pthread_exit(void *status);
```

Описание функции

Функция pthread_exit служит для завершения нити исполнения (thread) текущего процесса. Функция никогда не возвращается в вызвавший ее thread. Объект, на который указывает параметр status, может быть впоследствии изучен в другой нити исполнения, например, в нити, породившей завершившуюся нить. Поэтому он не должен указывать на динамический объект завершившегося thread'a.

Одним из вариантов получения адреса, возвращаемого завершившимся thread'ом, с одновременным ожиданием его завершения является использование функции pthread_join().

Нить исполнения, вызвавшая эту функцию, переходит в состояние «ожидание» до завершения заданного thread'a.

Функция pthread_join()

Прототип функции

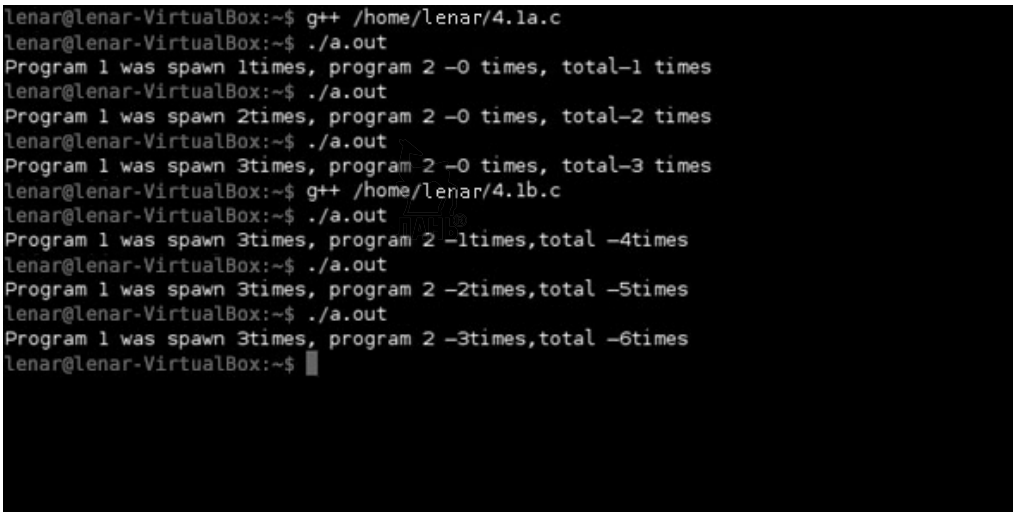
```
#include <pthread.h>
int pthread_join (pthread_t thread, void **status_addr);
```

Описание функции

Функция pthread_join блокирует работу вызвавшей ее нити исполнения до завершения thread'a с идентификатором thread. После разблокирования в указатель, расположенный по адресу status_addr, заносится адрес, который вернул

завершившийся thread либо при выходе из ассоциированной с ним функции, либо при выполнении функции pthread_exit().

Ход работы представлен на рисунке



Результат выполнения программ 4.1-a, 4.2-b.

Эти программы используют разделяемую память для хранения числа запусков каждой из программ и их суммы. В разделяемой памяти размещается массив из трех целых чисел. Первый элемент массива используется как счетчик для программы 1, второй элемент — для программы 2, третий элемент — для обеих программ суммарно.

6. Программы, осуществляющие взаимодействие через разделяемую память.

Листинг 2.a.c:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL,»Rus»);
char string[14]=«Hello, world!»; //Строка, которая будет присвоена
массиву.
int *array; /* Указатель на разделяемую память */
int shmid; /* IPC-дескриптор для области разделяемой памяти */
char pathname[] = «i»; /* Имя файла, используемое для генерации
ключа. Файл с таким именем должен существовать в текущей директо-
рии */
key_t key; /* IPC ключ */
```

```
/* Генерирование IPC ключа из имени файла для генерации ключа в
текущей директории и номера экземпляра области разделяемой памяти
0 */
if((key = ftok(pathname,0)) < 0)
{
cout<<«Ключ не создан»<<endl;
}
/* Эксклюзивное создание разделяемой памяти для сгенерированного
ключа, т. е. если для этого ключа она уже существует, системный
вызов вернет отрицательное значение. Размер памяти определяем, как
14 байт, права доступа 0666 — чтение и запись разрешены для всех
*/
if((shmid = shmget(key, 14, 0666|IPC_CREAT|IPC_EXCL)) < 0)
{
/* В случае ошибки необходимо определить: возникла ли она из-за
того, что сегмент разделяемой памяти уже существует, или по другой
причине */
if(errno != EEXIST)
{
/* Если по другой причине — прекращение работы */
cout<<«Can't create shared memory»<<endl;
}
else
{
/* Если из-за того, что разделяемая память уже существует, то не-
обходимо получить ее IPC-дескриптор и, в случае удачи, сброс флага
необходимости инициализации элементов массива */
if((shmid = shmget(key, 14, 0)) < 0)
{
cout<<«Can't find shared memory»<<endl;
}
}
}
/*Отображение разделяемой памяти в адресное пространство текущего
процесса. Обратите внимание на то, что для правильного сравнения
мы явно преобразовываем значение -1 к указателю на целое.*/
if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1))
{
cout<<«Can't attach shared memory»<<endl;
}
/*Отображение разделяемой памяти в адресное пространство текущего
процесса..*/
for (int r=0;r<14;r++)
{
array[r]=int(string[r]);
}
if(shmdt(array) < 0)
{
cout<<«Ошибка «<<endl;
}
return 0;
}
```

Листинг 2.b.c:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
#include <iostream>
using namespace std;
int main()
{
int *array; /* Указатель на разделяемую память */
int shmid; /* IPC-дескриптор для области разделяемой памяти */
char pathname[] = «i»; /* Имя файла, используемое для генерации
ключа. Файл с таким именем должен существовать в текущей директо-
рии */
key_t key; /* IPC ключ */
/* Генерирование IPC ключа из имени файла для генерации ключа в
текущей директории и номера экземпляра области разделяемой памяти
0 */
if((key = ftok(pathname,0)) < 0)
{
cout<<«Can't generate key»<<endl;
}
/* Эксклюзивное создание разделяемой памяти для сгенерированного
ключа, т. е. если для этого ключа она уже существует, системный
вызов вернет отрицательное значение. Размер памяти определяем как
14 байт, права доступа 0666 — чтение и запись разрешены для всех
*/
if((shmid = shmget(key, 14, 0666|IPC_CREAT|IPC_EXCL)) < 0)
{
/* В случае ошибки необходимо определить: возникла ли она из-за
того, что сегмент разделяемой памяти уже существует, или по другой
причине */
if(errno != EEXIST)
{
/* Если по другой причине — прекращение работы */
cout<<«Can't create shared memory»<<endl;
}
else
{
/* Если из-за того, что разделяемая память уже существует, то не-
обходимо получить ее IPC-дескриптор и, в случае удачи, сброс флага
необходимости инициализации элементов массива */
if((shmid = shmget(key, 14, 0)) < 0)
{
cout<<«Can't find shared memory»<<endl;
}
}
}
/*Отображение разделяемой памяти в адресное пространство текущего
процесса.*/
if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1))
{
```

```
cout<<«Can't attach shared memory»<<endl;
}
for (int r=0;r<14;r++)
{
cout<<char(array[r]);}
/* Вывод строки, удаление разделяемой памяти из адресного про-
странства текущего процесса и завершение работы */
if(shmdt(array) < 0)
return 0;
}
```

```
lenar@lenar-VirtualBox:~$ g++ /home/lenar/2a.c
lenar@lenar-VirtualBox:~$ ./a.out
lenar@lenar-VirtualBox:~$ g++ /home/lenar/2b.c
lenar@lenar-VirtualBox:~$ ./a.out
Hello, world!
```

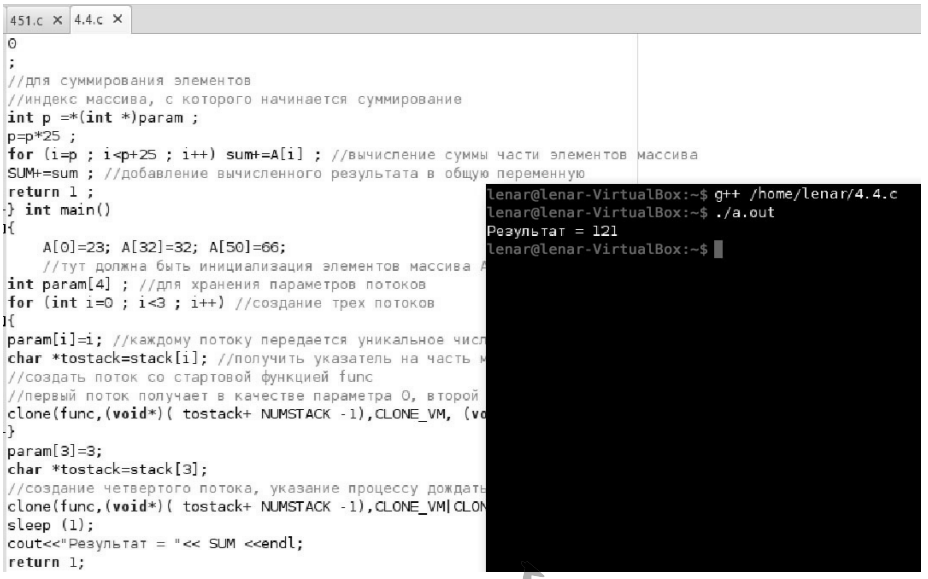
Результат выполения программ 2.a.c, 2.b.c.

7.

```
4.1a.c × 4.1b.c × 2a.c × 2b.c × 4.02.c ×
13    *mem_sum;
14    int main()
15   { int shmid = shmget(IPC_PRIVATE, 2, IPC_CREAT|0666) ;
16    if (shmid < 0 )
17   {
18    cout<<"error"<<endl;
19    return 0 ;
20    }
21    mem_sum = (mymem *)shmat(shmid,NULL,0) ;
22    A[2]=5; //инициализация массива
23    A[56]=90; //взял три числа в разных частях массива
24    A[4]=6; //в сумме должно дать 101
25    int pid, sum=0 ;
26    pid = fork() ;//Порождение
27    if ( pid == 0 )
28   { for (
29    int i=0
30    ;
31    i<50 ;
32    i++) sum+=A[i] ;
33    mem_sum->sum=sum ;
34    } if (
35    pid != 0
36    )
37   { for (
38    int i=50 ;
39    i<100 ;
40    i++) sum+=A[i] ;
41    wait(NULL);
42    cout<<"Calculate = " << sum+mem_sum->sum<<endl;
43    }return 1;
```

```
lenar@lenar-VirtualBox:~$ g++ /home/lenar/4.02.c
lenar@lenar-VirtualBox:~$ ./a.out
Calculate = 101
lenar@lenar-VirtualBox:~$
```

Результат выполнения программы 4.02.



Результат выполнения программы 4.04.

Индивидуальное задание

А. Определить, является ли матрица А симметричной относительно главной диагонали. Входные данные: целое положительное число n, массив чисел А размерности nxn.

Исходный код программы

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{
     setlocale(LC_ALL, «Rus»);
     int N;
     bool symmetric = true;
     cout << «Введите размерность матрицы: «;
     cin >> N; // Получение размера матрицы
     int **A = new int*[N]; // Объявление двумерного массива
     размерности N*N с с выделением памяти
     for(int i = 0; i < N; i++) A[i] = new int[N]; // Выделение
     памяти каждому внутреннему массиву
     for(int i = 0; i < N; i++)
     {
          for(int j = 0; j < N; j++)
          {
               cout << «Введите значение
               A[« << i+1 << «][« << j+1 << «]: «;
               cin >> A[i][j];
          }
     }
     cout << «Внесенный массив: « << endl;
     for(int i = 0; i < N; i++)
```

```
        {
            for(int j = 0; j < N; j++)
            {
                cout << A[i][j] << « «;
            }
            cout << endl;
        }
        for(int i = 0; i < N; i++)
        {
            for(int j = 0; j < N; j++)
            {
                if(i != j)
                {
                    if(A[i][j] != A[j][i]) // Проверка на
                    симметричность
                    {
                        symmetric = false;
                        break;
                    }
                }
            }
        }
        if(symmetric) cout << «Данная матрица является симметричной»
        << endl;
        else cout << «Данная матрица НЕ является симметричной»
        << endl;
        return 0;
}
```

4a.c × 3b1.c ×

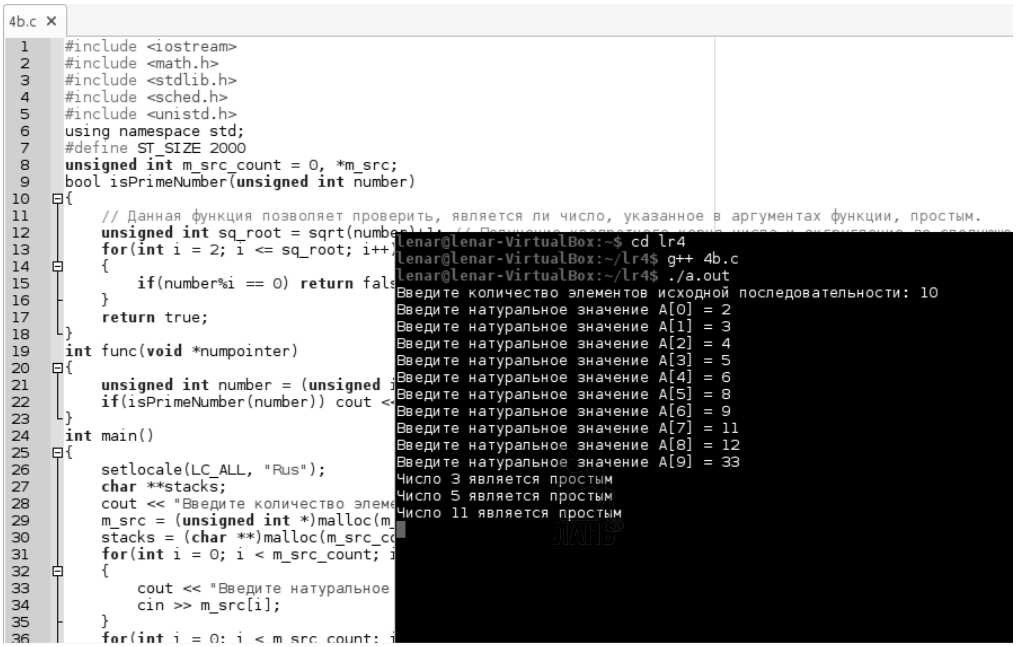
```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{
    setlocale(LC_ALL, "Rus");
    int N;
    bool symmetric = true;
    cout << "Введите размерность матрицы: ";
    cin >> N; // Получение размера матрицы
    int **A = new int*[N]; // Объявление двумерного массива размерности N*N с с выделением памяти
    for(int i = 0; i < N; i++) A[i] = new int[N]; // Выделение памяти каждому внутреннему массиву
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            cout << "Введите значение A[" << i+1 << "][" << j+1 << "]: ";
            cin >> A[i][j];
        }
    }
    cout << "Внесенный массив: " << endl;
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            cout << A[i][j] << " ";
        }
        cout << endl;
    }
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            if(i != j)
            {
                if(A[i][j] != A[j][i]) // Проверка на симметричность
                {
                    symmetric = false;
                    break;
                }
            }
        }
    }
    if(symmetric) cout << "Данная матрица является симметричной" << endl;
    else cout << "Данная матрица НЕ является симметричной" << endl;
    return 0;
}
```

```
lenar@lenar-VirtualBox:~$ g++ 4a.c
lenar@lenar-VirtualBox:~$ ./a.out
Введите размерность матрицы: 4
Введите значение A[1][1]: 1
Введите значение A[1][2]: 2
Введите значение A[1][3]: 3
Введите значение A[1][4]: 4
Введите значение A[2][1]: 2
Введите значение A[2][2]: 8
Введите значение A[2][3]: 4
Введите значение A[2][4]: 5
Введите значение A[3][1]: 3
Введите значение A[3][2]: 4
Введите значение A[3][3]: 9
Введите значение A[3][4]: 6
Введите значение A[4][1]: 4
Введите значение A[4][2]: 5
Введите значение A[4][3]: 6
Введите значение A[4][4]: 0
Внесенный массив:
1 2 3 4
2 8 4 5
3 4 9 6
4 5 6 0
Данная матрица является симметричной
lenar@lenar-VirtualBox:~$
```

В. Дана последовательность натуральных чисел a0…an-1. Создать многопоточное приложение для поиска всех ai, являющихся простыми числами.

Исходный код программы

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <sched.h>
#include <unistd.h>
using namespace std;
#define ST_SIZE 2000
unsigned int m_src_count = 0, *m_src;
bool isPrimeNumber(unsigned int number)
{
     // Данная функция позволяет проверить, является ли число,
     указанное в аргументах функции, простым.
     unsigned int sq_root = sqrt(number)+1; // Получение
     квадратного корня числа и окгругление до следующего целого
     for(int i = 2; i <= sq_root; i++)
     {
          if(number%i == 0) return false; // Проверка числа на
          наличие остатка от деления
     }
     return true;
}
int func(void *numpointer)
{
     unsigned int number = (unsigned int)numpointer;
     if(isPrimeNumber(number)) cout << «Число « << number << «
     является простым» << endl;
}
int main()
{
     setlocale(LC_ALL, «Rus»);
     char **stacks;
     cout << «Введите количество элементов исходной
     последовательности: «; cin >> m_src_count;
     m_src = (unsigned int *)malloc(m_src_count * sizeof(unsigned
     int)); // Выделение памяти под массив размерности m_count
     stacks = (char **)malloc(m_src_count * sizeof(char));
     for(int i = 0; i < m_src_count; i++)
     {
          cout << «Введите натуральное значение A[« << i << «] = «;
          cin >> m_src[i];
     }
     for(int i = 0; i < m_src_count; i++)
     {
          stacks[i] = (char *)malloc(ST_SIZE * sizeof(char));
          clone(func, stacks[i]+ST_SIZE-1, CLONE_VM | CLONE_VFORK,
          (void *)m_src[i]);
     }
     while(true) {} // Зацикливание программы
     return 0;
}
```



Лабораторная работа № 5

Цели и задачи

Познакомиться с общими принципами работы семафоров. Научиться использовать семафоры для синхронизации процессов и потоков и для защиты критических секций.

Семафоры в UNIX

При разработке средств System V IPC-семафоры вошли в их состав как неотъемлемая часть. Общими ресурсами процессов являются файлы, сегменты разделяемой памяти. Возможность одновременного изменения несколькими процессами общих данных называют критической секцией, так как такая совместная работа процессов может привести к возникновению ошибок. Например, если несколько процессов осуществляют запись данных в один и тот же файл, эти данные могут оказаться перемешанными. Наиболее простой механизм защиты критической секции состоит в расстановке «замков», пропускающих только один процесс для выполнения критической секции и останавливающий все остальные процессы, пытающиеся выполнить критическую секцию, до тех пор, пока эту критическую секцию не выполнит пропущенный процесс.

Общими данными процессов также являются каналы и сообщения. Но операции с каналами и сообщениями защищаются системой.

Набор операций над семафорами System V IPC отличается от классического набора операций {P, V}, предложенного Дейкстрой. Он включает три операции:

A(S, n) — увеличить значение семафора S на величину n;

D(S, n) — пока значение семафора S < n, процесс блокируется. Далее S = S – n;

Z(S) — процесс блокируется до тех пор, пока значение семафора S не станет равным 0.

Изначально все IPC-семафоры инициируются нулевым значением.

Классической операции P(S) соответствует операция D(S,1), а классической операции V(S) соответствует операция A(S,1). Аналогом ненулевой инициализации семафоров Дейкстры значением n может служить выполнение операции A(S,n) сразу после создания семафора S, с обеспечением атомарности создания семафора и ее выполнения посредством другого семафора.

Классические семафоры реализуются через семафоры System V IPC. Обратное не является верным. Используя операции P(S) и V(S), не получится реализовать операцию Z(S).

IPC-семафоры являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по его окончании. Пространством имен IPC-семафоров является множество значений ключа, генерируемых с помощью функции ftok(). Для совершения операций над семафорами системным вызовам в качестве параметра передаются IPC-дескрипторы семафоров, однозначно идентифицирующих их во всей вычислительной системе, а вся информация о семафорах располагается в адресном пространстве ядра операционной системы. Это позволяет организовывать через семафоры взаимодействие процессов, даже не находящихся в системе одновременно.

Системный вызов semget()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

Описание системного вызова

Системный вызов semget предназначен для выполнения операции доступа к массиву IPC-семафоров и, в случае ее успешного завершения, возвращает дескриптор System V IPC для этого массива (целое неотрицательное число, однозначно характеризующее массив семафоров внутри вычислительной системы и использующееся в дальнейшем для других операций с ним).

Параметр key является ключом System V IPC для массива семафоров, т. е. фактически его именем из пространства имен System V IPC. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции ftok(), или специальное значение IPC_PRIVATE. Использование значения IPC_PRIVATE всегда приводит к попытке создания нового массива семафоров с ключом, который не совпадает со значением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции ftok() ни при одной комбинации ее параметров.

Параметр nsems определяет количество семафоров в создаваемом или уже существующем массиве. В случае если массив с указанным ключом уже

имеется, но его размер не совпадает с указанным в параметре nsems, констатируется возникновение ошибки.

Параметр semflg — флаги — играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву, а также необходимость создания нового массива и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или — « | ») следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT — если массив для указанного ключа не существует, он должен быть создан;

IPC_EXCL — применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании массива с указанным ключом доступ к массиву не производится и констатируется ошибка, при этом переменная errno, описанная в файле <errno.h>, примет значение EEXIST;

0400 — разрешено чтение для пользователя, создавшего массив;

0200 — разрешена запись для пользователя, создавшего массив;

0040 — разрешено чтение для группы пользователя, создавшего массив;

0020 — разрешена запись для группы пользователя, создавшего массив;

0004 — разрешено чтение для всех остальных пользователей;

0002 — разрешена запись для всех остальных пользователей.

Вновь созданные семафоры инициируются нулевым значением.

Возвращаемое значение

Системный вызов возвращает значение дескриптора System V IPC для массива семафоров при нормальном завершении и значение –1 при возникновении ошибки.

int semop(int semid, sembuf *semop, size_t nops)

Функция выполняет над группой семафоров с дескриптором semid набор операций semop, nops — количество операций, выполняемых из набора semop.

Для задания операции над группой семафоров используется структура sembuf.

Первый параметр структуры sembuf определяет порядковый номер семафора в группе.

Семафоры в группе индексируются с нуля.

Второй параметр структуры sembuf представляет собой целое число = S и определяет действие, которое необходимо произвести над семафором, с индексом, записанным в первом параметре.

Если S > 0, к внутреннему значению семафора добавляется число S. Эта операция не блокирует процесс.

Если S = 0, процесс приостанавливается, пока внутреннее значение семафора не станет равно нулю.

Если S < 0, процесс должен отнять от внутреннего значения семафора модуль S.

Если значение семафора — |S| = 0, производится вычитание и процесс продолжает свою работу.

Если значение семафора — |S| < 0, процесс останавливается до тех пор, пока другой процесс не увеличит значение семафора на достаточную величину, чтобы операция вычитания выдала неотрицательный результат. Тогда производится операция вычитания и процесс продолжает свою работу. Например, если значение семафора равно трем, а процесс пытается выполнить над ним операцию –4, этот процесс будет заблокированным, пока значение семафора не увеличится хотя бы на единицу.

Третий параметр структуры sembuf может быть равен 0, тогда операция S <= 0 будет предполагать блокировку процесса, т. е. выполняться так, как описано выше. Также sembuf может быть равен IPC_NOWAIT, в этом случае работа процесса не будет останавливаться.

Если процесс будет пытаться выполнить вычитание от значения семафора, дающее отрицательный результат, эта операция просто игнорируется и процесс продолжает выполнение.

Последний параметр в функции semop определяет количество операций, берущихся для выполнения из второго параметра функции.

То есть следующий вызов:

```
sembuf Minus4 = {0, -4, 0} ;
semop( semid, &Minus4, 1) ;
```

нельзя заменить таким вызовом:

```
sembuf Minus1 = {0, -1, 0} ;
semop( semid, &Minus1, 4) ;
```

Корректной заменой может являться

```
sembuf Minus1[4] = {0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0} ;
semop( semid, Minus1, 4) ;
```

Функции для работы с семафорами (заголовочный файл — semaphore.h):

int sem_init (sem_t *sem, int pshared, unsigned int value)

Функция инициализирует семафор sem и присваивает ему значение value. Если параметр pshared больше нуля, семафор может быть доступен нескольким процессам, если pshared равен нулю, семафор создается для использования внутри одного процесса. Функция возвращает ноль в случае успеха.

Int sem_destroy (sem_t *sem)

Функция разрушает семафор sem и возвращает ноль в случае успеха.

int sem_getvalue (sem_t *sem, int *sval)

Функция записывает значение семафора **sem в *sval**. Если несколько потоков используют семафор, полученное значение семафора может быть устаревшим.

int sem_post (sem_t *sem)

Функция увеличивает значение семафора sem на единицу.

int sem_wait (sem_t *sem)

Если текущее значение семафора больше нуля, функция уменьшает значение семафора sem на единицу. Если текущее значение семафора равно нулю, выполнение потока, вызвавшего функцию sem_wait, приостанавливается до тех пор, когда значение семафора станет положительным, тогда значение семафора уменьшается на единицу и поток продолжает работу.

int sem_trywait (sem_t *sem)

Если текущее значение семафора больше нуля, функция уменьшает значение семафора sem на единицу и возвращает ноль. Если текущее значение семафора равно нулю, функция возвращает не нулевое значение. Функция sem_trywait не останавливает работу потока.

Системный вызов semctl()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, union semun arg);
```

Описание системного вызова

Системный вызов semctl предназначен для получения информации о массиве IPC-семафоров, изменения его атрибутов и удаления его из системы. Данное описание не является полным описанием системного вызова. Для изучения полного описания обращайтесь к UNIX Manual.

В данном курсе применяется системный вызов semctl только для удаления массива семафоров из системы. Параметр semid является дескриптором System V IPC для массива семафоров, т. е. значением, которое вернул системный вызов semget() при создании массива или при его поиске по ключу.

В качестве параметра cmd в рамках нашего курса мы всегда будем передавать значение IPC_RMID — команду для удаления сегмента разделяемой памяти с заданным идентификатором. Параметры semnum и arg для этой команды не используются, поэтому мы всегда будем подставлять вместо них значение 0.

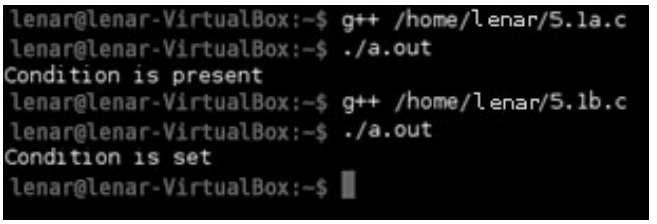
Если какие-либо процессы находились в состоянии «ожидание» для семафоров из удаляемого массива при выполнении системного вызова semop(), то они будут разблокированы и вернутся из вызова semop() с индикацией ошибки.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение –1 при возникновении ошибки.

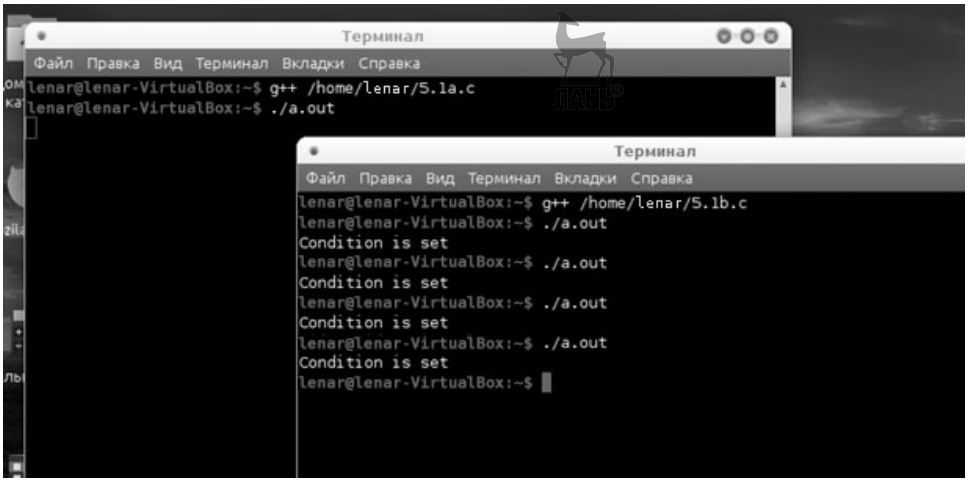
Ход работы:

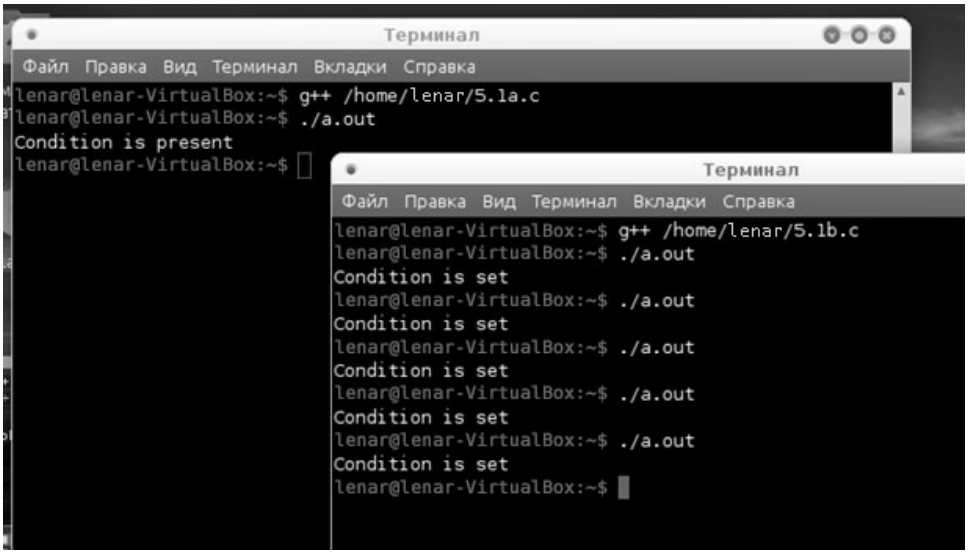
1. Первая программа выполняет над семафором S операцию D(S,1), вторая программа выполняет над тем же семафором операцию A(S,1). Первая программа блокируется до запуска программы 2, ждет, пока значение семафора не станет больше или равным, затем уменьшает его на 1.



Результат выполнения программ 5.1a.c, 5.2b.c.

2. Для того чтобы программа 5.1a.c работала без блокировки только после пятикратного выполнения 5.1.b.c, нужно в первой программе параметру структуры sembuf присвоить значение mybuf.sem_op = –5.



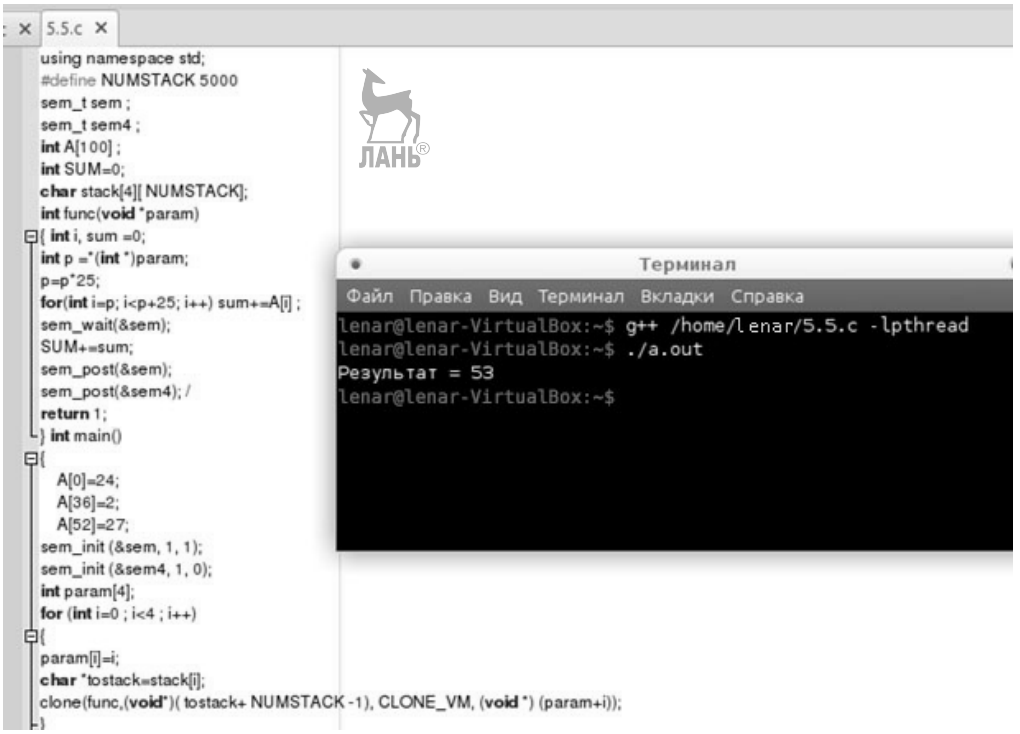


Блокировка программы 5.1a.c до пятикратного выполнения 5.2b.c.

3. Динамика работы семафоров для созданных приложений.

```
lenar@lenar-VirtualBox:~$ g++ /home/lenar/5.1a.c
lenar@lenar-VirtualBox:~$ ./a.out
Condition is present
lenar@lenar-VirtualBox:~$ ipcs -s

------ Массивы семафоров --------
ключ    semid      владелец права nsems
0xffffffff 0          lenar      666        1
0x0001b728 32769      lenar      666        1
0x0001af89 65538      lenar      666        1

lenar@lenar-VirtualBox:~$
```

4.



Пример использования семафора для синхронизации потоков.

Индивидуальное задание

Задана строка S и множество пар символов (ai, bi) i = 1, 2 … n, получить новую строку, заменив в строке S каждое вхождение ai символа на bi. Входные данные: строка S произвольной длины, целое положительное число n, множество пар символов (ai, bi) i = 1, 2 … n. Для решения задачи использовать четыре процесса (потока), разделив между ними строку S.

Исходный код программы

```
#include <iostream>
#include <stdlib.h>
#include <sched.h>
#include <unistd.h>
#include <string>
using namespace std;
#define ST_SIZE 2000
char *a, *b;
int N;
string inp_str; // Входная строка S
int replace_func(void *arg)
{
     int *ft = (int *)arg;
     for(int i = 0; i < N; i++)
     {
```

```
            for(int j = ft[0]; j <= ft[1]; j++)
            {
                inp_str[j] = (inp_str[j] == a[i]) ? b[i] :
                inp_str[j];
            }
        }
}
int main()
{
    cout << «Введите строку S, в которой необходимо произвести
    замену:» << endl;
    cin >> inp_str;
    cout << «Сколько символов в строке S необходимо
    заменить: «; cin >> N;
    a = (char *)malloc(N * sizeof(char));
    b = (char *)malloc(N * sizeof(char));
    for(int i = 0; i < N; i++)
    {
        cout << «Введите «<< i+1 << «-й символ: «; cin >> a[i];
        cout << «Введите символ для замены « << i+1 << «-го
        символа: «; cin >> b[i];
    }
    int offset = 0, section = 0, ft[4][2];
    char stacks[4][ST_SIZE];
    int inp_str_len = inp_str.length();
    if( ((inp_str_len + 1) % 4) < (inp_str_len % 4) ) // Сравнение
    остатков для определения, в какую сторону смещать сегмент
    {
        while( (inp_str_len + offset) % 4 != 0 ) offset++;
        // Смещение в большую сторону
    }
    else
    {
        while( (inp_str_len + offset) % 4 != 0 ) offset--;
        // Смещение в меньшую сторону
    }
    section = (inp_str_len + offset) / 4; // Разделение на
    сегменты
    cout << «Входная строка: « << inp_str << « Длина
    строки: « << inp_str_len << endl;
    for(int i = 0; i < N; i++)
    {
        cout << a[i] << «->« << b[i] << endl;
    }
    for(int i = 0; i < 4; i++) // Разделение строки на части
    для каждого потока
    {
        ft[i][0] = section*i;
        ft[i][1] = (i != 3) ? ( section*i + (section-1) ) :
        ( section*i + (inp_str_len-section*i-1) );
    }
    for(int i = 0; i < 4; i++) clone(replace_func,
    stacks[i]+ST_SIZE-1, (i == 3) ? CLONE_VM | CLONE_VFORK :
```

```
        CLONE_VM, (void *)ft[i]);
        sleep(3);
        cout << «Выходная строка: « << inp_str << endl;
        while(true) {} // Зацикливание программы
}
```

```
#include <iostream>
#include <stdlib.h>
#include <sched.h>
#include <unistd.h>
#include <string>
using namespace std;
#define ST_SIZE 2000
char *a, *b;
int N;
string inp_str; // Входная строка S

int replace_func(void *arg)
{
    int *ft = (int *)arg;
    for(int i = 0; i < N; i++)
    {
        for(int j = ft[0]; j <= ft[1]; j++)
        {
            inp_str[j] = (inp_str[j] == a[i]) ?
        }
    }
}

int main()
{
    cout << "Введите строку S, в которой необход
    cin >> inp_str;
    cout << "Сколько символов в строке S необход
    a = (char *)malloc(N * sizeof(char));
    b = (char *)malloc(N * sizeof(char));
    for(int i = 0; i < N; i++)
    {
        cout << "Введите "<< i+1 << "-й символ:
        cout << "Введите символ для замены " <<
    }
```

```
lenar@lenar-VirtualBox:~/lr5$ g++ 5.c
lenar@lenar-VirtualBox:~/lr5$ ./a.out
Введите строку S, в которой необходимо произвести замену:
MICROSOFT
Сколько символов в строке S необходимо заменить: 5
Введите 1-й символ: M
Введите символ для замены 1-го символа: m
Введите 2-й символ: I
Введите символ для замены 2-го символа: i
Введите 3-й символ: C
Введите символ для замены 3-го символа: c
Введите 4-й символ: R
Введите символ для замены 4-го символа: r
Введите 5-й символ: O
Введите символ для замены 5-го символа: o
Входная строка: MICROSOFT Длина строки: 9
M->m
I->i
C->c
R->r
O->o
Выходная строка: microSoFT
```

Лабораторная работа № 6

Цели и задачи

Изучить механизм коммуникации процессов — сообщения. Научиться использовать очереди сообщений для организации взаимодействия процессов и их синхронизации.

Очередь сообщений представляет собой однонаправленный связанный список, расположенный в адресном пространстве ядра. Процессы могут записывать сообщения в очередь и изымать их из очереди. Само сообщение включает в себя тип сообщения — целое положительное число и непосредственно данные. Очереди сообщений, как и семафоры, и разделяемая память, являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по окончании взаимодействия. Пространством имен очередей сообщений является то же самое множество значений ключа, генерируемых с помощью функции ftok(). Для выполнения примитивов send и receive соответствующим системным вызовам в качестве параметра передаются IPC-дескрипторы очередей сообщений, однозначно идентифицирующих их во всей вычислительной системе.

Системный вызов msgget()

Прототип системного вызова

```
#include <types.h>
#include <ipc.h>
#include <msg.h>
int msgget(key_t key, int msgflg);
```

Описание системного вызова

Системный вызов msgget предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор System V IPC для этой очереди (целое неотрицательное число, однозначно характеризующее очередь сообщений внутри вычислительной системы и использующееся в дальнейшем для других операций с ней).

Параметр key является ключом System V IPC для очереди сообщений, т. е. фактически ее именем из пространства имен System V IPC. В качестве значения этого параметра может быть использовано значение ключа, полученное с помощью функции ftok(), или специальное значение IPC_PRIVATE.

Параметр msgflg — флаги — играет роль только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди, а также необходимость создания новой очереди и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или — « | ») следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT — если очереди для указанного ключа не существует, она должна быть создана;

IPC_EXCL — применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании массива с указанным ключом доступ к очереди не производится и констатируется ошибочная ситуация, при этом переменная errno, описанная в файле <errno.h>, примет значение EEXIST;

0400 — разрешено чтение для пользователя, создавшего очередь;

0200 — разрешена запись для пользователя, создавшего очередь;

0040 — разрешено чтение для группы пользователя, создавшего очередь;

0020 — разрешена запись для группы пользователя, создавшего очередь;

0004 — разрешено чтение для всех остальных пользователей;

0002 — разрешена запись для всех остальных пользователей.

Возвращаемое значение

Системный вызов возвращает значение дескриптора System V IPC для очереди сообщений при нормальном завершении и значение –1 при возникновении ошибки.

Системный вызов msgsnd()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *ptr,
int length, int flag);
```

Описание системного вызова

Системный вызов msgsnd предназначен для помещения сообщения в очередь сообщений, т. е. является реализацией примитива send.

Параметр msqid является дескриптором System V IPC для очереди, в которую отправляется сообщение, т. е. значением, которое вернул системный вызов msgget() при создании очереди или при ее поиске по ключу.

Структура struct msgbuf описана в файле <sys/msg.h> как

```
struct msgbuf
{
long mtype;
char mtext[1];
};
```

Системный вызов msgrcv()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msqid, struct msgbuf *ptr,
int length, long type, int flag);
```

Описание системного вызова

Системный вызов msgrcv предназначен для получения сообщения из очереди сообщений, т. е. является реализацией примитива receive.

Параметр msqid является дескриптором System V IPC для очереди, из которой должно быть получено сообщение, т. е. значением, которое вернул системный вызов msgget() при создании очереди или при ее поиске по ключу.

Системный вызов msgctl()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Описание системного вызова

Системный вызов msgctl предназначен для получения информации об очереди сообщений, изменения ее атрибутов и удаления из системы.

Параметр msqid является дескриптором System V IPC для очереди сообщений, т. е. значением, которое вернул системный вызов msgget() при создании очереди или при ее поиске по ключу.

В качестве параметра cmd используется IPC_RMID — команда для удаления очереди сообщений с заданным идентификатором.

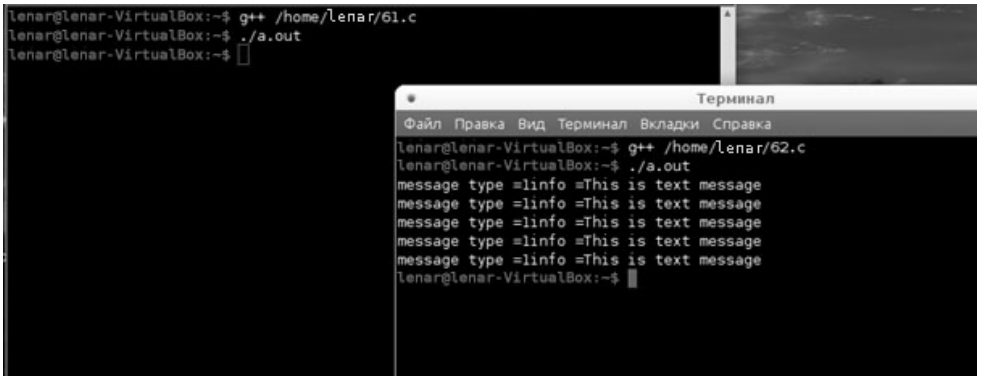
Параметр buf здесь имеет значение NULL.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение –1 при возникновении ошибки.

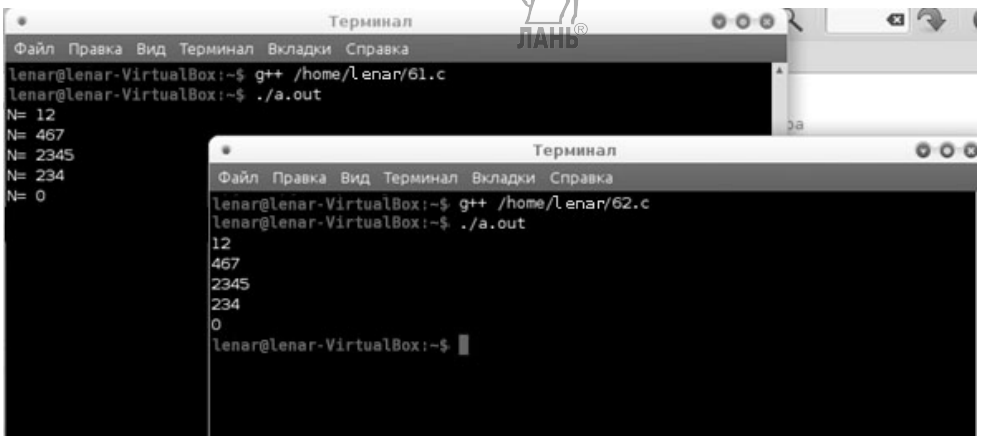
Ход работы:

1. Первая из этих программ посылает пять текстовых сообщений с типом 1 и одно сообщение нулевой длины с типом 255 второй программе. Вторая программа в цикле принимает сообщения любого типа в порядке FIFO и печатает их содержимое до тех пор, пока не получит сообщение с типом 255.



Результат выполнения программ 6.2a.c, 6.2b.c.

2. После изменений (для передачи в очередь целых чисел) в первой программе в цикле вводятся целые числа типа **int** и записываются в очередь сообщений, а вторая программа считывает их из очереди пока число не равно 0 и выводит на экран.



Результат выполнения программ 6.2a.c, 6.2b.c после изменений.

3. Пример сервера и клиентов для предложенной схемы мультиплексирования сообщений на нескольких программах. Программа 1 (сервер) ожидает получения в очереди сообщения с PID-м 2 программы (клиента) и записывает в ту же очередь сообщение с порядковым номером обработанного клиента. Такой процесс может повторяться для нескольких программ.

Листинг 1 (сервер):

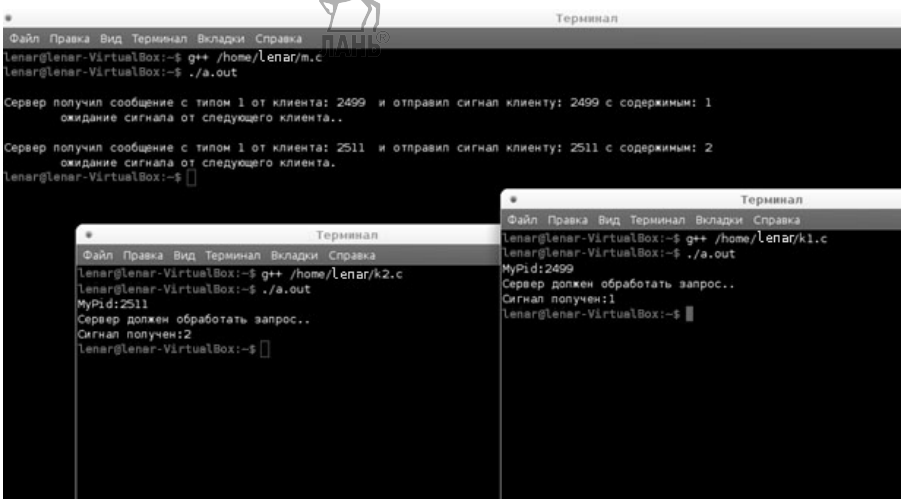
```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <iostream>
 #include <sys/types.h>
 #define N 2
using namespace std;
int msgid; //для хранения дескриптора очереди сообщений
struct mymsg //структура для сообщений
```

```
{
int mtype; //тип сообщения
int mdata; //данные сообщения
}
m;
 int main()
{
char pathname[] = «f1»; /* Имя файла, использующееся для генерации
ключа. Файл с
таким именем должен существовать в текущей директории */
key_t key; /* IPC ключ */
if((key = ftok(pathname,0)) < 0)
{
cout<<«Can't generate key»<<endl;
return 0;
}
msgid = msgget(key, 0666 | IPC_CREAT);
//если не удалось создать очередь сообщений, завершить выполнение
if (msgid < 0 )
{
cout<<«Ошибка»<<endl;
return 0;
}
for (int i=0 ; i<N; i++) //Чтение и запись в очередь для N
клиентов
{
msgrcv(msgid, (struct msgbuf *) &m, 2, 0, 0);
int pid=m.mdata;
cout<<«\nСервер получил сообщение с типом 1 от клиента: « <<pid;
m.mtype = pid; //установить тип сообщения pid
m.mdata=i+1;
msgsnd(msgid, (struct msgbuf *) &m, 2, 0);
cout<<« и отправил сигнал клиенту: «<<m.mtype<<« с содержимым:
«<<m.mdata<<endl;
cout<<«\tожидание сигнала от следующего клиента.»;
char a;
cin>>a;//чтобы клиент успел отправить в очередь свой PID
}
cout<<endl;
return 1;
}
```

Листинг 2 (клиент):

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <iostream>
#include <sys/types.h>
using namespace std;
int msgid; //для хранения дескриптора очереди сообщений
struct mymsg //структура для сообщений
{
int mtype; //тип сообщения
```

```
int mdata; //данные сообщения
}
m;
 int main()
{
char pathname[] = «f1»; /* Имя файла, использующееся для генерации
ключа. Файл с таким именем должен существовать в текущей директо-
рии */
key_t key; /* IPC ключ */
if((key = ftok(pathname,0)) < 0)
{
cout<<«Can't generate key»<<endl;
return 0;
}
msgid = msgget(key, 0666 | IPC_CREAT);
//если не удалось создать очередь сообщений, завершить выполнение
if (msgid < 0 )
{
cout<<«Ошибка»<<endl;
return 0;
}
char a;
m.mtype = 1; //установить тип сообщения в 1
m.mdata=getpid(); //записать вычисленную сумму в сообщение
int pid=m.mdata;
cout<<«MyPid:»<<pid<<endl;
msgsnd(msgid, (struct msgbuf *) &m, 2, 0);
cout<<«Сервер должен обработать запрос.»;
cin>>a;//чтобы сервер успел обработать запрос
//прочесть из очереди и вывести на экран сигнал если тип совпадает
с PID
msgrcv(msgid, (struct msgbuf *) &m, 2, 0, 0);
if (m.mtype==pid)
cout<<«Сигнал получен:»<<m.mdata<<endl;
return 1;
}
```



Пример сервера и клиентов для предложенной схемы мультиплексирования сообщений на нескольких программах.

Индивидуальное задание

Найти все n-значные числа, делящиеся нацело на каждую из своих цифр. Вычислить среднее арифметическое найденных чисел. Входные данные: целое положительное число n<11. Использовать девять процессов для решения задачи, где каждый процесс работает со своим числовым интервалом.

Лабораторная работа № 7

Цели и задачи

Изучение организации файловой системы UNIX. Знакомство с командами и системными вызовами, используемыми для работы с файлами и директориями. Рассматривается понятие memory mapped файлы.

Физические носители информации — магнитные или оптические диски, ленты и т. д., использующиеся как физическая основа для хранения файлов, в операционных системах принято логически делить на разделы (partitions) или логические диски. Причем слово «делить» не следует понимать буквально, в некоторых системах несколько физических дисков могут быть объединены в один раздел.

В операционной системе UNIX физический носитель информации обычно представляет собой один или несколько разделов. В большинстве случаев разбиение на разделы производится линейно, хотя некоторые варианты UNIX могут допускать некое подобие древовидного разбиения (Solaris). Количество разделов и их размеры определяются при форматировании диска.

Наличие нескольких разделов на диске может определяться требованиями операционной системы или пожеланиями пользователя. Если пользователь хочет разместить на одном жестком диске несколько операционных систем с возможностью попеременной работы в них, тогда он размещает каждую операционную систему в своем разделе. Или другая ситуация: необходимость работы с несколькими видами файловых систем. Под каждый тип файловой системы выделяется отдельный логический диск. Третий вариант — это разбиение диска на разделы для размещения в разных разделах различных категорий файлов.

Файл — именованный абстрактный объект, обладающий определенными свойствами. При этом в пространстве имен файлов одному файлу могут соответствовать несколько имен.

В операционной системе UNIX существуют файлы нескольких типов, а именно:

– обычные или регулярные файлы;
– директории или каталоги;
– файлы типа FIFO или именованные pip'ы;
– специальные файлы устройств;
– сокеты (sockets);
– специальные файлы связи (link).

Файлы всех перечисленных типов логически объединены в ациклический граф с однонаправленными ребрами, получающийся из дерева в результате сращивания нескольких терминальных узлов дерева или нескольких его нетерминальных узлов таким образом, чтобы полученный граф не содержал циклов. В нетерминальных узлах такого ациклического графа могут располагаться только файлы типов «директория» и «связь».

В отличие от древовидной структуры набора файлов, где имена файлов связывались с узлами дерева, в таком ациклическом графе имя файла связывается не с узлом, соответствующим файлу, а с входящим в него ребром. Ребра, выходящие из узлов, соответствующих файлам типа «связь», являются неименованными.

В файловой системе s5fs пространство имен файлов (ребер) содержит имена длиной не более 14 символов, а максимальное количество inode в одном разделе файловой системы не может превышать значения 65 535. Эти ограничения не позволяют давать файлам осмысленные имена и приводят к необходимости разбиения больших жестких дисков на несколько разделов. Зато они помогают упростить структуру хранения информации в директории.

С точки зрения пользовательского процесса каждый файл представляет собой линейный набор байт, снабженный указателем текущей позиции процесса в этом наборе. Все операции чтения из файла и записи в файл производятся в этом наборе с того места, на которое показывает указатель текущей позиции. По завершении операции чтения или записи указатель текущей позиции помещается после конца, прочитанного или записанного участка файла. Значение этого указателя является динамической характеристикой файла для использующего его процесса и также должно храниться в PCB.

Системный вызов open(). Для выполнения большинства операций над файлами через системные вызовы пользовательский процесс обычно должен указать в качестве одного из параметров системного вызова дескриптор файла, над которым нужно совершить операцию. Поэтому, прежде чем совершать операции, необходимо поместить информацию о файле в таблицы файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом open().

Системный вызов close(). Обратным системным вызовом по отношению к системному вызову open() является системный вызов close(). После завершения работы с файлом процесс освобождает выделенные ресурсы операционной системы и, возможно, синхронизирует информацию о файле, содержащуюся в таблице индексных узлов открытых файлов, с информацией на диске, используя этот системный вызов.

Системный вызов creat()

Прототип системного вызова

```
#include <fcntl.h>
int creat(char *path, int mode);
```

Системный вызов creat эквивалентен системному вызову open() с параметром flags, установленным в значение O_CREAT | O_WRONLY | O_TRUNC.

Параметр path является указателем на строку, содержащую полное или относительное имя файла.

Операция чтения атрибутов файла. Системные вызовы stat(), fstat() и lstat()

Для чтения всех атрибутов файла в специальную структуру могут применяться системные вызовы stat(), fstat() и lstat().

Прототипы системных вызовов

```
#include <sys/stat.h>
#include <unistd.h>
int stat(char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(char *filename, struct stat *buf);
```

Системный вызов stat читает информацию об атрибутах файла, на имя которого указывает параметр filename, и заполняет ими структуру, расположенную по адресу buf.

Системный вызов lstat идентичен системному вызову stat за одним исключением: если имя файла относится к файлу типа «связь», то читается информация о самом файле типа «связь».

Системный вызов fstat идентичен системному вызову stat, только файл задается не именем, а своим файловым дескриптором (естественно, файл к этому моменту должен быть открыт).

Системный вызов ftruncate()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int ftruncate(int fd, size_t length);
```

Системный вызов ftruncate предназначен для изменения длины открытого регулярного файла. Параметр fd является дескриптором соответствующего файла, т. е. значением, которое вернул системный вызов open(). Параметр length — значение новой длины для этого файла.

Системный вызов lseek()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Системный вызов lseek предназначен для изменения положения указателя текущей позиции в открытом регулярном файле. Параметр fd является дескриптором соответствующего файла, т. е. значением, которое вернул системный вызов open(). Параметр offset совместно с параметром whence определяют новое положение указателя текущей позиции.

Системный вызов symlink()

Прототип системного вызова

```
#include <unistd.h>
int symlink(char *pathname, char *linkpathname);
```

Системный вызов symlink служит для создания символической (мягкой) связи к файлу с именем, на которое указывает параметр pathname.

Системный вызов unlink()

Прототип системного вызова

```
#include <unistd.h>
int unlink(char *pathname);
```

Системный вызов unlink служит для удаления имени, на которое указывает параметр pathname, из файловой системы.

Функция opendir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(char *name);
```

Функция opendir служит для открытия потока информации для директории, имя которой расположено по указателю name. Тип данных DIR представляет собой некоторую структуру данных, описывающую такой поток. Функция opendir подготавливает почву для функционирования других функций, выполняющих операции над директорией.

Функция readdir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

Функция readdir служит для чтения очередной записи из потока информации для директории. Параметр dir представляет собой указатель на структуру, описывающую поток директории, который вернула функция opendir().

Функция rewinddir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dir);
```

Функция rewinddir служит для позиционирования потока информации для директории, ассоциированного с указателем dir, на первой записи (или на начале) директории.

Функция closedir()

Прототип функции

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

Функция closedir служит для закрытия потока информации для директории, ассоциированного с указателем dir (т. е. с тем, что вернула функция opendir()). После закрытия поток директории становится недоступным для дальнейшего использования.

С появлением концепции виртуальной памяти, когда физические размеры памяти перестали играть роль сдерживающего фактора в развитии вычислительных систем, стало возможным отображать файлы непосредственно в адресное пространство процессов. Иными словами, появилась возможность рабо-

тать с файлами как с обычной памятью, заменив выполнение базовых операций над ними с помощью системных вызовов на использование операций обычных языков программирования.

Файлы, чье содержимое отображается непосредственно в адресное пространство процессов, получили название файлов, отображаемых в память (с *англ.* — **memory mapped файлов**). Необходимо отметить, что такое отображение может быть осуществлено не только для всего файла в целом, но и для его части.

Системный вызов mmap()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
void *mmap (void *start, size_t length, int prot, int flags, int
fd, off_t offset);
```

Системный вызов mmap служит для отображения предварительно открытого файла (например, с помощью системного вызова open()) в адресное пространство вычислительной системы. После его выполнения файл может быть закрыт, что никак не повлияет на дальнейшую работу с отображенным файлом.

Параметр fd является файловым дескриптором для файла, который нужно отобразить в адресное пространство (т. е. значением, которое вернул системный вызов open()). В память будет отображаться часть файла, начиная с позиции внутри него, заданной значением параметра offset — смещение от начала файла в байтах, и длиной, равной значению параметра length (естественно, тоже в байтах). Значение параметра length может и превышать реальную длину от позиции offset до конца существующего файла. На поведении системного вызова это никак не отразится, но в дальнейшем при попытке доступа к ячейкам памяти, лежащим вне границ реального файла, возникнет сигнал SIGBUS (реакция по умолчанию — прекращение процесса).

Параметр flags определяет способ отображения файла в адресное пространство.

Параметр prot определяет разрешенные операции над областью памяти, в которую будет отображен файл.

Возвращаемое значение

При нормальном завершении системный вызов возвращает начальный адрес области памяти, в которую отображен файл (или его часть), при возникновении ошибки — специальное значение MAP_FAILED.

- После этого с содержимым файла можно работать, как с содержимым обычной области памяти.
- По окончании работы с содержимым файла необходимо освободить дополнительно выделенную процессу область памяти, предварительно синхронизировав содержимое файла на диске с содержимым этой области (если, конечно, необходимо). Эти действия выполняет системный вызов munmap().

Системный вызов munmap

Прототип системного вызова

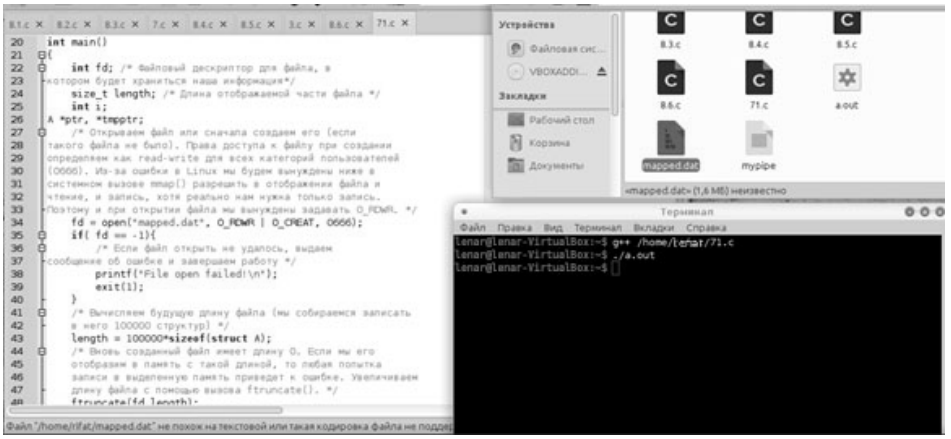
```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
int munmap (void *start, size_t length);
```

Системный вызов munmap служит для прекращения отображения memory mapped файла в адресное пространство вычислительной системы. Если при системном вызове mmap() было задано значение параметра flags, равное MAP_SHARED, и в отображении файла была разрешена операция записи (в параметре prot использовалось значение PROT_WRITE), то munmap синхронизирует содержимое отображения с содержимым файла во вторичной памяти. После его выполнения области памяти, использовавшиеся для отображения файла, становятся недоступны текущему процессу.

Параметр start является адресом начала области памяти, выделенной для отображения файла, т. е. значением, которое вернул системный вызов mmap().

Параметр length определяет ее длину, и его значение должно совпадать со значением соответствующего параметра в системном вызове mmap().

1. Программа 7.c из методического пособия создает файл, отображает его в адресное пространство процесса и заносит в него информацию.



Результат выполнения программы.

2. **Листинг 1.** Перемещение файлов, содержащих в названии определенное слово в другую директорию.

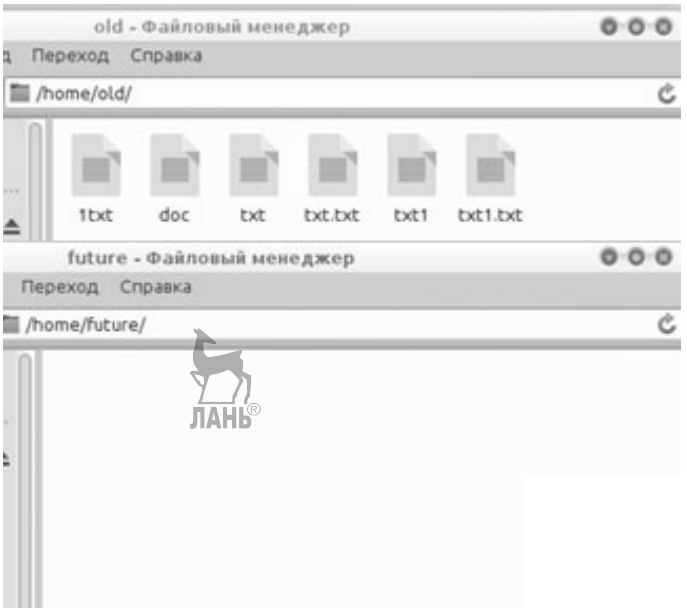
```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[], char *envp[])
{
(void) execlp(«/bin/c», «bin/c», NULL);
cout<<«Error»;
return 0;
}
```

Файл /bin/c

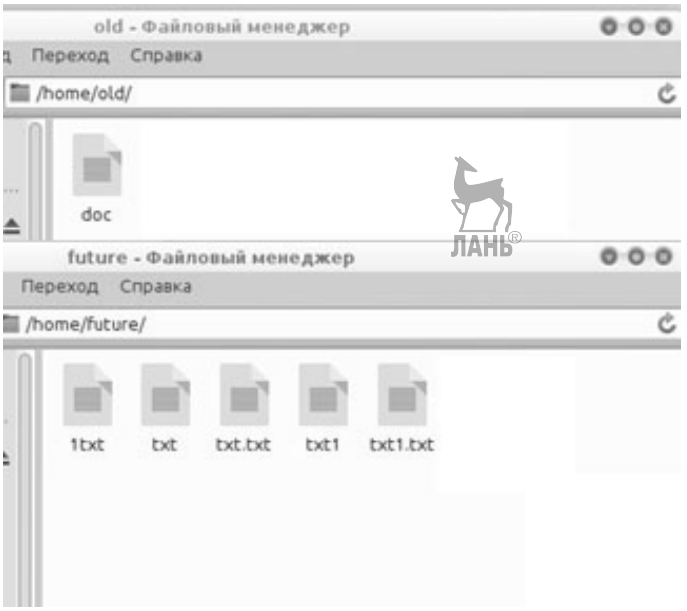
```
echo «Введите откуда переместить:»;
read old;
echo «Введите куда переместить:»;
read future;
echo «Введите имя:»;
read name;
cd $old;
mv *$name* $future;
echo «Файлы содержащие <$name> в папке <$old> перемещены в <$fu-
ture>.»;
cd;
```

```
lenar@lenar-VirtualBox:~$ g++ /home/lenar/7.c
lenar@lenar-VirtualBox:~$ sudo ./a.out
[sudo] password for lenar:
Введите откуда переместить:
/home/old/
Введите куда переместить:
/home/future/
Введите имя:
txt
Файлы содержащие <txt> в папке </home/old/> перемещены в </home/future/>.
lenar@lenar-VirtualBox:~$
```

Результат выполнения Листинга 1.



Содержимое директорий до выполнения программы.



Содержимое директорий после выполнения программы.

3. Модифицируйте программу 7-01 так, чтобы она отображала файл, записанный программой из раздела «Анализ, компиляция и прогон программы для создания memory mapped файла и записи его содержимого», в память и считала сумму квадратов чисел от 1 до 100 000, которые уже находятся в этом файле.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <iostream>
#include <sys/types.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/mman.h>
using namespace std;
/* Ниже следует описание типа структуры, которым мы забьем
файл, и двух указателей на подобный тип. Указатель ptr
будет использоваться в качестве начального адреса
выделенной области памяти, а указатель tmpptr — для
перемещения внутри этой области. */
struct A
{
    double f;
    double f2;
};
int main()
{
    int fd;
double summa=0;
  size_t length; /* Длина отображаемой части файла */
    int i;
```

```
A *ptr, *tmpptr;
    fd = open(«mapped.dat», O_RDWR, 0666);
    if( fd == -1){
        printf(«File open failed!\n»);
        exit(1);
    }
    length = 100000*sizeof(struct A);
    ftruncate(fd,length);
    ptr = (struct A*)mmap(NULL, length, PROT_WRITE | PROT_READ,
    MAP_SHARED, fd, 0);
    close(fd);
    if( ptr == MAP_FAILED ){
        printf(«Mapping failed!\n»);
        exit(2);
    }
    tmpptr = ptr;
    for(i = 0; i <100000; i++){
  summa=summa+tmpptr->f2;
tmpptr++;
    }
cout<<<«Sum: «<<<summa<<<«\n»;
    munmap((void *)ptr, length);
    return 0;
}
```

```
lenar@lenar-VirtualBox:~$ g++ /home/lenar/71.c
lenar@lenar-VirtualBox:~$ ./a.out
Sum: 3.33338e+14
lenar@lenar-VirtualBox:~$
```

Результат вычисления суммы квадратов от 1 до 1 000 000.

4. Программа, в которой выводится список файлов из директории, который вводится пользователем.

Листинг:

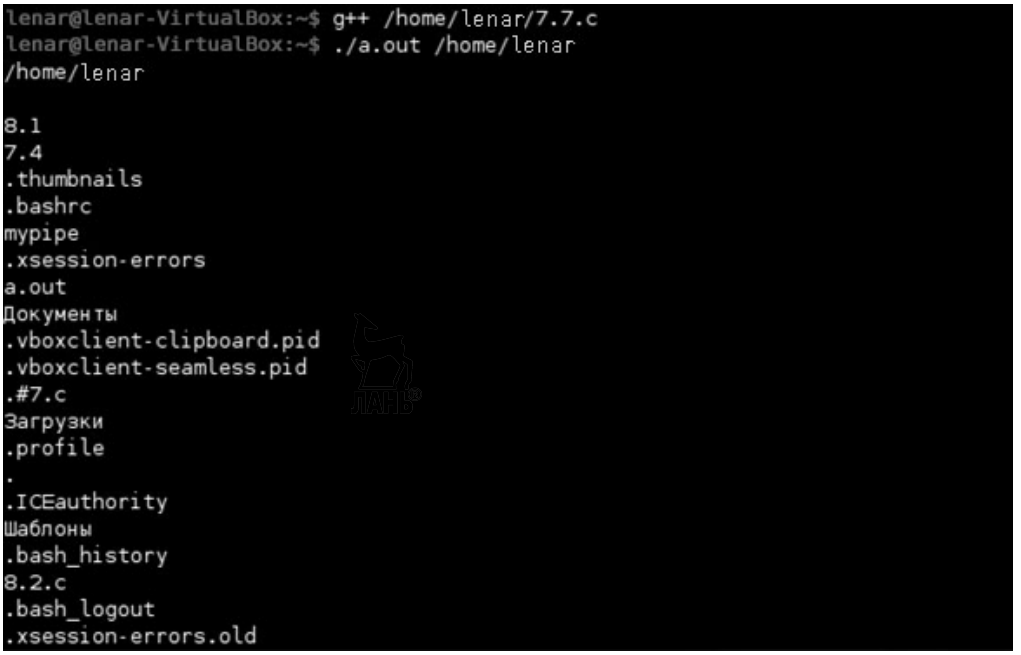
```
#include <stdio.h>
#include <dirent.h>
#include <iostream>
#include <sys/types.h>
```

```
using namespace std;
int main(int argc, char * argv[])
{
    char* directory;
    DIR*  dir;
struct dirent* direntry;
    if (argc == 1) {
        cout<<<«Используется текущая директория.»;
        *directory = '\\';
    }
    else{
        cout<<<«Текущая директория: «<<argv[1];
        directory = argv[1];
    }
    dir = opendir(directory); //открытия потока информации для
директории
    while ( (direntry = readdir(dir)) != NULL )
    {
        cout<<direntry->d_name<<<«\n»;//Вывод названия файла
        direntry = readdir(dir);//Чтение очередной записи из
директории
    }
    closedir(dir);//закрытия потока информации для директории
    return 0;
}
```

5. Напишите программу, распечатывающую содержимое заданной директории в формате, аналогичном формату выдачи команды ls -al.

Листинг:

```
#include <stdio.h>
#include <dirent.h>
#include <string.h>
int main(int argc, char **argv)
{
    DIR *dfd;
    struct dirent *dp;
    char filename[NAME_MAX];
    if ( argc < 2 )
        strcpy(filename, «.»);
    else
        strcpy(filename, argv[1]);
    printf(«%s\n\n», filename);
    dfd=opendir(filename);
    while( (dp=readdir(dfd)) != NULL )
            printf(«%s\n», dp->d_name);
    closedir(dfd);
    return 0;
}
```



Результат выполнения задания 5.

Лабораторная работа № 8

Цели и задачи

Изучение организации файловой системы UNIX. Знакомство с командами и системными вызовами, используемыми для работы с файлами и директориями. Рассматривается понятие memory mapped файлы.

Операции над файловыми системами. Монтирование файловых систем

Синтаксис команды

```
mount [-hV]
mount [-rw] [-t fstype] device dir
```

Описание команды

Команда mount предназначена для выполнения операции монтирования файловой системы и получения информации об уже смонтированных файловых системах.

Опции -h, -V используются при вызове команды без параметров и служат для следующих целей:

-h — вывести краткую инструкцию по пользованию командой;

-V — вывести информацию о версии команды mount.

Команда mount без опций и без параметров выводит информацию обо всех уже смонтированных файловых системах.

Команда mount с параметрами служит для выполнения операции монтирования файловой системы.

Параметр device задает имя специального файла для устройства, содержащего файловую систему.

Параметр dir задает имя точки монтирования (имя некоторой уже существующей пустой директории). При монтировании могут использоваться следующие опции:

-r — смонтировать файловую систему только для чтения (read only);

-w — смонтировать файловую систему для чтения и для записи (read/write).

Используется по умолчанию:

-t fstype — задать тип монтируемой файловой системы как fstype.

umount <имя точки монтирования>

где <имя точки монтирования> — это <имя пустой директории>, использованное ранее в команде mount, или в форме umount /dev/fd0, где /dev/fd0 — специальный файл устройства, соответствующего первому накопителю.

Команда umount

Синтаксис команды

```
umount [-hV]
umount device
umount dir
```

Описание команды

Команда umount предназначена для выполнения операции логического разъединения ранее смонтированных файловых систем.

Опции -h, -V используются при вызове команды без параметров и служат для следующих целей:

-h — вывести краткую инструкцию по пользованию командой;

-V — вывести информацию о версии команды umount.

Аппаратные прерывания (interrupt), исключения (exception), программные прерывания (trap, software interrupt) и их обработка

После выдачи запроса ввода-вывода у процессора существует два способа узнать о том, что обработка запроса устройством завершена. Первый способ заключается в регулярной проверке процессором бита занятости в регистре состояния контроллера соответствующего устройства (polling). Второй способ заключается в использовании прерываний. При втором способе процессор имеет специальный вход, на который устройства ввода-вывода, используя контроллер прерываний или непосредственно, выставляют сигнал запроса прерывания (interrupt request) при завершении операции ввода-вывода. При наличии такого сигнала процессор после выполнения текущей команды не выполняет следующую, а сохранив состояние ряда регистров и, возможно, загрузив в часть регистров новые значения, переходит к выполнению команд, расположенных по некоторым фиксированным адресам.

После окончания обработки прерывания можно восстановить состояние процессора и продолжить его работу с команды, выполнение которой было отложено. Аналогичный механизм часто используется при обработке исключительных ситуаций (exception), возникающих при выполнении команды процессором (неправильный адрес в команде, защита памяти, деление на ноль и т. д.).

В этом случае процессор не завершает выполнение команды, а поступает, как и при прерывании, сохраняя свое состояние до момента начала ее выполнения.

Этим же механизмом часто пользуются и для реализации так называемых программных прерываний (software interrupt, trap), применяемых, например, для переключения процессора из режима пользователя в режим ядра внутри системных вызовов. Для выполнения действий, аналогичных действиям по обработке прерывания, процессор в этом случае должен выполнить специальную команду.

Системные вызовы getpgrp(), setpgrp(), getpgid(), setpgid(), getsid(), setsid()

Системный вызов getpgid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgid(pid_t pid);
```

Описание системного вызова

Системный вызов возвращает идентификатор группы процессов для процесса с идентификатором pid. Узнать номер группы процесс может только для себя самого или для процесса из своего сеанса. При других значениях pid системный вызов возвращает значение –1.

Тип данных pid_t является синонимом для одного из целочисленных типов языка Си.

Системный вызов getpgrp()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp(void);
```

Описание системного вызова

Системный вызов getpgrp возвращает идентификатор группы процессов для текущего процесса.

Системный вызов setpgid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

Описание системного вызова

Системный вызов setpgid служит для перевода процесса из одной группы процессов в другую, а также для создания новой группы процессов.

Параметр pid является идентификатором процесса, который нужно перевести в другую группу, а параметр pgid — идентификатором группы процессов, в которую предстоит перевести этот процесс.

Не все комбинации этих параметров разрешены. Перевести в другую группу процесс может либо сам себя (и то не во всякую, и не всегда), либо свой процесс-ребенок, который не выполнял системный вызов exec(), т. е. не запускал на выполнение другую программу.

Если параметр pid равен 0, то считается, что процесс переводит в другую группу сам себя.

Если параметр pgid равен 0, то в Linux считается, что процесс переводится в группу с идентификатором, совпадающим с идентификатором процесса, определяемого первым параметром.

Если значения, определяемые параметрами pid и pgid, равны, то создается новая группа с идентификатором, совпадающим с идентификатором переводимого процесса, состоящая первоначально только из этого процесса. Переход в другую группу без создания новой группы возможен только в пределах одного сеанса.

Системный вызов setpgrp()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int setpgrp(void);
```

Описание системного вызова

Системный вызов setpgrp служит для перевода текущего процесса во вновь создаваемую группу процессов, идентификатор которой будет совпадать с идентификатором текущего процесса.

Системный вызов getsid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
pid_t getsid(pid_t pid);
```

Описание системного вызова

Системный вызов возвращает идентификатор сеанса для процесса с идентификатором pid. Если параметр pid равен 0, то возвращается идентификатор сеанса для данного процесса.

Системный вызов setsid()

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
int setsid(void);
```

Описание системных вызовов

Этот системный вызов может применять только процесс, не являющийся лидером группы, т. е. процесс, идентификатор которого не совпадает с идентификатором его группы.

Системный вызов kill() и команда kill

Команда kill обычно используется в следующей форме:

```
kill [-номер] pid
```

Здесь pid — это идентификатор процесса, которому посылается сигнал, а номер — номер сигнала, который посылается процессу.

Команда kill

Синтаксис команды

```
kill [-signal] [--] pid
kill -l
```

Описание команды

Команда kill предназначена для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

Параметр pid определяет процесс или процессы, которым будут доставляться сигналы. Он может быть задан одним из следующих четырех способов:

- Число n > 0 — определяет идентификатор процесса, которому будет доставлен сигнал.
- Число 0 — сигнал будет доставлен всем процессам текущей группы для данного управляющего терминала.
- Число –1 с предваряющей опцией -- — сигнал будет доставлен (если позволяют полномочия) всем процессам с идентификаторами, большими 1.
- Число n < 0, где n не равно –1, с предваряющей опцией -- — сигнал будет доставлен всем процессам из группы процессов, идентификатор которой равен -n.

Параметр -signal определяет тип сигнала, который должен быть доставлен, и может задаваться в числовой или символьной форме, например –9 или -SIGKILL. Если этот параметр опущен, процессам по умолчанию посылается сигнал SIGTERM.

Системный вызов kill()

Прототип системного вызова

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signal);
```

Описание системного вызова

Системный вызов kill() предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

Аргумент pid описывает, кому посылается сигнал, а аргумент sig — какой сигнал посылается.

Этот системный вызов умеет делать много разных вещей в зависимости от значения аргументов.

- Если pid > 0 и sig > 0, то сигнал номером sig (если позволяют привилегии) посылается процессу с идентификатором pid.
- Если pid = 0, а sig > 0, то сигнал с номером sig посылается всем процессам в группе, к которой принадлежит посылающий процесс.
- Если pid = –1, sig > 0 и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным дентификатором пользователя процесса, посылающего сигнал.
- Если pid = –1, sig > 0 и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов с pid = 0 и pid = 1).
- Если pid < 0, но не –1, sig > 0, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента pid (если позволяют привилегии).

• Если значение sig = 0, то производится проверка на ошибку, а сигнал не посылается, так как все сигналы имеют номера > 0. Это можно использовать для проверки правильности аргумента pid (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Возвращаемое значение

Системный вызов возвращает 0 при нормальном завершении и –1 при ошибке.

Системный вызов signal()

Прототип системного вызова

```
#include <signal.h>
void (*signal (int sig,
void (*handler) (int)))(int);
```

Описание системного вызова

Системный вызов signal служит для изменения реакции процесса на какой-либо сигнал.

Приведенное выше описание можно словесно изложить следующим образом:

– функция signal, возвращающая указатель на функцию с одним параметром типа int, которая ничего не возвращает, и имеющая два параметра: параметр sig типа int и параметр handler, служащий указателем на ничего не возвращающую функцию с одним параметром типа int;

– параметр sig — это номер сигнала, обработку которого предстоит изменить;

– параметр handler описывает новый способ обработки сигнала — это может быть указатель на пользовательскую функцию — обработчик сигнала, специальное значение IG_DFL или специальное значение SIG_IGN. Специальное значение SIG_IGN используется для того, чтобы процесс игнорировал поступившие сигналы с номером sig, специальное значение SIG_DFL — для восстановления реакции процесса на этот сигнал по умолчанию.

Возвращаемое значение

Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Ниже приведен пример скелета конструкции для пользовательской обработки сигнала SIGHUP.

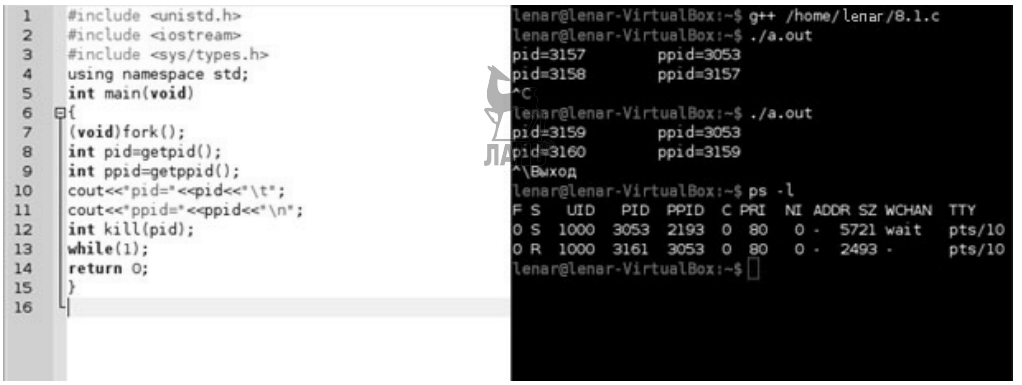
```
void *my_handler(int nsig)
{
<обработка сигнала>
} int main()
{
...
(void)signal(SIGHUP, my_handler);
...
}
```

В качестве значения параметра в пользовательскую функцию обработки сигнала (в скелете — параметр nsig) передается номер возникшего сигнала, так

что одна и та же функция может быть использована для обработки нескольких сигналов.

1. Листинг 1:

```
#include <unistd.h>
int main(void)
{
(void)fork();
int pid=getpid();
int ppid=getppid();
while(1);
return 0;
}
```

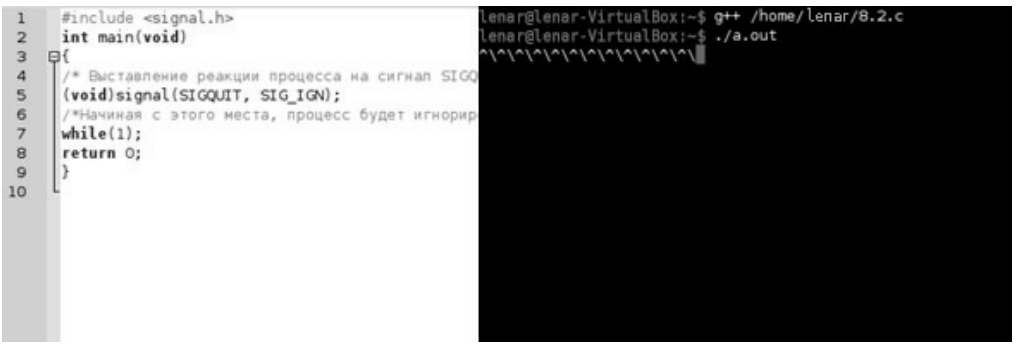


```
#include <unistd.h>
#include <iostream>
#include <sys/types.h>
using namespace std;
int main(void)
{
(void)fork();
int pid=getpid();
int ppid=getppid();
cout<<"pid="<<pid<<"\t";
cout<<"ppid="<<ppid<<"\n";
int kill(pid);
while(1);
return 0;
}
```

```
lenar@lenar-VirtualBox:~$ g++ /home/lenar/8.1.c
lenar@lenar-VirtualBox:~$ ./a.out
pid=3157        ppid=3053
pid=3158        ppid=3157
^C
lenar@lenar-VirtualBox:~$ ./a.out
pid=3159        ppid=3053
pid=3160        ppid=3159
^\Выход
lenar@lenar-VirtualBox:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY
0 S  1000  3053  2193  0  80   0 -  5721 wait   pts/10
0 R  1000  3161  3053  0  80   0 -  2493 -      pts/10
lenar@lenar-VirtualBox:~$
```

2. Листинг 2:

```
#include <signal.h>
int main(void)
{
/* Выставление реакции процесса на сигнал SIGQUIT на игнорирование
*/
(void)signal(SIGQUIT, SIG_IGN);
/*Начиная с этого места, процесс будет игнорировать возникновение
сигнала SIGQUIT */
while(1);
return 0;
}
```



```
#include <signal.h>
int main(void)
{
/* Выставление реакции процесса на сигнал SIGQ
(void)signal(SIGQUIT, SIG_IGN);
/*Начиная с этого места, процесс будет игнорир
while(1);
return 0;
}
```

```
lenar@lenar-VirtualBox:~$ g++ /home/lenar/8.2.c
lenar@lenar-VirtualBox:~$ ./a.out
^\^\^\^\^\^\^\^\^\^\^\^\^\^\^\
```

3. Листинг 3:

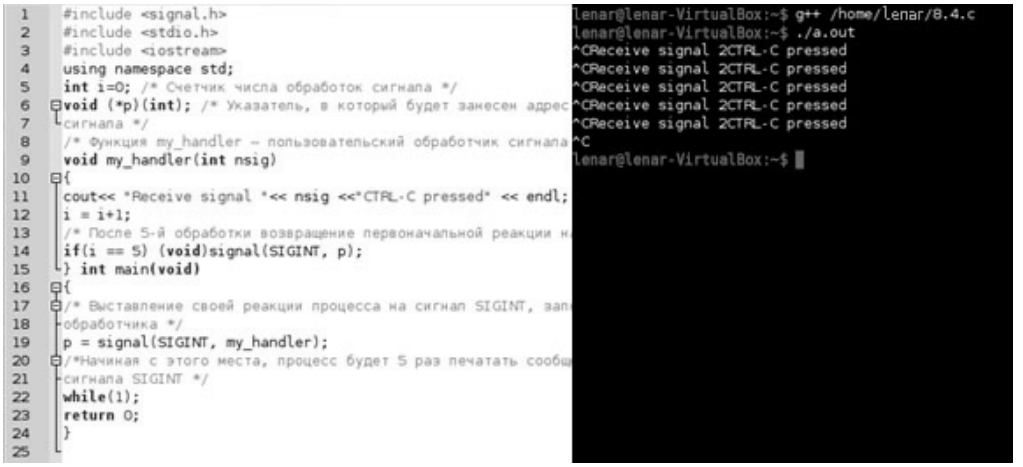
```
#include <signal.h>
```

```
#include <stdio.h>
#include <iostream>
using namespace std;
void my_handler1(int nsig)
{
cout<< «Receive signal «<< nsig <<« CTRL-C pressed» << endl;
}
void my_handler2(int nsig)
{
cout<< «Receive signal «<< nsig <<« CTRL-4 pressed» << endl;
}
int main(void)
{
(void)signal(SIGINT, my_handler1);
(void)signal(SIGQUIT, my_handler2);
while(1);
return 0;
}
```

4. Листинг 4:

```
#include <signal.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int i=0; /* Счетчик числа обработок сигнала */
void (*p)(int); /* Указатель, в который будет занесен адрес преды-
дущего обработчика сигнала */
/* Функция my_handler — пользовательский обработчик сигнала */
void my_handler(int nsig)
{
cout<< «Receive signal «<< nsig <<«CTRL-C pressed» << endl;
i = i+1;
/* После 5-й обработки возвращение первоначальной реакции на сиг-
нал */
if(i == 5) (void)signal(SIGINT, p);
} int main(void)
{
/* Выставление своей реакции процесса на сигнал SIGINT, запоминая
адрес предыдущего обработчика */
```

```
p = signal(SIGINT, my_handler);
/*Начиная с этого места, процесс будет 5 раз печатать сообщение о
возникновении сигнала SIGINT */
while(1);
return 0;
}
```



5. Листинг 5:

```
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>
#include <stdio.h>
#include <iostream>
using namespace std;
/* Функция my_handler – обработчик сигнала SIGCHLD */
void my_handler(int nsig)
{
int status;
pid_t pid;
}
int main(void)
{
pid_t pid;
(void) signal(SIGCHLD, my_handler);
if((pid = fork()) < 0)
{
cout<<«Can't fork child 1»<< endl;
return 1;
}
else if (pid == 0)
return 200;
if((pid = fork()) < 0)
{
cout<<«Can't fork child 2»<< endl;
return 1;
}
```

```
else if (pid == 0)
{
     kill(getpid(),9);//KILLED 2-TH PROSSES
while(1);
}
kill(getpid(),9); //KILLED PARENT'S PROSSES
while(1);
return 0;}
```

```
 9   void my_handler(int nsig)
10  {
11   int status;
12   pid_t pid;
13  }
14   int main(void)
15  {
16   pid_t pid;
17   (void) signal(SIGCHLD, my_handler);
18   if((pid = fork()) < 0)
19  {
20   cout<<"Can't fork child 1"<< endl;
21   return 1;
22  }
23   else if (pid == 0)
24   return 200;
25   if((pid = fork()) < 0)
26  {
27   cout<<"Can't fork child 2"<< endl;
28   return 1;
29  }
30   else if (pid == 0)
31  {
32       kill(getpid(),9);//KILLED 2-TH PROSSES
33   while(1);
34  }
35   kill(getpid(),9); //KILLED PARENT'S PROSSES
36   while(1);
37   return 0;}
```

```
lenar@lenar-VirtualBox:~$ g++ /home/lenar/8.5.c
lenar@lenar-VirtualBox:~$ ./a.out
убито
lenar@lenar-VirtualBox:~$
```

Лабораторная работа № 9

Цели и задачи

Рассмотреть функции администратора и суперпользователей. Научиться устанавливать программы различными способами. Рассмотреть различные способы архивирования.

Суперпользователь

Во всех системах на базе Linux всегда есть один привилегированный пользователь, который называется root или суперпользователь. В суперпользователя можно превратиться. Для этого необходимо выполнить команду:

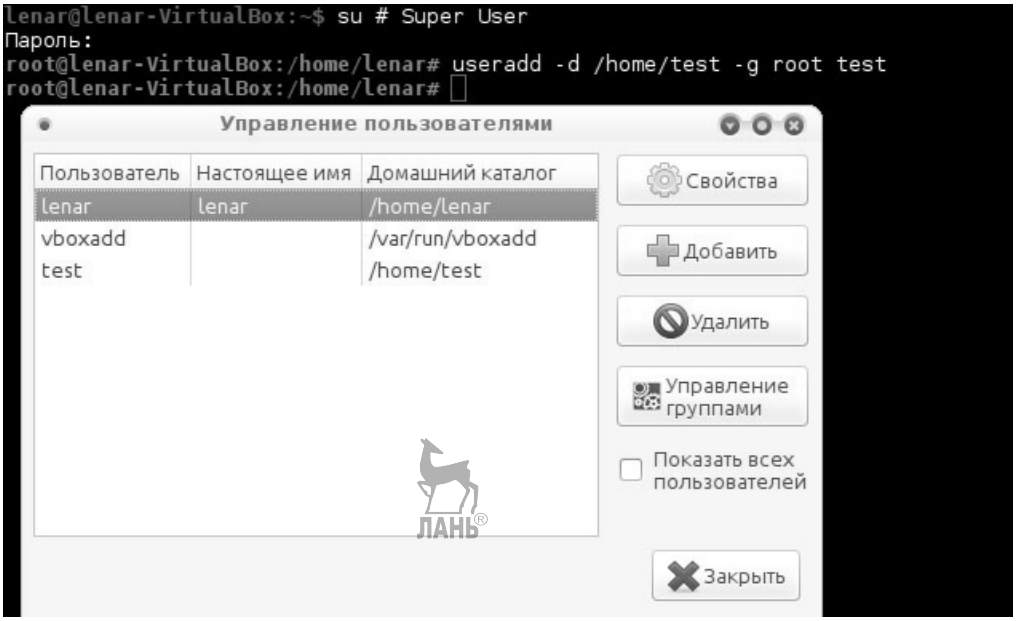
```
su # Super User
```

Администратор

Администратор по умолчанию может по запросу делать все то же самое, что и суперпользователь, однако перед выполнением каждого опасного действия система спрашивает у пользователя-администратора его пароль. Администратор является обычным пользователем, однако при необходимости он может вмешаться в работу системы, но для этого ему потребуется ввести свой пароль.

Добавление пользователя осуществляется при помощи команды useradd. Пример использования:

```
sudo useradd <имя пользователя>
```

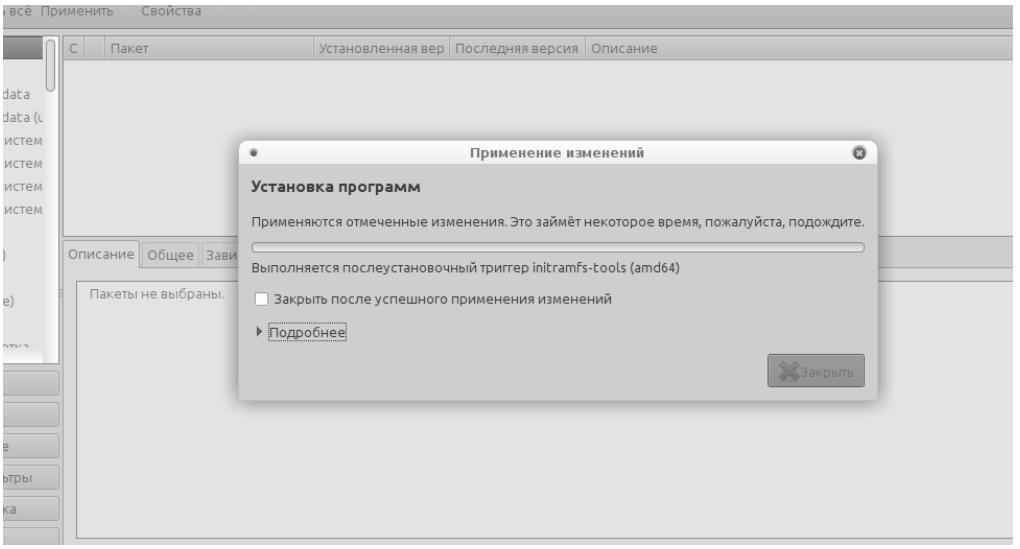


Получение прав суперпользователя, создание нового пользователя системы

В UNIX-системах, как и в других операционных системах, есть понятие зависимостей. Это значит, что программу можно установить, только если уже установлены пакеты, от которых она зависит. Такая схема позволяет избежать дублирования данных в пакетах (например, если несколько программ зависят от одной и той же библиотеки, то она установится один раз отдельным пакетом).

Установка программ:

– через менеджер пакетов;



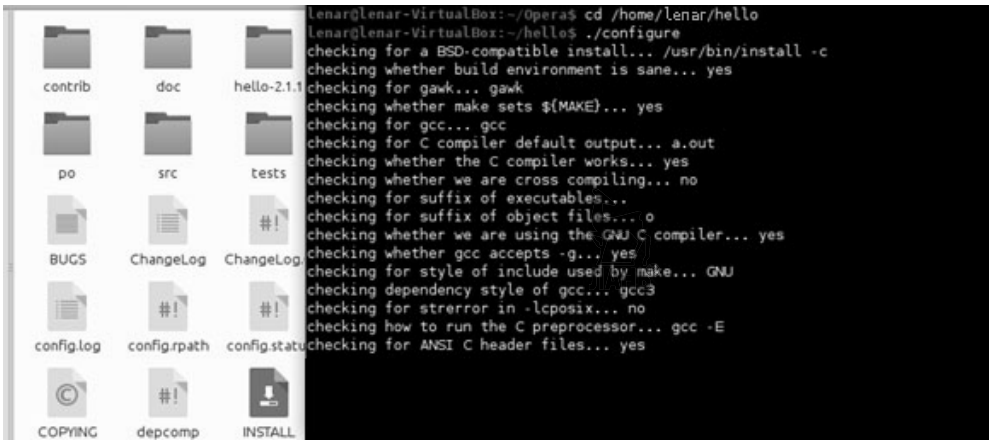
Установка программ через менеджер пакетов

– с использованием командной строки.

```
lenar@lenar-VirtualBox:~$ sudo apt-get install nano
Чтение списков пакетов… Готово
Построение дерева зависимостей
Чтение информации о состоянии… Готово
Предлагаемые пакеты:
  spell
НОВЫЕ пакеты, которые будут установлены:
  nano
обновлено 0, установлено 1 новых пакетов, для удаления отмечено 0 пакетов, и 165
 пакетов не обновлено.
Необходимо скачать 196 kБ архивов.
После данной операции, объём занятого дискового пространства возрастёт на 729 kB
.
Get:1 http://archive.ubuntu.com/ubuntu xenial/main i386 nano i386 2.5.3-2 [196 k
B]
Получено 196 kБ за 0с (227 kБ/с)
Выбор ранее не выбранного пакета nano.
(Чтение базы данных … на данный момент установлено 156094 файла и каталога.)
Подготовка к распаковке …/archives/nano_2.5.3-2_i386.deb …
Распаковывается nano (2.5.3-2) …
Обрабатываются триггеры для man-db (2.7.5-1) …
Настраивается пакет nano (2.5.3-2) …
update-alternatives: используется /bin/nano для предоставления /usr/bin/editor (
editor) в автоматическом режиме
update-alternatives: используется /bin/nano для предоставления /usr/bin/pico (pi
co) в автоматическом режиме
```

Установка программ через терминал

– установка из исходных кодов.



Установка программы из исходного кода

Архивирование

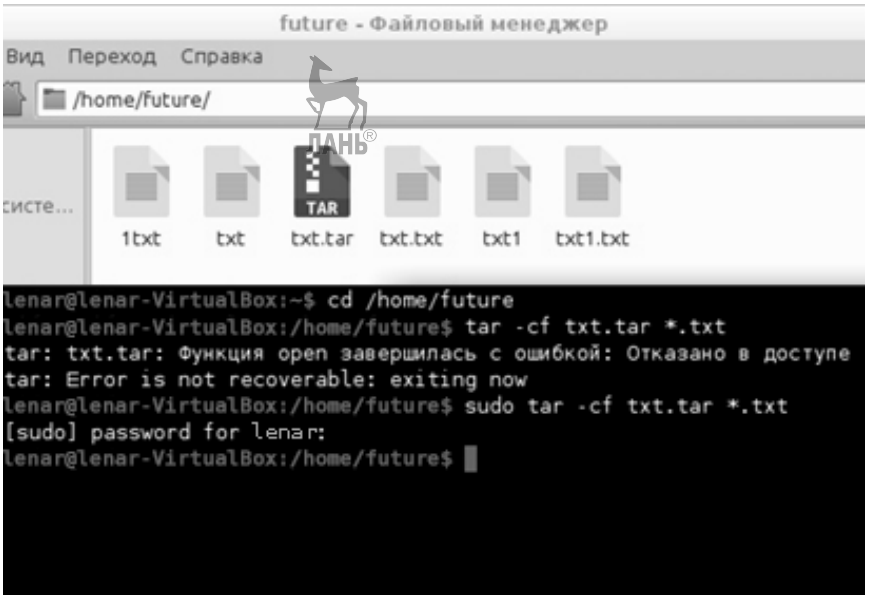
tar — наиболее распространенный архиватор, используемый в Linux-системах. Сам по себе tar не является архиватором в привычном понимании этого слова, так как он самостоятельно не использует сжатие. В то же время многие архиваторы (например, Gzip или bzip2) не умеют сжимать несколько файлов, а работают только с одним файлом или входным потоком. Поэтому чаще всего эти программы используются вместе. Tar создает несжатый архив, в который помещаются выбранные файлы и каталоги, при этом сохраняя некоторые их ат-

рибуты (такие, как права доступа). После этого полученный файл *.tar сжимается архиватором, например, gzip. Вот почему архивы обычно имеют расширение .tar.gz или .tar.bz2 (для архиваторов gzip и bzip2 соответственно).

Создание архива

Для создания архива нужно указать tar соответствующее действие, что делается с помощью ключа -c. Кроме того, для упаковки содержимого в файл необходим ключ -f. Далее указываются сначала имя будущего архива, а затем те файлы, которые необходимо упаковать.

```
tar -cf txt.tar *.txt
```



Упаковка файлов с расширением txt в архив, без сжатия

Для использования сжатия не нужно запускать что-либо еще, достаточно указать tar, каким архиватором следует сжать архив. Для двух самых популярных архиваторов gzip и bzip2 ключи будут -z и -j соответственно.

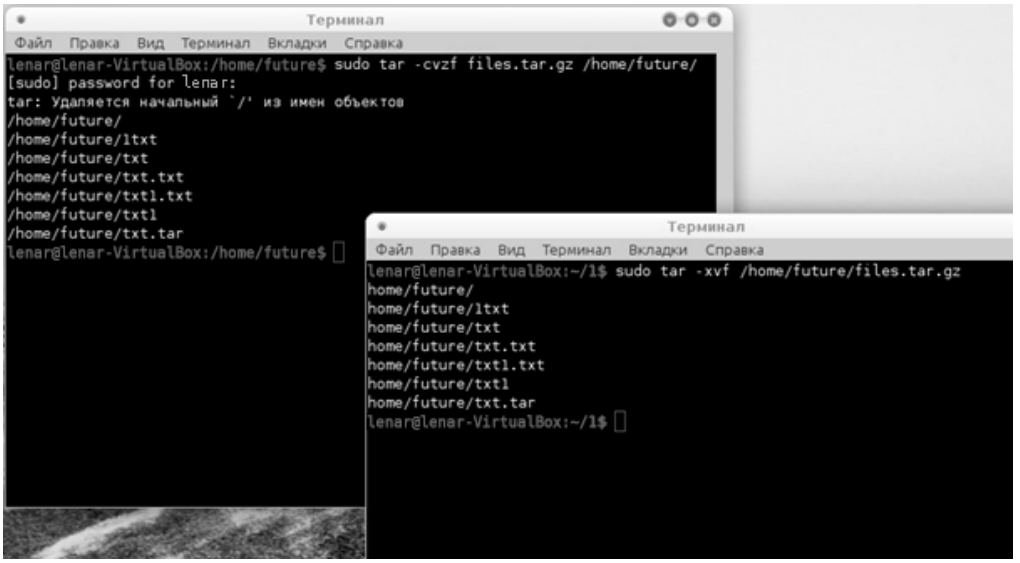
tar -cvzf files.tar.gz ~/files упакует папку ~/files со всем содержимым в сжатый с помощью gzip архив.

Действие «распаковка» задается с помощью ключа -x. И тут снова потребуется ключ -f для указания имени файла архива. Также необходим ключ -v для визуального отображения хода процесса.

```
tar -xvf /path/to/archive.tar.bz2
```

Для просмотра содержимого архива используется следующую команду:

```
tar -tf archive.tar.gz
```



Упаковка и распаковка сжатого gzip-архива

Архивация со сжатием.

Команда **cpio -o** берет с системного ввода список имен и склеивает эти файлы вместе в один архив, выталкивая его на свой системный вывод.

o - (output) — создавать архив.

```
find katalog -print | cpio -ovc > arhiwnyj-fajl.cpio
```

Команда cpio -i читает с системного ввода cpio-архив и извлекает из него файлы.

Просмотреть содержание стримера:

```
cpio -itB < /dev/rmt/ctape
```

Извлечь файлы со стримера:

```
cpio -idmvB [«шаблон» ...] < /dev/rmt/ctape
```

-B — размер блока 5120 байт — стримерный формат.

-d — создавать каталоги в случае необходимости.

-v — вывести список имен обработанных файлов.

-m — сохранять прежнее время последней модификации.

-f — брать все файлы, кроме указанного шаблоном.

-u — безусловно заменять существующий файл архивным.

-l — где можно, не копировать, а делать ссылки, получить сжатый архив, нужно воспользоваться специализированной командой compress или gzip.

Создать сжатый tar-архив:

```
tar -cvf - emacs-19.28 | compress > emacs-19.28.tar.Z
```

Прочитать оглавление сжатого tar-архива:

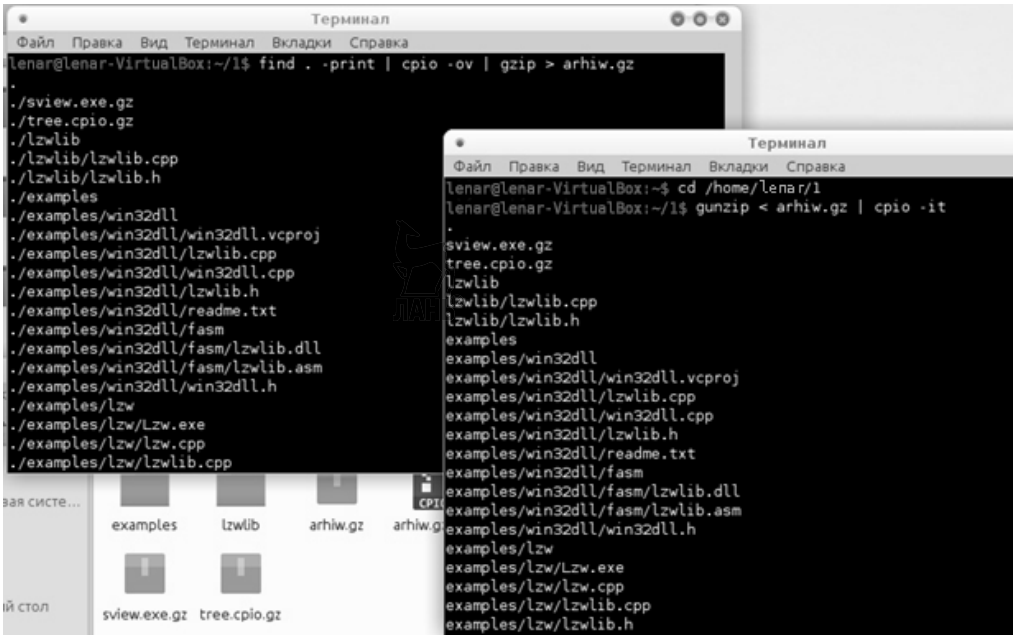
```
zcat < emacs-19.28.tar.Z | tar -tvf -name.tar.gz
```

Создать сжатый cpio-архив, используя «компрессор» gzip:

```
find . -print | cpio -ovcaB | gzip > arhiw.gz
```

Извлечь файлы из сжатого cpio-архива:

```
gunzip < arhiw.gz | cpio -idmv
```



Упаковка и распаковка сжатого cpio-архива

Лабораторная работа № 10

Цели и задачи

Изучить семейство протоколов TCP/IP. Рассмотреть использование модели клиент-сервер для взаимодействия удаленных процессов. Научиться организации связи между удаленными процессами с помощью датаграмм. Рассмотреть функции преобразования IP-адресов inet_ntoa(), inet_aton(). Научиться использовать основные функции и системные вызовы для работы с сокетами.

Описание теории:

TCP/IP — набор сетевых протоколов передачи данных, используемых в сетях, включая сеть Интернет. Название TCP/IP происходит из двух наиважнейших протоколов семейства — Transmission Control Protocol (TCP) и Internet Protocol (IP), которые были разработаны и описаны первыми в данном стандарте.

В семействе **протоколов TCP/IP** можно выделить четыре уровня.

- Уровень сетевого интерфейса.
- Уровень Internet.
- Транспортный уровень.
- Уровень приложений/процессов.

Уровень сетевого интерфейса составляют протоколы, которые обеспечивают передачу данных между узлами связи, физически напрямую соединенными друг с другом или, иначе говоря, подключенными к одному сегменту сети, и соответствующие физические средства передачи данных. К этому уровню относятся протоколы Ethernet, Token Ring, SLIP, PPP и т. д. и такие физические средства, как витая пара, коаксиальный кабель, оптоволоконный кабель и т. д.

К протоколам транспортного уровня относятся протоколы TCP и UDP.

Протокол TCP реализует потоковую модель передачи информации, хотя в его основе, как и в основе протокола UDP, лежит обмен информацией через пакеты данных. Он представляет собой ориентированный на установление логической связи (connection-oriented), надежный (обеспечивающий проверку контрольных сумм, передачу подтверждения в случае правильного приема сообщения, повторную передачу пакета данных в случае неполучения подтверждения в течение определенного промежутка времени, правильную последовательность получения информации, полный контроль скорости передачи данных) дуплексный способ связи между процессами в сети.

Протокол UDP, наоборот, является способом связи ненадежным, ориентированным на передачу сообщений (датаграмм). От протокола IP он отличается двумя основными чертами: использованием для проверки правильности принятого сообщения контрольной суммы, посчитанной по всему сообщению, и передачей информации не от узла сети к другому узлу, а от отправителя к получателю. Полный адрес удаленного процесса или промежуточного объекта для конкретного способа связи с точки зрения операционных систем определяется парой адресов: <числовой адрес компьютера в сети, локальный адрес>. Такая пара получила название socket (гнездо, панель), так как по сути дела является виртуальным коммуникационным узлом (можно представить себе виртуальный разъем или ящик для приема/отправки писем), ведущим от объекта во внешний мир и наоборот. При непрямой адресации сами промежуточные объекты для организации взаимодействия процессов также именуются сокетами.

Иерархическая система адресации, используемая в семействе протоколов TCP/IP включает в себя несколько уровней.

- Физический пакет данных, передаваемый по сети, содержит физические адреса узлов сети (MAC-адреса) с указанием на то, какой протокол уровня Internet должен использоваться для обработки передаваемых данных.
- IP-пакет данных содержит 32-битовые IP-адреса компьютера-отправителя и компьютера-получателя и указание на то, какой вышележащий протокол (TCP, UDP или еще что-нибудь) должен использоваться для их дальнейшей обработки.
- Служебная информация транспортных протоколов (UDP-заголовок к данным и TCP-заголовок к данным) должна содержать 16-битовые номера портов для сокета-отправителя и сокета-получателя.

Наиболее простой для взаимодействия удаленных процессов является схема организации общения клиента и сервера с помощью датаграмм, т. е. использование протокола UDP.

Процесс-сервер должен сначала совершить подготовительные действия: создать UDP-сокет (изготовить почтовый ящик) и связать его с определенным номером порта и IP-адресом сетевого интерфейса (прикрепить почтовый ящик в определенном месте) — настроить адрес сокета. При этом сокет может быть привязан к конкретному сетевому интерфейсу или к компьютеру в целом, то есть в полном адресе сокета может быть либо указан IP-адрес конкретного сетевого интерфейса, либо дано указание операционной системе, что информация

может поступить через любой сетевой интерфейс, имеющийся в наличии. После настройки адреса сокета операционная система начинает принимать сообщения, пришедшие на этот адрес и складывать их в сокет. Сервер дожидается поступления сообщения, читает его, определяет, от кого оно поступило и через какой сетевой интерфейс, обрабатывает полученную информацию и отправляет результат по обратному адресу. После чего процесс готов к приему новой информации от того же или другого клиента.

Процесс-клиент должен сначала совершить те же самые подготовительные действия: создать сокет и настроить его адрес. Затем он передает сообщение, указав, кому оно предназначено (IP-адрес сетевого интерфейса и номер порта сервера), ожидает от него ответа и продолжает свою деятельность.

Создание сокета производится с помощью системного вызова socket(). Для привязки созданного сокета к IP-адресу и номеру порта (настройка адреса) служит системный вызов **bind().** Ожиданию получения информации, ее чтению и, при необходимости, определению адреса отправителя соответствует системный вызов **recvfrom().** За отправку датаграммы отвечает системный вызов **sendto().**

Функции преобразования порядка байт

Прототипы функций

```
#include <netinet/in.h>
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Описание функций

- Функция htonl осуществляет перевод целого длинного числа из порядка байт, принятого на компьютере, в сетевой порядок байт.
- Функция htons осуществляет перевод целого короткого числа из порядка байт, принятого на компьютере, в сетевой порядок байт.
- Функция ntohl осуществляет перевод целого длинного числа из сетевого порядка байт в порядок байт, принятый на компьютере.
- Функция ntohs осуществляет перевод целого короткого числа из сетевого подрядка байт в порядок байт, принятый на компьютере.

В архитектуре компьютеров i80x86 принят порядок байт, при котором младшие байты целого числа имеют младшие адреса. При сетевом порядке байт, принятом в Internet, младшие адреса имеют старшие байты числа.

Параметр у них — значение, которое необходимо конвертировать. Возвращаемое значение — то, что получается в результате конвертации. Направление конвертации определяется порядком букв h (host) и n (network) в названии функции, размер числа — последней буквой названия, то есть htons — это host to network short, ntohl — network to host long.

Для чисел с плавающей точкой все обстоит гораздо хуже. На разных машинах может различаться не только порядок байт, но и форма представления такого числа. Простых функций для их корректной передачи по сети не существует. Если требуется обмениваться действительными данными, то либо это

нужно делать на гомогенной сети, состоящей из одинаковых компьютеров, либо использовать символьные и целые данные для передачи действительных значений.

Функции преобразования IP-адресов

Прототипы функций

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
int inet_aton(const char *strptr,
struct in_addr *addrptr);
char *inet_ntoa(struct in_addr *addrptr);
```

Описание функций

Функция **inet_aton** переводит символьный IP-адрес, расположенный по указателю ***strptr***, в числовое представление в сетевом порядке байт и заносит его в структуру, расположенную по адресу ***addrptr***. Функция возвращает значение 1, если в строке записан правильный IP-адрес, и значение 0 в противном случае. Структура типа ***struct in_addr*** используется для хранения IP-адресов в сетевом порядке байт и выглядит так:

```
struct in_addr
{
in_addr_t s_addr;
};
```

Функция **inet_ntoa** применяется для обратного преобразования. Числовое представление адреса в сетевом порядке байт должно быть занесено в структуру типа ***struct in_addr***, адрес которой ***addrptr*** передается функции как аргумент. Функция возвращает указатель на строку, содержащую символьное представление адреса. Эта строка располагается в статическом буфере, при последующих вызовах ее новое содержимое заменяет старое содержимое.

Функция bzero

Прототип функции

```
#include <string.h>
void bzero(void *addr, int n);
```

Описание функции

Функция bzero заполняет первые n байт, начиная с адреса addr, нулевыми значениями. Функция ничего не возвращает.

Системный вызов для создания сокета

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol)
```

Описание системного вызова

Системный вызов **socket** служит для создания виртуального коммуникационного узла в операционной системе. Данное описание не является полным описанием системного вызова, а предназначено только для использования в нашем курсе. Параметр ***domain*** определяет семейство протоколов, в рамках которого будет осуществляться передача информации. Мы рассмотрим только два

таких семейства из нескольких существующих. Для них имеются предопределенные значения параметра:

- PF_INET — для семейства протоколов TCP/IP;
- PF_UNIX — для семейства внутренних протоколов UNIX, иначе называемого еще UNIX domain.

Параметр ***type*** определяет семантику обмена информацией: будет ли осуществляться связь через сообщения (datagrams), с помощью установления виртуального соединения или еще каким-либо способом. Будем пользоваться только двумя способами обмена информацией с предопределенными значениями для параметра type:

- SOCK_STREAM — для связи с помощью установления виртуального соединения;
- SOCK_DGRAM — для обмена информацией через сообщения.

Параметр ***protocol*** специфицирует конкретный протокол для выбранного семейства протоколов и способа обмена информацией. Он имеет значение только в том случае, когда таких протоколов существует несколько.

Возвращаемое значение

В случае успешного завершения системный вызов возвращает файловый дескриптор (значение большее или равное 0), который будет использоваться как ссылка на созданный коммуникационный узел при всех дальнейших сетевых вызовах.

Системный вызов для привязки сокета к конкретному адресу

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockd, struct sockaddr *my_addr, int addrlen);
```

Описание системного вызова

Системный вызов **bind** служит для привязки созданного сокета к определенному полному адресу вычислительной сети.

Параметр ***sockd*** является дескриптором созданного ранее коммуникационного узла, т. е. значением, которое вернул системный вызов **socket().**

Параметр ***my_addr*** представляет собой адрес структуры, содержащей информацию о том, куда именно мы хотим привязать наш сокет — то, что принято называть адресом сокета. Он имеет тип указателя на структуру-шаблон struct sockaddr, которая должна быть конкретизирована в зависимости от используемого семейства протоколов и заполнена перед вызовом.

Параметр ***addrlen*** должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина в разных семействах протоколов и даже в пределах одного семейства протоколов может быть различной (например, для UNIX Domain).

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и отрицательное значение — в случае ошибки.

Системные вызовы sendto и recvfrom

Прототипы системных вызовов

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockd, char *buff, int nbytes, int flags, struct
sockaddr *to, int addrlen);
int recvfrom(int sockd, char *buff, int nbytes, int flags, struct
sockaddr *from, int *addrlen);
```

Описание системных вызовов

Системный вызов **sendto** предназначен для отправки датаграмм. Системный вызов **recvfrom** предназначен для чтения пришедших датаграмм и определения адреса отправителя. По умолчанию при отсутствии пришедших датаграмм вызов recvfrom блокируется до тех пор, пока не появится датаграмма. Вызов sendto может блокироваться при отсутствии места под датаграмму в сетевом буфере. Данное описание не является полным описанием системных вызовов, а предназначено только для использования в нашем курсе.

Параметр ***sockd*** является дескриптором созданного ранее сокета, т. е. значением, возвращенным системным вызовом **socket()**, через который будет отсылаться или получаться информация.

Параметр ***buff*** представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр ***nbytes*** для системного вызова sendto определяет количество байт, которое должно быть передано, начиная с адреса памяти **buff.** Параметр ***nbytes*** для системного вызова **recvfrom** определяет максимальное количество байт, которое может быть размещено в приемном буфере, начиная с адреса buff.

Параметр **to** для системного вызова sendto определяет ссылку на структуру, содержащую адрес сокета-получателя информации, которая должна быть заполнена перед вызовом. Если параметр from для системного вызова recvfrom не равен NULL, то для случая установления связи через пакеты данных он определяет ссылку на структуру, в которую будет занесен адрес сокета-отправителя информации после завершения вызова. В этом случае перед вызовом эту структуру необходимо обнулить.

Параметр addrlen для системного вызова sendto должен содержать фактическую длину структуры, адрес которой передается в качестве параметра to. Для системного вызова **recvfrom** параметр ***addrlen*** является ссылкой на переменную, в которую будет занесена фактическая длина структуры адреса сокета-отправителя, если это определено параметром ***from***. Заметим, что перед вызовом этот параметр должен указывать на переменную, содержащую максимально допустимое значение такой длины. Если параметр from имеет значение NULL, то и параметр ***addrlen*** может иметь значение NULL.

Параметр ***flags*** определяет режимы использования системных вызовов. Рассматривать его применение мы в данном курсе не будем, и поэтому берем значение этого параметра равным 0.

Возвращаемое значение

В случае успешного завершения системный вызов возвращает количество реально отосланных или принятых байт. При возникновении какой-либо ошибки возвращается отрицательное значение.

Системный вызов connect()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockd, struct sockaddr *servaddr, int addrlen);
```

Описание системного вызова

Системный вызов connect служит для организации связи клиента с сервером. Чаще всего он используется для установления логического соединения, хотя может быть применен и при связи с помощью датаграмм (connectionless).

Параметр ***sockd*** является дескриптором созданного ранее коммуникационного узла, т. е. значением, которое вернул системный вызов socket().

Параметр ***servaddr*** представляет собой адрес структуры, содержащей информацию о полном адресе сокета-сервера. Он имеет тип указателя на структуру-шаблон ***struct sockaddr,*** которая должна быть конкретизирована в зависимости от используемого семейства протоколов и заполнена перед вызовом.

Параметр ***addrlen*** должен содержать фактическую длину структуры, адрес которой передается в качестве второго параметра. Эта длина меняется в зависимости от семейства протоколов и различается даже в пределах одного семейства протоколов (например, для UNIX Domain).

При установлении виртуального соединения системный вызов не возвращается до его установления или до истечения установленного в системе времени — timeout. При использовании его в connectionless связи вызов возвращается немедленно.

Системный вызов listen()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockd, int backlog);
```

Описание системного вызова

Системный вызов **listen** используется сервером, ориентированным на установление связи путем виртуального соединения, для перевода сокета в пассивный режим и установления глубины очереди для соединений.

Параметр ***sockd*** является дескриптором созданного ранее сокета, который должен быть переведен в пассивный режим, т. е. значением, которое вернул системный вызов socket().

Системный вызов listen требует предварительной настройки адреса сокета с помощью системного вызова bind().

Параметр ***backlog*** определяет максимальный размер очередей для сокетов, находящихся в состояниях полностью и не полностью установленных соединений.

Системный вызов accept()

Прототип системного вызова

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockd, struct sockaddr *cliaddr, int *clilen);
```

Описание системного вызова

Системный вызов accept используется сервером, ориентированным на установление связи путем виртуального соединения, для приема полностью установленного соединения.

Параметр ***sockd*** является дескриптором созданного и настроенного сокета, предварительного переведенного в пассивный (слушающий) режим с помощью системного вызова listen().

Системный вызов accept требует предварительной настройки адреса сокета с помощью системного вызова bind().

Параметр ***cliaddr*** служит для получения адреса клиента, установившего логическое соединение, и должен содержать указатель на структуру, в которую будет занесен этот адрес.

Параметр ***clilen*** содержит указатель на целую переменную, которая после возвращения из вызова будет содержать фактическую длину адреса клиента.

Заметим, что перед вызовом эта переменная должна содержать максимально допустимое значение такой длины. Если параметр cliaddr имеет значение NULL, то и параметр clilen может иметь значение NULL.

Возвращаемое значение

Системный вызов возвращает при нормальном завершении дескриптор присоединенного сокета, созданного при установлении соединения для последующего общения клиента и сервера, и значение –1 при возникновении ошибки.

Ход работы:

1. Эта программа является UDP-клиентом для стандартного системного сервиса echo. Стандартный сервис принимает от клиента текстовую датаграмму и, не изменяя ее, отправляет обратно. За сервисом зарезервирован номер порта 51000.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
#include <stdlib.h>
void exit (int code);
using namespace std;
int main(int argc, char **argv)
{
int sockfd; /* Дескриптор сокета */
int n, len; /* Переменные для различных длин и количества символов
*/
```

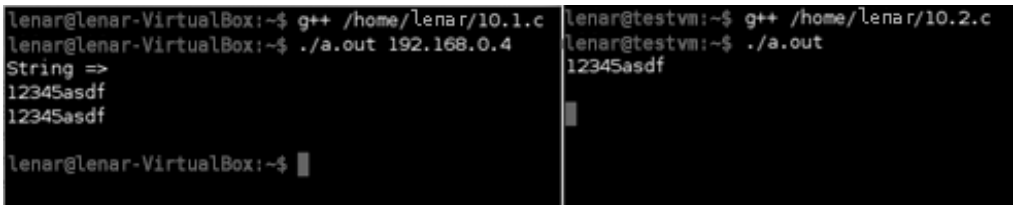
```
char sendline[1000], recvline[1000]; /* Массивы для отсылаемой и
принятой строки */
struct sockaddr_in servaddr, cliaddr; /* Структуры для адресов
сервера и клиента */
/* Сначала проверяем наличие второго аргумента в командной строке.
При его
отсутствии прекращаем работу */
if(argc != 2){
cout<<<«Usage: a.out <IP address>«<<endl;
exit(1);
}
/* Создание UDP-сокета */
if((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0){
perror(NULL); /* Печать сообщения об ошибке */
exit(1);
}
/* Заполнение структуры для адреса клиента: семейство протоколов
TCP/IP, сетевой интерфейс — любой, номер порта — по усмотрению
операционной системы. Поскольку в структуре содержится
дополнительное ненужное поле, которое должно быть нулевым, перед
заполнением обнуляем ее всю */
bzero(&cliaddr, sizeof(cliaddr));
cliaddr.sin_family = AF_INET;
cliaddr.sin_port = htons(0);
cliaddr.sin_addr.s_addr = htonl(INADDR_ANY);
/* Настройка адреса сокета */
if(bind(sockfd, (struct sockaddr *) &cliaddr, sizeof(cliaddr)) <
0){
perror(NULL);
close(sockfd); /* По окончании работы закрываем дескриптор сокета
*/
exit(1);
}
/* Заполнение структуры для адреса сервера: семейство протоколов
TCP/IP, сетевой интерфейс — из аргумента командной строки, номер
порта 7. Поскольку в структуре содержится дополнительное ненужное
поле, которое должно быть нулевым, перед  заполнением обнуляем ее
всю */
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(51000);
if(inet_aton(argv[1], &servaddr.sin_addr) == 0){
cout<<<«Invalid IP address»<<endl;
close(sockfd); /* По окончании работы закрываем дескриптор сокета
*/
exit(1);
}
/* Ввод строки, которую отошлем серверу */
cout<<<«String => «<<endl;
fgets(sendline, 1000, stdin);
/* Отсылка датаграммы */
if(sendto(sockfd, sendline, strlen(sendline)+1, 0, (struct
sockaddr *) &servaddr,
```

```
sizeof(servaddr)) < 0){
perror(NULL);
close(sockfd);
exit(1);
}
/* Ожидание ответа и чтение его. Максимальная допустимая длина
датаграммы — 1000 символов, адрес отправителя не нужен */
if((n = recvfrom(sockfd, recvline, 1000, 0, (struct sockaddr *)
NULL, NULL)) < 0){
perror(NULL);
close(sockfd);
exit(1);
}
/* Печатаем пришедший ответ и закрываем сокет */
cout<< recvline << endl;
close(sockfd);
return 0;
}
```

Пример UDP-сервера для сервиса echo:


```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{
socklen_t sockfd, clilen;
int n; /* Переменные для различных длин и количества символов */
char line[1000]; /* Массив для принятой и отсылаемой строки */
struct sockaddr_in servaddr, cliaddr; /* Структуры для адресов
сервера и клиента */
/* Заполнение структуры для адреса сервера: семейство протоколов
TCP/IP, сетевой интерфейс — любой, номер порта 51000. Поскольку в
структуре содержится дополнительное ненужное поле, которое должно
быть нулевым, перед заполнением обнуляем ее всю */
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(51000);
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
/* Создание UDP сокета*/
if((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0){
perror(NULL); /* Печать сообщения об ошибке */
exit(1);
}
/* Настройка адреса сокета */
if(bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) <
0){
```

```
perror(NULL);
close(sockfd);
exit(1);
}
while(1) {
/* Основной цикл обслуживания*/
/* В переменную clilen заносим максимальную длину для ожидаемого
адреса клиента */
clilen = sizeof(cliaddr);
/* Ожидание прихода запроса от клиента и чтение его. Максимальная
допустимая длина датаграммы — 999 символов, адрес отправителя
помещаем в структуру cliaddr, его реальная длина будет занесена в
переменную clilen */
if((n = recvfrom(sockfd, line, 999, 0, (struct sockaddr *)
&cliaddr, &clilen)) < 0){
perror(NULL);
close(sockfd);
exit(1);
}
/* Печать принятого текста на экране */
cout<< line<<endl;
/* Принятый текст отправляем обратно по адресу отправителя */
if(sendto(sockfd, line, strlen(line), 0, (struct sockaddr *)
&cliaddr, clilen) < 0){
perror(NULL);
close(sockfd);
exit(1);
} /* Ожидание новой датаграммы*/
}
return 0;
}
```



Результат выполнения программ 10.1 и 10.2.

2. Пример TCP-клиента для сервиса echo:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
#include <stdlib.h>
using namespace std;
int main(int argc, char **argv)
```

```
{
int sockfd; /* Дескриптор сокета */
int n; /* Количество переданных или прочитанных символов */
int i; /* Счетчик цикла */
char sendline[1000],recvline[1000]; /* Массивы для отсылаемой и
принятой строки */
struct sockaddr_in servaddr; /* Структура для адреса сервера */
/* Сначала проверка наличия второго аргумента в командной строке.
При его отсутствии — прекращение работы */
if(argc != 2){
cout<<«Usage: a.out <IP address>«<<endl;
exit(1);
}
/* Обнуление символьных массивов */
bzero(sendline,1000);
bzero(recvline,1000);
/* Создание TCP сокета */
if((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0){
perror(NULL); /* Печать сообщения об ошибке */
exit(1);
}
/* Заполнение структуры для адреса сервера: семейство протоколов
TCP/IP, сетевой интерфейс — из аргумента командной строки, номер
порта 7. Поскольку в структуре содержится дополнительное ненужное
поле, которое должно быть нулевым, перед заполнением обнуляем ее
всю */
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(51000);
if(inet_aton(argv[1], &servaddr.sin_addr) == 0){
cout<<«Invalid IP address»<< endl;
close(sockfd);
exit(1);
}
/* Установка логического соединения через созданный сокет с
сокетом сервера, адрес которого занесли в структуру */
if(connect(sockfd, (struct sockaddr *) &servaddr,
sizeof(servaddr)) < 0){
perror(NULL);
close(sockfd);
exit(1);
}
/* Три раза в цикле ввод строки с клавиатуры, отправка ее серверу
и чтение полученного ответа */
for(i=0; i<3; i++){
cout<<«String => «<<endl;
fflush(stdin);
fgets(sendline, 1000, stdin);
if( (n = write(sockfd, sendline, strlen(sendline)+1)) < 0){
perror(«Can\'t write\n»);
close(sockfd);
exit(1);
}
```

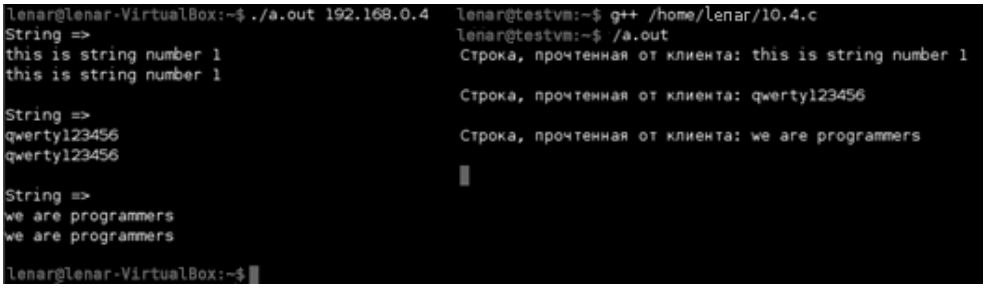
```
if ( (n = read(sockfd,recvline, 999)) < 0){
perror(«Can\'t read\n»);
close(sockfd);
exit(1);
}
cout<<recvline<<endl;
}
/* Завершение соединения*/
close(sockfd);
}
```

Пример TCP-сервера для сервиса echo:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{
       setlocale(LC_ALL,»Rus»);
socklen_t sockfd, newsockfd; /* Дескрипторы для слушающего и
присоединенного сокетов */
socklen_t clilen; /* Длина адреса клиента */
int n; /* Количество принятых символов */
char line[1000]; /* Буфер для приема информации */
struct sockaddr_in servaddr, cliaddr; /* Структуры для размещения
полных адресов сервера и клиента */
/* Создаем TCP-сокет */
if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
perror(NULL);
exit(1);
}
/* Заполнение структуры для адреса сервера: семейство протоколов
TCP/IP, сетевой интерфейс — любой, номер порта 51000. Поскольку в
структуре содержится дополнительное ненужное поле, которое должно
быть нулевым, обнуляем ее всю перед заполнением */
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family= AF_INET;
servaddr.sin_port= htons(51000);
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
/* Настраиваем адрес сокета */
if(bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) <
0){
perror(NULL);
close(sockfd);
exit(1);
}
```

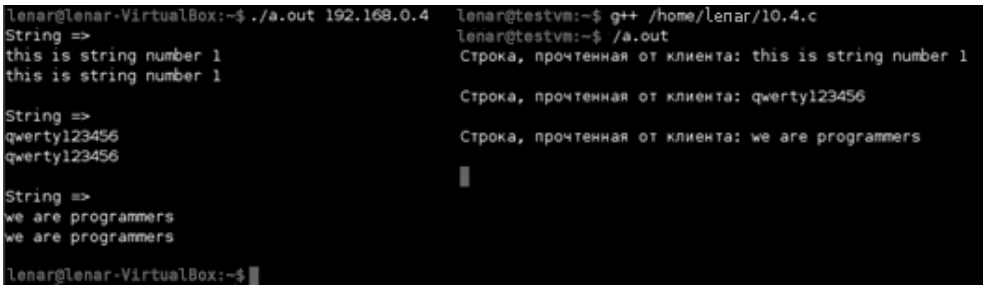
```
/* Переводим созданный сокет в пассивное (слушающее) состояние.
Глубину очереди для установленных соединений описываем значением 5
*/
if(listen(sockfd, 5) < 0){
perror(NULL);
close(sockfd);
exit(1);
}
/* Основной цикл сервера */
while(1){
/* В переменную clilen заносим максимальную длину ожидаемого
адреса клиента */
clilen = sizeof(cliaddr);
/* Ожидание полностью установленного соединения на слушающем
сокете. При нормальном завершении в структуре cliaddr будет лежать
полный адрес клиента, установившего соединение, а в переменной
clilen — его фактическая длина. Вызов вернет дескриптор
присоединенного сокета, через который будет происходить общение с
клиентом. Информация о клиенте у нас в дальнейшем никак не
используется, поэтому вместо второго и третьего параметров можно
было поставить значения NULL. */
if((newsockfd = accept(sockfd, (struct sockaddr *) &cliaddr,
&clilen)) < 0){
perror(NULL);
close(sockfd);
exit(1);
}
/* В цикле принимаем информацию от клиента до тех пор, пока не
произойдет ошибка (вызов read() вернет отрицательное значение) или
клиент не закроет соединение (вызов read() вернет значение 0).
Максимальную длину одной порции данных от клиента ограничим 999
символами. В операциях чтения и записи пользуемся дескриптором
присоединенного сокета, т. е. значением, которое вернул вызов
accept().*/
while((n = read(newsockfd, line, 999)) > 0){
/* Принятые данные отправить обратно */
cout<<«Строка, прочтенная от клиента: «<<line<<endl;
if((n = write(newsockfd, line, strlen(line)+1)) < 0){
perror(NULL);
close(sockfd);
close(newsockfd);
exit(1);
}
}
/* Если при чтении возникла ошибка — завершение работы */
if(n < 0){
perror(NULL);
close(sockfd);
close(newsockfd);
exit(1);
}
/* Закрытие дескриптора присоединенного сокета и ожидание нового
соединения */
```

```
close(newsockfd);
}
}
```



Результат выполнения программ 10.3 и 10.4.

3. Пример взаимодействия нескольких TCP-клиентов и сервера для сервиса echo.



4. UNIX Domain протокол сервера для сервиса echo, общающегося через датаграммы:

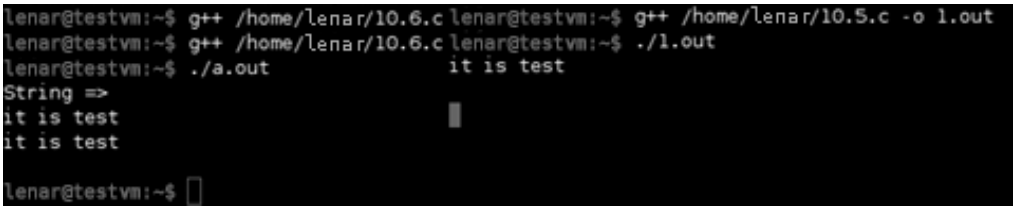
```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h> /* Новый include-файл вместо netinet/in.h и
arpa/inet.h */
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
socklen_t sockfd;
socklen_t clilen;
int n;
char line[1000];
struct sockaddr_un servaddr, cliaddr; /* новый тип данных под
адреса сокетов */
if((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) /* Изменен тип
семейства протоколов */
{
perror(NULL);
exit(1);
}
```

```
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_UNIX; /* Изменен тип семейства протоколов
и имя поля в структуре */
strcpy(servaddr.sun_path,»media/sf_connect/BBBB»); /* Локальный
адрес сокета сервера — BBBB — в текущей директории */
if(bind(sockfd, (struct sockaddr *) &servaddr, SUN_LEN(&servaddr))
< 0) /* Изменено вычисление фактической длины адреса */
{
perror(NULL);
close(sockfd);
exit(1);
}
while(1) {
clilen = sizeof(struct sockaddr_un); /* Изменено вычисление
максимальной длины для адреса клиента */
if((n = recvfrom(sockfd, line, 999, 0, (struct sockaddr *)
&cliaddr, &clilen)) < 0)
{
perror(NULL);
close(sockfd);
exit(1);
}
cout<<line<<endl;
if(sendto(sockfd, line, strlen(line), 0, (struct sockaddr *)
&cliaddr, clilen) < 0)
{
perror(NULL);
close(sockfd);
exit(1);
}
}
return 0;
}
```

UNIX Domain протокол клиента для сервиса echo, общающегося через датаграммы:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h> /* Новый include-файл вместо netinet/in.h и
arpa/inet.h */
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
#include <stdlib.h>
using namespace std;
int main() /* Аргументы командной строки не нужны, так как сервис
является локальным, не нужно указывать, к какой машине мы
обращаемся с запросом */
{
socklen_t sockfd;
int n, len;
char sendline[1000], recvline[1000];
```

```
struct sockaddr_un servaddr, cliaddr; /* новый тип данных под
адреса сокетов */
if((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
/* Изменен тип семейства протоколов */
{
perror(NULL);
exit(1);
}
bzero(&cliaddr, sizeof(cliaddr));
cliaddr.sun_family= AF_UNIX; /* Изменен тип семейства протоколов и
имя поля в структуре */
strcpy(cliaddr.sun_path,» media/sf_connect/AAAA»); /* Локальный
адрес сокета клиента — AAAA — в текущей директории */
if(bind(sockfd, (struct sockaddr *) &cliaddr, SUN_LEN(&cliaddr)) <
0) /* Изменено вычисление фактической длины адреса */
{
perror(NULL);
close(sockfd);
exit(1);
}
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_UNIX; /* Изменен тип семейства протоколов
и имя поля в структуре */
strcpy(servaddr.sun_path,»media/sf_connect/BBBB»); /* Локальный
адрес сокета сервера — BBBB — в текущей директории */
cout<<«String => «<<endl;
fgets(sendline, 1000, stdin);
if(sendto(sockfd, sendline, strlen(sendline)+1, 0, (struct
sockaddr *) &servaddr,
SUN_LEN(&servaddr)) < 0) /* Изменено вычисление фактической длины
адреса */
{
perror(NULL);
close(sockfd);
exit(1);
}
if((n = recvfrom(sockfd, recvline, 1000, 0,(struct sockaddr *)
NULL, NULL)) < 0){
perror(NULL);
close(sockfd);
exit(1);
}
recvline[n] = 0;
cout<< recvline<< endl;
close(sockfd);
return 0;
}
```



Результат выполнения программ 10.5 и 10.6.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. *Назаров, С. В.* Операционные системы : учеб. пособие для студентов вузов, обучающихся по направлению подготовки 080700 «Бизнес-информатика» / С. В. Назаров, Л. П. Гудыно, А. А. Кириченко. — М. : Кнорус, 2012. — 371 с.

2. *Назаров, С. В.* Современные операционные системы : учеб. пособие / С. В. Назаров, А. И. Широков. — 2-е изд., испр. и доп. — М. : БИНОМ. Лаборатория знаний, 2013. — 367 с.

3. Концепция виртуальной машины и операционные системы, основанные на данной концепции / Н. А. Староверова, Р. К. Хакимзянов // Естественные и технические науки. — 2018. — № 8(122). — С. 216–218.

4. Современные тенденции и перспективы развития операционных систем / Н. А. Староверова, Д. Морозов, И. Калаева, Г. Кадырова // Вестник Технологического университета. — 2015. — Т. 18. — № 21. — С. 134–136.

5. Принцип уровней абстракции и его использование при разработке операционных систем / Д. Николаев, Н. Староверова, А. Хасанова // Вестник Технологического университета. — 2015. — Т. 18. — № 10. — С. 180–183

6. Аналитический обзор и сравнение возможностей операционных систем для мобильных устройств / А. Зиятдинова, Н. А. Староверова // Фундаментальные исследования. — 2015. — № 9–2. — С. 227–231.

7. *Таненбаум, Э.* Современные операционные системы / Э. Таненбаум, Х. Босх — 4-е изд. — СПб. : Питер, 2016. — 1120 с.

8. *Таненбаум, Э.* Архитектура компьютера / Э. Таненбаум. — 5-е изд. — СПб. : Питер, 2013. — 844 с.

9. *Таненбаум, Э.* Операционные системы. Разработка и реализация. Классика CS / Э. Таненбаум, А. Вудхалл. — 3-е изд. — СПб. : Питер, 2007. — 704 с.

10. Олифер, В. Г. Компьютерные сети. Принципы, технологии, протоколы : учебник для вузов / В. Г. Олифер. — 3-е. изд. — СПб. : Питер, 2010. — 958 с.

11. http://www.intuit.ru/studies/courses/2249/52/info — НОУ ИНТУИТ Академия Intel : Основы операционных систем. Практикум.

12. www.linuxcenter.ru. — магазин, дистрибутивы Linux, подборка статей по Linux, новости в мире Linux.

13. www.linux-online.ru. — интернет-магазин, дистрибутивы Linux, книги по администрированию, ссылки.

14. www.citfonim.ru. — сервер информационных технологий, содержит большой объем информации по информационным технологиям, в том числе и по Linux.

15. www.linux.webclub.ru. — новости в мире Linux.

16. www.linuxrsp.ru. — интернет-проект «Все о Linux по-русски» (статьи, рассылки, документация и т. п.).

17. Операционная система Linux: Часть 2. Терминал и командная строка.

18. Учебное пособие от компании IBM // http://www.ibm.com/developer works/ru/edu/l-dw-linuxredbook2.html.
19. Программирование на Shell (UNIX) // Библиотека http://www.linux center.ru/lib/books/shell/.
20. Novell Open Suse — официальный сайт проекта на русском https://ru.opensuse.org/.

ОГЛАВЛЕНИЕ

Наталья Александровна СТАРОВЕРОВА

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебник

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*
Ответственный редактор *Н. А. Кривилёва*
Подготовка макета *Э. Я. Юзеев*
Корректор *Е. М. Матвеева*
Выпускающий *О. В. Шилкова*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com
196105, Санкт-Петербург, пр. Юрия Гагарина, д.1, лит. А.
Тел.: (812) 336-25-09, 412-92-72.
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 26.12.20.
Бумага офсетная. Гарнитура Школьная. Формат 70×100 $^{1}/_{16}$.
Печать офсетная. Усл. п. л. 33,48. Тираж 30 экз.

Заказ № 020-21.

Отпечатано в полном соответствии
с качеством предоставленного оригинал-макета
в АО «Т8 Издательские технологии»
109316, г. Москва, Волгоградский пр., д. 42, к. 5.