КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ ФИЗИКИ

Кафедра радиофизики

Е.Ю. РЯБЧЕНКО

АРХИТЕКТУРА И БЕЗОПАСНОСТЬ ОПЕРАЦИОННЫХ СИСТЕМ

Учебное пособие

КАЗАНЬ — 2015

УДК 004.2, 004.4, 004.5

Принято на заседании кафедры радиофизики
Протокол № 8 от 25 марта 2015 года

Рецензент:
кандидат физико-математических наук,
старший инженер-программист, ООО «ДжиДиСи Сервисез» **А.Ю. Елькин**

Рябченко Е.Ю.

Архитектура и безопасность операционных систем. Учебное пособие. / Е.Ю. Рябченко. — Казань: Казан. ун-т, 2015. — 157 с.

В пособии излагаются фундаментальные принципы построения современных операционных систем на примере архитектуры UNIX. Рассматриваются технологии построения ядра операционных систем, механизмы управления памятью и защиты данных, файловые системы, процессы и механизмы их взаимодействия, а также аспекты многопользовательской системы контроля доступа. Особое внимание уделяется UNIX-совместимым операционным системам Solaris и Linux.

Пособие рекомендуется студентам Института физики КФУ — слушателям курсов лекций «Безопасность операционных систем» и «Сетевые операционные системы».

Содержание

Введение

В настоящем учебном пособии рассматриваются фундаментальные принципы построения современных операционных систем (ОС) на примере архитектуры ОС UNIX. Выбор данной архитектуры связан, в первую очередь, с доминирующим положением ОС UNIX в сфере интернет-серверов, центров обработки и хранения данных, суперкомпьютеров, систем управления технологическими процессами. Необходимо отметить также возрастающую популярности рабочих станций, персональных и мобильных компьютеров на основе UNIX-совместимых ОС Linux и MacOS X.

В последнее время возросла доля ОС Linux и в области встраиваемых и мобильных систем. Сегодня большинство всех домашних и офисных сетевых устройств (маршрутизаторов, точек доступа Wi-Fi), а также мультимедийные устройства, электронные книги имеют в качестве ОС систему Linux. Отдельно стоит отметить ОС Android, основанную на ядре Linux, доля которой на рынке мобильных устройств в 2014 г. составила 85%.

Выбор ОС UNIX для изучения оправдан также с методической точки зрения. Традиционно UNIX-системы являются более открытыми для пользователей и разработчиков как в плане спецификации программного интерфейса, так и в плане реализации. Для ОС Linux свободно доступны исходные коды ядра, библиотек и большей части программного обеспечения (ПО) по лицензии GNU/GPL, не требующей каких-либо выплат. В последнее время с появлением проекта свободной ОС OpenSolaris, разрабатываемой сообществом на основе частично открытых исходных кодов коммерческой ОС Solaris, стало доступным и более детальное изучение реализации последней.

Архитектура UNIX-систем, отличительной чертой которой является высокая степень модульности при хорошей документированности, способствует лучшему пониманию принципов функционирования не только самой ОС, но и компьютерной системы в целом. При этом не последнюю роль играет ставший для ОС UNIX стандартным (но не единственным) пользовательский интерфейс командной строки с широчайшими возможностями автоматизации управления компьютером посредством сценариев. Многие идеи, заложенные в ОС UNIX, стали впоследствии основой для разработки ряда других популярных ОС, таких как IBM OS/2 или Microsoft Windows.

Изучение ОС UNIX будет крайне полезно, в первую очередь, будущим специалистам в области сетевого и системного администрирования, независимо от того, с какой конкретно ОС им придется иметь дело. Кроме того, знания ОС UNIX понадобятся всем разработчикам ПО для компьютеров и встраиваемых систем, основанных на данной ОС.

При написании учебного пособия автор старался максимально последовательно излагать материала. Однако, рассматривая такой сложный объект как ОС, часто приходится забегать вперед, чтобы ввести очередное понятие. Поэтому в начале пособия, в гл. 1, кратко рассматривается структура ОС и даются основные определения, которые в последующих главах используются при подробном изложении материала.

Некоторые аспекты архитектуры ОС, такие как механизмы защиты ядра и виртуальная память, рассмотрены в данном пособии на основе конкретной аппаратной архитектуры (или *аппаратной платформы*) — x86, включая 32-битный (IA-32) и 64-битный (AMD64) варианты. Особенности реализации ОС UNIX для других платформ можно найти в [1] или в других многочисленных источниках.

Данное пособие призвано дать читателю основные теоретические сведения об ОС UNIX и подготовить его для дальнейших практических занятий по работе и программированию в этой ОС. По мере изложения материала рассматриваются основные системные вызовы и функции стандартной библиотеки без строгого указания их прототипа и синтаксиса вызова. Для практического их применения читателю необходимо обратиться к справочному материалу по конкретной реализации ОС UNIX и, в первую очередь, к поставляемым с каждой версией ОС руководствам `man`. Предполагается, что читатель имеет основные представления об аппаратном обеспечении компьютеров, а также владеет основами программирования на языке Си.

1. Назначение и структура операционной системы

1.1. Назначение операционной системы

ОС является важнейшим компонентом любого современного вычислительного устройства от смартфона до суперкомпьютера. ОС появились практически сразу же после создания первых электронных вычислительных машин (ЭВМ) в 1950–1960-е гг. В Энциклопедии кибернетики, изданной в 1974 г. [2], ОС определяется как «комплекс программ, осуществляющих управление вычислительным процессом и реализующих наиболее общие алгоритмы обработки информации на данной ЭВМ». Такое определение является наиболее общим и в целом остается актуальным и сегодня.

С учетом некоторых уточнений, современную ОС можно определить как *базовый комплекс программ, обеспечивающих взаимодействие прикладного программного обеспечения (ПО) и аппаратных средств вычислительной системы*, включая распределение вычислительного ресурса центрального процессора (или процессоров), управление оперативной памятью и устройствами ввода-вывода, хранение данных на носителях информации, а также интерфейс с пользователем.

ОС позволяет абстрагироваться от деталей реализации аппаратного обеспечения, предоставляя разработчику унифицированный *программный интерфейс*, через который происходит взаимодействие ПО и компьютерного оборудования.

Перечислим основные задачи, которые должна решать ОС:

1) управление оперативной памятью, включая виртуальную память,
2) стандартизированный доступ к устройствам ввода-вывода,
3) хранение данных на внешних носителях (файловая система),
4) загрузка файлов приложений в оперативную память и их выполнение,
5) организация пользовательского интерфейса.

Большинство современных ОС имеет значительно больший спектр решаемых задач, определяемый назначением каждой конкретной ОС. Наиболее общими дополнительными задачами являются:

1) параллельное или псевдопараллельное выполнение процессов,

2) организация взаимодействия между процессами (обмен данными, взаимная синхронизация),
3) сетевое взаимодействие с другими компьютерами (сетевые и телекоммуникационные функции),
4) многопользовательский режим работы и разграничение прав доступа между процессами разных пользователей,
5) защита системы, а также пользовательских данных и программ от действий других пользователей или приложений.

Таким образом, в простейшем случае ОС должна уметь выполнять одну программу, т.е. обеспечить её загрузку в оперативную память и передать ей управление. Сегодня такие ОС уходят в прошлое или имеют очень узкую сферу применения. В большинстве случаев мы имеем дело с *многозадачными ОС*, способными выполнять одновременно несколько задач, т.е. загружать в оперативную память несколько программ и обеспечивать распределение вычислительного ресурса между ними. В этом случае мы имеем дело уже не просто с загруженной программной, а с *процессом*, исполняемым и контролируемым ОС. *Процесс* является ключевым понятием в многозадачной ОС и определяется как *совокупность инструкций, выполняемых процессором, данных программы, а также информации о выполняемой задаче, включая параметры выделенной оперативной памяти, открытые файлы и статус процесса*.

В многозадачной ОС возможно как *псевдопараллельное* выполнение задач, т. е. поочередное переключение общего вычислительного ресурса между задачами, если в системе один процессор, так и истинно *параллельное* выполнение задач, применяемое в многопроцессорных (распределенных) вычислительных системах. Существуют также ОС, в которых применяется *невытесняющая многозадачность*, при которой ОС одновременно загружает в память несколько приложений, но процессорное время предоставляется только основному (активному) приложению; для выполнения фонового приложения оно должно быть активизировано[1]. В современных версиях ОС UNIX применяется параллельное или псевдопараллельное выполнение задач. Процессы будут подробно рассмотрены в гл. 7.

[1]Такой тип многозадачности существовал в системах NetWare и Windows 3.x.

Большинство современных ОС обеспечивают многозадачный режим работы на основе технологии *виртуальной памяти*, позволяющей каждому процессу для программного кода, стека и данных предоставлять отдельные *виртуальные адресные пространства*. При выполнении микропроцессорных инструкций данного процесса все виртуальные адреса в конечном счете особым образом отображаются в пространство реальной физической памяти. Применение виртуальной памяти позволяет создавать позиционно независимый программный код, изолировать память процессов друг от друга, контролировать доступ в выделяемые области памяти для каждого процесса. Виртуальная память будет подробно рассмотрена в гл. 3.

1.2. Структура ОС

Структуру ОС в самом общем виде можно представить при помощи схемы, показанной на рис. 1. Центральной частью ОС является *ядро*, реализующее бóльшую часть рассмотренного выше функционала. Взаимодействие всех остальных компонентов ОС с аппаратным обеспечением происходит через ядро. Для этого ядро реализует набор так называемых *системных вызовов*(англ. system calls), представляющих собой подпрограммы ядра, которые идентифицируются номером. Таким образом, для выполнения какой-либо функции ядра процессу необходимо сгенерировать специальное программное прерывание с указанием номера требуемого системного вызова. Механизм реализации таких прерываний зависит от архитектуры процессора.

Для каждой из рассмотренных групп задач в ОС могут присутствовать

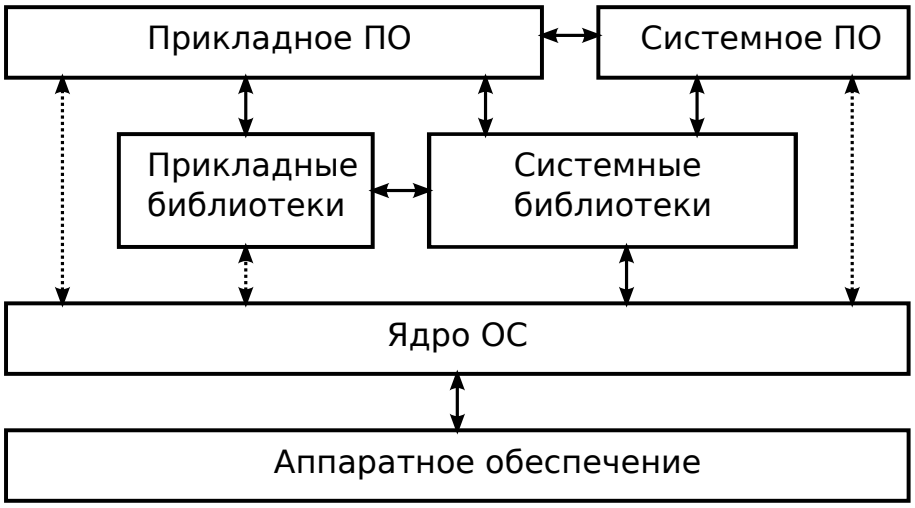

Рис. 1. Структура операционной системы.

десятки системных вызовов. Общее количество системных вызовов может варьироваться от нескольких десятков до нескольких сотен. Например, ядро ОС Linux версии 3.x включает около 400 системных вызовов.

Важной частью ОС являются *системные библиотеки*, содержащие основной набор системных функций, необходимых для работы большинства системных и прикладных программ. Для удобства разработки ПО и возможности создания переносимых программ (на уровне исходного кода) вызовы ядра также оформляются в виде функций системной библиотеки. Это позволяет организовать удобный программный интерфейс, в котором все системные вызовы имеют легко запоминающиеся названия библиотечных функций. Некоторые функции системных библиотек могут содержать код, непосредственно не связанный с обращением к системным вызовам ядра.

Кроме системных библиотек, в состав ОС обычно входят некоторые *прикладные библиотеки*, содержащие функции, часто используемые системным и прикладным ПО. Прикладные библиотеки, в свою очередь, могут непосредственно обращаться к вызовам ядра или использовать функции системных библиотек для своих нужд. Таким образом, классификация библиотеки как системной или прикладной носит условный характер, поскольку технически они реализуются одинаково. Тем не менее, ряд наиболее важных библиотек обычно однозначно относят к системным.

Системные и прикладные библиотеки технически могут быть реализованы как *статически связываемые* или как *разделяемые* (динамически связываемые). Статическая библиотека представляет собой специальный файл, содержащий объектный код функций данной библиотеки. При компиляции и последующей сборке приложения специальная программа *компоновщик*, или *линкер* (англ. link — связывать), выполняет компоновку исполняемого файла из объектного кода самой программы и используемых в ней функций, извлекая объектный код последних из файлов библиотек. После этого исполняемый файл приложения становится уже независимым от файлов статических библиотек, поскольку сам содержит необходимый код.

Формат файлов статических библиотек зависит не только от самой ОС, но и от применяемых компиляторов. В ОС Linux статическая библиотека представляет собой архивный файл в формате специального архиватора `ar`,

включающий один или более объектных файлов, полученных в результате компиляции исходного кода библиотеки компилятором `gcc`.

Разделяемая библиотека представляет собой особый файл — объектный модуль, также содержащий код функций библиотеки, но загружаемый в оперативную память вместе с исполняемым файлом приложения. В процессе загрузки приложения ОС производит связывание объектного кода исполняемого файла и библиотеки. Смысл разделяемой библиотеки состоит в том, что она загружается в оперативную память только один раз при первой необходимости. Все другие процессы, для работы которых также необходима данная библиотека, будут использовать один и тот же её экземпляр, находящийся в оперативной памяти. Таким образом, преимущество разделяемых библиотек заключается в том, что экономится место и на носителях информации, и в оперативной памяти. При этом в программах отсутствует повторяющийся код разделяемых функций, а в оперативной памяти эти функции присутствуют только в одном экземпляре.

Формат и технология динамического связывания разделяемых библиотек с приложениями в разных ОС сильно отличаются. На данный момент во многих реализация ОС UNIX, включая Linux и Solaris, для разделяемых библиотек и исполняемых файлов применяется формат ELF (англ. Executable and Linkable Format — формат исполняемых и компонуемых файлов). В ОС UNIX имена файлов разделяемых библиотек содержат суффикс `.so` (от англ. shared object — разделяемый объект) и хранятся, как правило, в каталогах `/lib` и `/usr/lib` (см. п. 4.6). В ОС Windows аналогом разделяемых библиотек являются DLL-библиотеки (от англ. Dynamic Link Library — динамически подгружаемая библиотека). Особенности динамического связывания описаны в [1, 3].

В состав ОС входит также *системное ПО*, предназначенное для выполнения основных задач по обслуживанию и администрированию ОС. Как правило, к системному ПО традиционно относят:

- утилиты настройки оборудования;
- утилиты настройки сетевых соединений;
- программы управления процессами;
- программы для работы с файловой системой (разметка диска и разделов,

монтирование, резервное копирование и пр.);

- программы управления контролем доступа и учетными записями пользователей;
- утилиты управления программным обеспечением;
- службы системного назначения, включая файловые серверы и удаленный сетевой доступ;
- командные оболочки — интерпретаторы команд, предоставляющие базовый интерфейс для пользователя.

В ОС UNIX к системному ПО традиционно также относят средства разработки приложений, включая компилятор языков Си/Си++, ассемблер, компоновщик, отладчик, заголовочные файлы стандартной библиотеки и пр., т.к. обычно они поставляются вместе с ОС.

Командные оболочки (англ. термин shell — оболочка) представляют собой интерактивные интерпретаторы команд и обеспечивают интерфейс общения пользователя с ОС. Любая современная ОС имеет хотя бы одну базовую командную оболочку, через которую обеспечивается запуск любых системных программ и приложений пользователя. Как правило, командная оболочка имеет также возможность пакетной обработки команд, указанных в текстовом файле, именуемом *сценарием*. Подробнее командная оболочка ОС UNIX будет рассмотрена в п. 9.2.

1.3. Программный интерфейс

Программным интерфейсом называется совокупность функций, определенных пользовательских типов данных, классов и констант какой-либо библиотеки или программы, предназначенной для использования в других приложениях. Таким образом, любая библиотека по определению предоставляет некоторый программный интерфейс. В зарубежной литературе и технической документации традиционно используется термин *API* (от англ. Application Programming Interface — интерфейс программирования приложений), который также будет использоваться далее.

Как было сказано выше, ядро ОС предоставляет набор системных вызовов, оформляемый, как правило, в виде одной или нескольких системных библиотек. Таким образом, можно говорить о *программном интерфейсе системных*

Таблица 1. Структура справочной системы man ОС UNIX. Стандартные разделы.

Раздел	Назначение
1	Исполняемые программы или команды оболочки (shell)
2	Системные вызовы (функции, предоставляемые ядром)
3	Библиотечные вызовы (функции, предоставляемые программными библиотеками)
4	Специальные файлы (обычно находящиеся в каталоге /dev)
5	Форматы файлов и соглашения, например о /etc/passwd
6	Игры
7	Разное (включает пакеты макросов и соглашения), например man(7), groff(7)
8	Команды администрирования системы (обычно, запускаемые только суперпользователем)

вызовов или об *API ядра*. Совокупность всех системных и часто используемых прикладных библиотек создает некий объединенный API, который является отличительной чертой каждой ОС. Так, в литературе можно встретить термины «UNIX API» или «Windows API»[1], которые уже сразу предполагают определенный набор функций и даже стиль разработки системного или прикладного ПО.

В некотором смысле к программному интерфейсу можно отнести также набор основных системных и прикладных команд, используемых в командных оболочках. Особенно это справедливо для ОС UNIX, где ассортимент команд очень обширен, а интерфейс пользователя в виде командной строки позволяет решать любым задачи по управлению и администрированию ОС.

Программный интерфейс любой библиотеки или системы должен быть документирован. В ОС UNIX имеются давние традиции по интеграции справочной системы в ОС в виде четко структурированного каталога страниц (файлов) документации, распределенных по нескольким разделам. В табл. 1 приведена традиционная структура справочной системы ОС UNIX. Получение справки в командной строке осуществляется командой `man` (от англ. manual — руководство) с указанием названия страницы (функции, команды), а также необязательным указанием раздела.

[1] Традиционно API 32-разрядных версий ОС Windows именуется как Win32API, так же называется и основной заголовочный файл системной библиотеки.

1.4. Технологии построения ядра

Существует несколько технологий построения ядра. Отметим две базовые концепции: *монолитную* и *микроядерную*.

Монолитное ядро представляет собой наиболее распространенную, классическую технологию построения ядра ОС, когда все функции ядра ОС реализуются фактически в одной программе. Все части монолитного ядра работают в одном адресном пространстве с максимальным уровнем привилегий (см. гл. 3). Такой подход в проектировании ядра ОС позволяет достичь максимального быстродействия. Однако недостатком монолитного подхода является необходимость перекомпиляции ядра при малейших изменениях в функциональности (например, при добавлении или удалении драйвера устройства).

Модульное ядро — современная, усовершенствованная модификация архитектуры монолитных ядер. В отличие от классических монолитных ядер, считающихся ныне устаревшими, модульные ядра обычно не требуют полной перекомпиляции ядра при изменении состава аппаратного обеспечения компьютера. Вместо этого модульные ядра предоставляют механизм подгрузки *модулей ядра*, необходимых для функционирования какого-либо аппаратного обеспечения. Типичным примером модулей ядра могут служить *драйверы устройств*, обеспечивающие интерфейс ОС и конкретного типа оборудования, модули поддержки различных файловых систем, модули поддержки сетевых протоколов и пр. Подгрузка модулей может быть как динамической, выполняемой «на лету» без перезагрузки ОС, так и статической, выполняемой при перезагрузке ОС после переконфигурирования системы для загрузки тех или иных модулей. Все модули ядра работают в адресном пространстве ядра и могут пользоваться всеми функциями, предоставляемыми ядром. Поэтому модульные ядра продолжают оставаться монолитными.

Модульные ядра предоставляют специальный API для связывания модулей с ядром, для обеспечения динамической подгрузки и выгрузки модулей. В свою очередь, не любая программа может быть сделана модулем ядра. На модули ядра накладываются определенные ограничения в части используемых функций. Например, они не могут пользоваться функциями стандартных библиотек Си/Си++ и должны использовать специальные аналоги, являющиеся функциями API ядра. Кроме того, модули ядра обязаны экспортировать

определенные функции, нужные ядру для правильного подключения и распознавания модуля, для его корректной инициализации при загрузке и корректного завершения при выгрузке, для регистрации модуля в таблице модулей ядра и для обращения из ядра к сервисам, предоставляемым модулем. Не все части ядра ОС могут быть сделаны модулями. Некоторые части ядра всегда обязаны присутствовать в оперативной памяти и должны быть жестко внедрены в ядро. Примерами современных ОС с модульной архитектурой ядра являются ОС Linux, Solaris, FreeBSD, NetBSD, OpenBSD и многие другие.

Отметим основные достоинства монолитной архитектуры ядра:

- высокое быстродействие из-за отсутствия задержек при взаимодействии между частями ядра, поскольку не требуется переключения из привилегированного режима в пользовательский и обратно, а также отсутствуют временные затраты на средства межпроцессного взаимодействия (см. гл. 8);
- относительно простое проектирование.

Можно отметить следующие недостатки монолитного ядра:

- вероятность сбоя всего ядра (и всей ОС) при нарушении правильной работы одной из его частей;
- необходимость перекомпиляции всего ядра при добавлении новой функции (например, драйвера устройства) — недостаток частично устраняется с помощью модулей;
- достаточно большой объем ядра, и, как следствие, малая пригодность к применению во встраиваемых системах с ограниченным объемом ОЗУ;
- сложность отладки, добавления новых или удаления старых функций.

Микроядро — это ядро с минимальным набором функций ОС, обеспечивающих управление памятью и процессами, при этом все остальные функции реализуются в виде отдельных процессов. Классическое микроядро предоставляет лишь очень небольшой набор низкоуровневых примитивов, или системных вызовов, реализующих базовые сервисы операционной системы, к которым относятся:

- управление оперативной памятью, включая виртуальную память;
- управление процессами и вычислительными потоками (нитями);

- средства межпроцессного взаимодействия.

Все остальные функции ОС, которые в монолитных ядрах предоставляются непосредственно ядром, в микроядерных архитектурах реализуются в адресном пространстве пользователя и называются *сервисами*. Примерами таких сервисов являются стек сетевых протоколов, файловая система, драйверы аппаратуры.

Основное достоинство микроядерной архитектуры — высокая степень модульности ядра ОС, что существенно упрощает добавление в него новых компонентов. В микроядерной ОС можно, не прерывая ее работы, загружать и выгружать драйверы, файловые системы и т. д. Существенно упрощается и процесс отладки компонентов ядра, так как новая версия драйвера может загружаться без перезапуска всей ОС. Компоненты ядра ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства. Микроядерная архитектура повышает надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра. Теоретически небольшое микроядро может уместиться в кэше процессора, что значительно увеличит быстродействие системы.

В то же время, микроядерная архитектура вносит дополнительные накладные временные расходы, связанные с постоянным взаимодействием всех компонентов ОС (процессов) посредством передачи сообщений, что отрицательно влияет на производительность. Для того чтобы микроядерная ОС по скорости не уступала ОС с монолитным ядром, требуется очень тщательное проектирование всех компонентов системы, минимизируя взаимодействие между ними.

Отметим основные достоинства микроядерной архитектуры:

- устойчивость к сбоям оборудования, ошибкам в компонентах системы;
- высокая степень модульности и простота добавления или удаления новых сервисов и драйверов;
- компактность ядра и возможность работы во встраиваемых системах.

Можно отметить основные недостатки микроядра:

- снижение производительности из-за накладных расходов по переключению и передаче данных между процессами-сервисами;

- необходимость очень тщательного проектирования.

Примерами ОС с микроядерной архитектурой являются: ОС реального времени (ОСРВ) QNX, свободная UNIX-подобная ОС MINIX, ChorusOS, а также основанные на микроядре Mach ОС GNU/Hurd и Mac OS X.

1.5. Пользовательская среда

На раннем этапе своего развития ОС UNIX, впрочем, как и многие другие ОС тех лет, не имела графического интерфейса. Все программы и пользовательская среда в целом проектировались с расчетом на терминальный режим работы. Таким образом, основным интерфейсом пользователя в ОС UNIX стал интерфейс командной строки (англ. CLI — Command Line Interfacea), реализуемый командной оболочкой и терминалом[1]. Пользователь набирает в алфавитно-цифровом терминале команды, и после нажатия клавиши `Enter` команда отправляется на выполнение. Результат работы команды (программы) в текстовом формате возвращается в терминал. Для пользователя ОС Windows, привыкшего к запуску приложений щелчком клавиши мыши по иконке рабочего стола, запуск приложения в ОС UNIX набором имени его исполняемого файла в командной строке может показаться довольно необычным процессом, хотя и в ОС Windows имеется такая возможность. Интерфейс командной оболочки UNIX всегда ориентировал пользователя на более детальное изучение и более глубокое понимание архитектуры этой ОС.

В командной оболочке многозадачность ОС UNIX можно использовать, применяя фоновый режим работы процессов или несколько терминалов (включая виртуальные терминалы). При этом пользователь должен всегда иметь «под рукой» подробное справочное руководство по системе и командам (программы текстового режима с меню — большая редкость). ОС UNIX всегда была хорошо документирована и имела развитую справочную систему. Несмотря на аскетичность своего интерфейса, система `man` и по сей день является актуальной, хотя к ней уже разработаны программы с графическим интерфейсом, например, программа `xman`.

Другой особенностью ОС UNIX является хранение практически всей конфигурационной и настроечной информации по системе и прикладным программам в текстовых файлах, расположенных в строго определенных ка-

[1] Подробно интерфейс пользователя рассматривается в гл. 9.

талогах файловой системы. Для удобства работы эти файлы, как правило, содержат комментарии, а также подробно документируются в системе `man`. Поэтому второй по частоте использования программой после `man` в ОС UNIX можно назвать текстовый редактор.

Далее в пособии будут детально описаны теоретические аспекты архитектуры ОС UNIX. Изложение материала сопровождается кратким описанием необходимых для работы команд оболочки и функций стандартной библиотеки Си. Для практического ознакомления с ОС Linux и Solaris рекомендуются источники [4–14].

2. Обзор операционных систем

2.1. Классификация ОС

В общем случае, классификация ОС не может быть проведена однозначно, поскольку современная ОС представляет собой очень большой набор ПО, решающий огромный круг задач в вычислительной системе. Тем не менее, классификацию ОС можно провести по следующим признакам:

- *архитектура ядра*: а) монолитное ядро, б) модульное ядро, в) микроядро, г) гибридное ядро;
- *способ управления памятью*: а) открытое линейное пространство, б) виртуальная память (сегментная или страничная организация), в) виртуальная память с защитой;
- *поддержка многозадачности и разделения времени*: а) однозадачные ОС, б) невытесняющая многозадачность, в) вытесняющая многозадачность;
- *способ разделения времени*: а) обычные ОС, б) ОС реального времени (мягкого и жесткого РВ);
- *наличие системы контроля доступа*: а) однопользовательские, б) многопользовательские;
- *наличие сетевой подсистемы*: а) ОС без поддержки сети, б) сетевые ОС.
- *область применения*: а) рабочие станции, б) серверы, в) встраиваемые системы, г) мобильные устройства.

Далее будет кратко рассмотрена история создания и этапы развития различных ОС, будут рассмотрены различные концепции и семейства современных систем.

2.2. Краткая история ОС

Первое поколение ЭВМ на радиолампах и полупроводниковых диодах как таковой ОС не имело (в СССР: М-20, МЭСМ-1, БЭСМ-1, БЭСМ-2, БЭСМ-3М, БЭСМ-4 и др.; в Германии: Z3, Z4; в США: ENIAC, UNIVAC I, IBM-701, IBM-650 и др.). Исключением, вероятно, можно считать ЭВМ IBM 704, производимую в США в 1954–1957 гг., для которой в 1955 г. была разработана

достаточно развитая ОС [2].

Свой отсчет история ОС начала с появлением ЭВМ *второго поколения*, выполненных на полупроводниковых транзисторах.

В СССР в 1959–1964 гг. под руководством Полина В.С. разрабатывалась ЭВМ «Весна» [2,15]. ЭВМ выпускалась на Минском заводе до 1972 г., всего выпущено 19 машин. Первый экземпляр машины поступил в вычислительный центр Министерства обороны СССР. Применялась она также в Гидрометцентре СССР и в др. учреждениях. ЭВМ содержала 80 тыс. транзисторов, 200 тыс. диодов. В её состав входили: центральный (48-битный) и периферийный процессоры с тактовой частотой 5 МГц, что обеспечивало быстродействие до 250 тыс. операций в секунду. Память ЭВМ включала 32 быстродействующих регистра, ОЗУ формата 2 x 1024 чисел (время доступа 1 мкс), ОЗУ формата 4 x 16384 чисел (время доступа 10 мкс). Впоследствии выпускался также сокращенный вариант системы — ЭВМ «Снег» (главный конструктор — Левин В.К.). ЭВМ «Весна» имела ОС со следующими функциями:

1) управление памятью и устройствами ввода-вывода,
2) аппаратная защита памяти,
3) многозадачность,
4) работа с несколькими пользователями через терминалы.

В конце 1966 г. в СССР была разработана первая супер-ЭВМ второго поколения на транзисторах БЭСМ-6, которая являлась самой высокопроизводительной ЭВМ в Европе в течение нескольких лет. Главным конструктором БЭСМ-6 был Сергей Алексеевич Лебедев (1902–1974), основоположник вычислительной техники в СССР, директор ИТМиВТ (Института точной механики и вычислительной техники АН СССР), академик АН СССР (1953). В период с 1968 по 1987 гг. на заводе счётно-аналитических машин (САМ) в Москве было произведено 355 машин БЭСМ-6.

БЭСМ-6 обладала следующими техническими характеристиками:

- элементная база: транзисторный парафазный усилитель с диодной логикой на входе;
- процессор: конвейерный центральный процессор (ЦП) с отдельными конвейерами для устройства управления (УУ) и совмещенного арифметического устройства (АУ) для целой и плавающей арифметики;

- тактовая частота: 10 МГц;
- разрядность: 48 бит;
- быстродействие: около 1 млн. операций в секунду;
- память: 32 768 слов, 8-слойная физическая организация памяти, виртуальная адресация памяти и расширяемые регистры страничной приписки; фактически реализована виртуальная память со страничной организацией (по 1024 48-разрядных слов) (см. гл. 3);
- кэш: 16 x 48-битных слов (4 - чтения данных, 4 - чтения команд, 8 - буфер записи);
- система команд: 50 24-битных команд (по две в слове);
- развитая система прерываний.

Для БЭСМ-6 были разработаны несколько ОС:

- ОС «Диспетчер-68» (Д-68) (Королёв Л.Н., Иванников В.П., Томилин А.Н., 1967 г.), имевшая следующие функции: 1) мультипрограммный режим пакетной обработки заданий, 2) управление виртуальной памятью, 3) управление внешними запоминающими устройствами, 4) управление устройствами ввода-вывода;
- «Новый диспетчер-70», 1970 г. (НД-70), имевшая следующие функции: 1) возможность организации параллельных вычислений, 2) работа в реальном времени, 3) работа в составе многомашинного комплекса (масштабируемость);
- ОС «Дубна» (Говорун Н.Н., Силин И.Н.,1970 г.);
- ОС «Диспак» (Тюрин В.Ф., 1971 г.), имевшая следующие функции: 1) пакетный режим, 2) диалого-пакетный режим, 3) режим разделения времени.

В конце 1960 г. в США был разработан *суперкомпьютер CDC 1604*. Главным конструктором ЭВМ стал американский инженер Сеймур Роджер Крэй (англ. Seymour Roger Cray) (1925–1996), который впоследствии разработал ряд других всемирно известных суперкомпьютеров. К 1964 г. компанией CDC (Control Data Corporation) было создано 50 машин CDC 1604, которые в основном поставлялись в ВМФ США. Данная ЭВМ имела память в 32768 48-битных слов, систему из 24-битных команд (по две в слове), уступала по

быстродействию БЭСМ-6 в 10 раз и как таковой ОС не имела.

С 1964 г. компанией CDC выпускался *суперкомпьютер CDC 6600*, разработанный группой из 30 инженеров. Первая машина была поставлена в Европейский Центр ядерных исследований (CERN). Она имела технические характеристики, по уровню схожие с БЭСМ-6:

- элементная база: планарные кремниевые транзисторы компании Fairchild Semiconductor;
- центральный процессор с конвейером команд, реализующий логические и арифметические операции;
- 10 периферийных процессоров;
- тактовая частота: 10 МГц;
- разрядность: 60 бит, 12 бит для периферийных процессоров;
- быстродействие: около 1 млн. операций в секунду.

Для CDC 6600 были разработаны несколько ОС:

- SCOPE OS, имевшая такие функции как: 1) поддержка файловой системы на внешних носителях и в ОЗУ, 2) контроль выполнения задач, 3) управление памятью (загрузка сегментов и оверлеев), 4) операции ввода-вывода, 5) ведение логов, отладчик, 6) запуск компиляторов и отладчиков;
- KRONOS, имевшая такие функции как: 1) разделение времени, 2) поддержка различных ASCII-терминалов, 3) аутентификация пользователей
- сетевая ОС NOS.

На рубеже 1960-х и 1970-х гг. появляется *третье поколение* ЭВМ на основе интегральных микросхем. В 1964 г. американская компания IBM анонсировала универсальную компьютерную систему IBM System/360, особенностью которой стал модульный подход, открытость и масштабируемость, а также разделение понятий «архитектура» и «реализация». Такие ЭВМ стали называться термином «мэйнфрейм» (англ. mainframe), который происходит от названия типовых процессорных стоек этой системы. Вплоть до начала 1980-х гг. IBM/360 была лидером на рынке мэйнфреймов. В СССР выпускался программно- и частично аппаратно-совместимый аналог этой системы в виде серии ЕС ЭВМ («Единая система ЭВМ») [16].

Благодаря широкому распространению IBM/360, применяемые в ней 8-разрядные ячейки памяти, которые сегодня мы называем *байтами*, и шестнадцатеричная система счисления стали стандартом для всей компьютерной техники. ЭВМ IBM/360 была первой 32-разрядной компьютерной системой. Старшие модели семейства IBM/360 и последовавшее за ними семейство IBM/370 были одними из первых компьютеров с виртуальной памятью и первыми серийными компьютерами, поддерживающими реализацию *виртуальных машин* [17].

Первыми ОС для данного семейства ЭВМ в 1966 г. стали PCP (англ. Primary Control Program — первичная управляющая программа) и DOS/360 (англ. Disk Operating System — дисковая ОС). Архитектура обеих ОС была типична для вычислительных систем второго поколения — это были пакетные мониторы, рассчитанные на работу одной прикладной программы без защиты памяти [1]. В 1967 г. были выпущены две версии PCP:

- MVT (англ. Multiprogramming with a Variable number of Tasks — многопрограммная [система] с переменным числом задач),
- MFT (англ. Multiprogramming with a Fixed number of Tasks — то же, но с фиксированным числом задач).

Обе системы реализовывали вытесняющую многозадачность в едином адресном пространстве, при этом MFT использовала разделы памяти, а MVT — относительную загрузку с базовым регистром. Позднее, к MVT была добавлена подсистема работы с несколькими терминалами в режиме разделения времени TSO (англ. Timesharing Option — возможность разделения времени), ASP (англ. Asymmetric Multiprocessing System — асимметричная многопроцессорность) и ряд других прикладных подсистем [1].

В 1972 г., после появления мэйнфреймов System/370 с диспетчером памяти, была выпущена переходная система OS/SVS (англ. Single Virtual Storage — единая виртуальная память), которая позволяла использовать страничную подкачку, но не имела механизма защиты заданий друг от друга.

Наконец, в 1974 г. была выпущена MVS (англ. Multiple Virtual Storage — множественная виртуальная память), которая предоставляла каждой задаче собственное виртуальное адресное пространство объемом до 2 Гбайт (в System/360 и первых моделях System/370 адрес был 24-разрядным). Боль-

шая часть дополнительных подсистем MVT была включена в стандартную поставку MVS. MVS имела следующие функциональные особенности [1]:

- раздельные адресные пространства (см. гл. 3);
- пакетный и интерактивный (разделение времени) запуск задач;
- вытесняющую многозадачность;
- выполнение вычислительных тредов в общем адресном пространстве (см. гл. 7);
- многопоточное ядро;
- примитивы взаимоисключения — замки (lock) (см. гл. 8);
- симметричную многопроцессорность;
- динамическое подключение и отключение процессоров (как центральных, так и канальных) и их горячую замену;
- библиотеки — аналог вложенных каталогов;
- последовательные, блочные и индексно-последовательные файлы;
- отображение файлов, в том числе и индексно-последовательных, в адресное пространство (VSAM);
- систему безопасности на основе ACL (см. гл. 6);
- развитые средства системного мониторинга;
- сетевые средства на основе стека протоколов SNA (англ. Standard Network Architecture).

В середине 1990-х гг. вышла новая версия системы — OS/390, пришедшая на смену MVS. В OS/390 были добавлены следующие возможности:

- поддержка многомашинных кластеров (Parallel Sysplex);
- развитие сетевых средств равноправного (peer-to-peer) взаимодействия;
- поддержка сетевых протоколов семейства TCP/IP;
- совместимость с ОС UNIX.

В 1999 г., в связи с началом выпуска 64-разрядного семейства компьютеров z900, вышла 64-разрядная версия системы — z/OS. Системы под управлением OS/390 и z/OS применяются главным образом в качестве серверов транзакций и СУБД масштаба предприятия и составляют становой хребет вычислительных систем большинства крупных компаний [1].

Большое влияние на развитие вычислительной техники и операционных систем оказала американская компания DEC (англ. Digital Equipment Corporation). Особенностью первых компьютеров фирмы DEC являлось их обозначение PDP (англ. Programmable Data Processor — программируемый обработчик данных): компания решила не употреблять слово «компьютер», которое в те года предполагало огромный и дорогой комплекс, требующий отдельного помещения и солидного обслуживающего персонала. Отметим наиболее значимые первые модели PDP:

- PDP-1 — 18-разрядный компьютер, 1960 г.; использовался в основном для разработки операционных систем с разделением времени. На этом компьютере были реализованы первый текстовый процессор (TJ-2) и первая видеоигра (Spacewar);
- PDP-6 — 36-разрядный компьютер, имел диспетчер памяти, 1964 г.;
- PDP-10 — 36-разрядный компьютер с набором инструкций PDP-6, 1968 г.

Компания DEC известна также тем, что создала самые популярные в те годы терминалы для работы с ЭВМ — VT52, VT100, VT200 и другие, ставшие стандартом в своей области.

Для компьютеров PDP-6 была разработана ОС PDP-6 Monitor с открытым исходным кодом на ассемблере, включающая интерпретатор команд CONSOLE, который являлся частью ядра ОС, а не отдельной программой. Компанией BBN Technologies для компьютеров PDP-6 и PDP-10 была разработана ОС TOPS-10. TOPS-10 известна как первая массовая ОС, поддерживавшая разделение времени [1]. Вскоре после создания TOPS-10 компанией BBN Technologies была разработана ОС TENEX для PDP-6, использующая диспетчер памяти. Для использования этой ОС на PDP-10 компанией также был разработан собственный страничный диспетчер памяти. ОС TENEX имела механизм управления виртуальной памятью, собственный командный язык, развитую файловую систему со возможностью указания версий файлов. Вскоре компания DEC стала встраивать страничный диспетчер памяти в новые компьютеры DEC-20, под которые компанией BBN Technologies была выпущена ОС TOPS-20.

Во второй половине 1960-х гг. компания DEC разработала серию компьютеров PDP-11 с байтовой адресацией и разрядностью шины 16 бит. Особен-

ностью компьютеров были относительно небольшие габариты и невысокая стоимость, что сделало PDP-11 самыми популярными из серии PDP. Младшие модели серии не имели диспетчера памяти, и их физическое адресное пространство совпадало с логическим. Старшие модели были оснащены сегментным диспетчером памяти с раздельными адресными пространствами для пользовательского и системного режимов. В СССР производились совместимые с PDP-11 ЭВМ серии «СМ».

Для архитектуры PDP-11, просуществовавшей более двух десятилетий, было разработано несколько ОС. Одной из первых удачных ОС была RT-11, представлявшая собой компактную однопользовательскую ОС с простой файловой системой и асинхронным вводом-выводом. Существовала версия системы, которая поддерживала кооперативную многозадачность для двух задач [1].

Первые версии системы поставлялись с командным интерпретатором, схожим с TOPS-10. В начале 1980-х гг. для RT-11 был реализован новый командный интерпретатор. В его основу был положен язык DCL (DEC Command Language) — командный язык, разработанный DEC на основе командного языка ОС TENEX и использовавшийся в последующих ОС RSX-11 и VMS [1]. Примечательно, что язык командных интерпретаторов `command.com` для ОС DOS и Windows 9x и `cmd.exe` для ОС OS/2 и Windows NT основан на версии языка DCL.

В 1969 г. DEC разработала 18-разрядный компьютер PDP-15, реализованный на интегральных микросхемах. Компьютер использовался во многих системах реального времени. Для него была разработана ОС RSX-15 — многозадачная система с незащищенной памятью, развитой системой приоритетов процессов и асинхронным вводом-выводом.

Впоследствии стал разрабатываться аналог ОС RSX-15 и для компьютеров PDP-11. Наиболее удачной завершенной версией стала ОС RSX-11/M, которая представляла собой многозадачную ОС разделенного времени, способную работать на процессорах с диспетчером памяти. В ней были реализованы сегментная подкачка, разделяемые библиотеки, режим реального времени. ОС RSX-11/M поддерживала несколько терминалов и была в полной мере многопользовательской: для терминальных сессий и каждого процесса сохра-

нялся идентификатор пользователя. ОС имела сложную файловую систему с поддержкой вложенных каталогов и версий файлов. В качестве командного языка применялся DCL. Примечательно, что такая функциональная ОС предъявляла минимальные требования к объему ОЗУ в 32 кбайт [1].

В 1975 г. компания DEC начала разрабатывать 32-разрядную ЭВМ, которая должна была стать последователем серии PDP-11. В 1977 г. был представлен компьютер VAX (от англ. Virtual Address Extended — расширенный виртуальный адрес). Наиболее важной его особенностью, унаследованной от PDP-11, была *ортогональная система команд* процессора[1]; количество регистров процессора было увеличено с 8 до 16, теперь они допускали и операции с плавающей точкой. Архитектура VAX обеспечивала совместимость ПО с PDP-11 на уровне кодов ассемблера, а также предусматривала запуск отдельных задач в режиме бинарной эмуляции PDP-11 [1].

Основной ОС для VAX стала VAX/VMS (от англ. Virtual Memory System — система с виртуальной памятью), архитектура которой во многом была аналогична RSX-11. Перечислим отличительные черты ОС VMS:

- страничная организация виртуальной памяти;
- журналируемая файловая система;
- имена файлов формата 32.32 (максимум 32 символа для имени и для расширения) из ASCII символов.

В первой половине 1980-х гг. для ОС VAX/VMS и RSX-11 был реализован стек сетевых протоколов DECNet, предоставляющий возможности обмена почтой, сетевую файловую систему, удаленный терминальный доступ, а также *кластеризацию* — общую сетевую базу данных пользовательских учетных записей. Во второй половине 1980-х гг. для ОС VAX/VMS были также реализованы стек протоколов TCP/IP и графическая подсистема X Window, которые в то время стали уже стандартом для ОС UNIX [1].

В начале 1990-х гг. компания DEC, чтобы не отставать от конкурентов, стала разрабатывать однокристальные 64-битные RISC-процессоры Alpha. Для новой аппаратной платформы была разработана новая ОС OpenVMS, обеспечивающая полную совместимость с ОС VAX/VMS на уровне исходных кодов.

[1] В ортогональной системе команд все команды процессора допускают использование во всех операндах всех его регистров и режимов адресации.

Впоследствии ОС OpenVMS также была портирована на 64-битную платформу Intel Itanium. Сегодня ОС OpenVMS владеет компания VMS Software, Inc., а сама ОС существует для платформ DEC VAX, DEC Alpha, Intel Itanium и Intel x86. Отметим основные особенности современной версии OpenVMS:

- высокая степень отказоустойчивости,
- возможность применения в системах реального времени,
- высокая степень безопасности и защищенности.

Среди заказчиков OpenVMS, в том числе отечественных, преобладают оборонные структуры и банки, телекоммуникационные компании, предприятия непрерывного цикла, такие как АЭС.

2.3. UNIX-совместимые ОС

Первая версия ОС UNIX была разработана в 1969 г. в подразделении Bell Telephone Laboratories (Bell Labs) американской телекоммуникационной компании AT&T[1] Кеном Томпсоном (Ken Tompson) и Дэннисом Ритчи (Dennis Ritchie) на основе ОС Multics, созданной ранее Bell Labs совместно с General Electric Company [1,3]. ОС разрабатывалась на языке ассемблер для компьютера DEC PDP-7. К 1972 г. ядро ОС было переписано на языке Си для компьютера DEC PDP-11. Впоследствии бóльшая часть системы стала разрабатываться на языке Си, который был создан самими авторами ОС UNIX.

Применение высокоуровневого языка программирования Си позволяло легко переносить ОС UNIX на другие аппаратные платформы, что способствовало быстрому распространению системы. ОС UNIX тесно связана с языком программирования Си, влияние которого встречается в системе практически везде. Большой вклад в развитие системы внес также Брайан Керниган (Brian Kernighan), ставший соавтором всемирно известных книг по программированию на языке Си, он же придумал название данной ОС [3].

В 1973 г. дочерняя компания AT&T выдала разрешение на некоммерческое использование ОС UNIX, и она начала распространяться в университетах США. В университете Беркли, штат Калифорния, было создано специальное подразделение для развития и распространения ОС — Berkeley Software

[1] American Telephone and Telegraph (AT&T Corp.) — одна из крупнейших американских телекоммуникационных компаний, основана в 1885 г.

Distribution (BSD). В 1978 г. вышла первая версия ОС BSD, которая породила одну из самых крупных ветвей в генеалогии ОС UNIX. В ОС BSD UNIX появился целый ряд нововведений [1]:

- сегментная (на старших моделях PDP-11) и страничная (на VAX-11/780) виртуальная память,
- раздельные адресные пространства процессов и выделенное адресное пространство ядра,
- абсолютные загрузочные модули формата a.out,
- примитивная форма разделяемых библиотек,
- усовершенствованный механизм обработки сигналов,
- управление сессиями и заданиями в пределах сессии.

Самое важное нововведение было сделано в начале 1980-х гг., когда в рамках проекта DARPA[1] сетевое программное обеспечение ARPANET было перенесено с TOPS/20 на BSD UNIX. Вскоре сетевой стек BSD стал эталонной реализацией того, что ныне известно как семейство протоколов TCP/IP [1,3].

В 1980 г. компания AT&T стала предоставлять коммерческую лицензию на ОС UNIX, в результате чего стали появляться разработки системы от других компаний. Одной из первых коммерческих версий была ОС Xenix компании Microsoft. Как и BSD UNIX, ОС Xenix использовала виртуальную память и имела отдельное адресное пространство для ядра [1]. Первые версии ОС работали на компьютерах DEC PDP-7. В 1983 г. эксклюзивные права на ОС Xenix приобрела компания Santa Cruz Operation (SCO), которая с 1988 г. стала продавать ОС под маркой SCO UNIX. Во второй половине 1980-х гг. ОС SCO UNIX была перенесена на платформу Intel 80386 и стала одной из первых 32-разрядных ОС для данного процессора.

В 1984 г. в результате реорганизации AT&T компания получила возможность непосредственно заниматься коммерческой реализацией ОС UNIX. Так появилось новое ядро UNIX System V — первая поддерживаемая компанией AT&T версия ядра UNIX. В 1987 г. вышла версия UNIX System V Release

[1] DARPA (англ. Defense Advanced Research Projects Agency — агентство передовых оборонных исследовательских проектов) — агентство Министерства обороны США, отвечающее за разработку новых технологий для использования в вооружённых силах; известно разработкой сетевой технологии ARPANET — фундаментом современной сети Интернет на основе протоколов TCP/IP.

3, включавшая в себя асинхронные драйверы последовательных устройств (STREAMS), универсальный API для доступа к сетевым протоколам (TLI), средства межпроцессного взаимодействия (семафоры, очереди сообщений и сегменты разделяемой памяти, см. гл. 8), ныне известные как SysV IPC, BSD-совместимые сокеты [1, 3].

В 1980-х гг. стала набирать популярность SunOS — версия ОС UNIX компании Sun Microsystems[1], основанная на BSD UNIX. В 1987 г. AT&T и Sun Microsystems заключили стратегическое соглашение о разработке перспективного ядра UNIX System VI, которое должно было обеспечить совместимость с ОС AT&T UNIX System V, BSD UNIX и Xenix.

В итоге в 1989 г. была выпущена новая версия UNIX — *System V Release 4 (SVR4)*. Она объединила возможности нескольких версий ОС UNIX: SunOS, BSD UNIX и предыдущей версии System V. В системе появились следующие новшества [3]:

- командные оболочки Korn (`ksh`) и C (`csh`),
- символьные ссылки в файловой системе,
- система терминального ввода-вывода на основе потоков STREAMS,
- отображаемые в оперативную память файлы,
- сетевая файловая система NFS,
- система удаленного вызова процедур RPC,
- быстрая файловая система FFS,
- сетевой API сокетов,
- поддержка диспетчеризации реального времени.

ОС SunOS, начиная с версии 5.0 (1992 г.), стала основываться на UNIX SVR4 и продаваться под торговой маркой Solaris. В настоящий момент Solaris является одной из самых распространенных коммерческих реализаций ОС UNIX. В 2005 г. Sun Microsystems инициировала разработку ОС OpenSolaris — альтернативного варианта Solaris с полностью открытым и свободным исходным кодом.

В 1994 г. вышла финальная университетская версия BSD UNIX — 4.4BSD,

[1] Sun Microsystems — американская компания, производитель программного и аппаратного обеспечения, основана в 1982 г. Акроним «SUN» означает Stanford University Networks. В 2010 г. поглощена корпорацией Oracle.

после чего права на исходный код перешли компании BSDI (Berkeley Software Design, Inc). На базе BSD UNIX впоследствии было разработано несколько десятков различных версий UNIX. Перечислим наиболее распространенные версии:

- 386BSD (BSD/OS) — реализация BSD UNIX для архитектуры Intel 80386;
- FreeBSD — версия BSD UNIX с открытым исходным кодом, реализованная на различных аппаратных платформах, которая является одной из самых распространенных свободных версий ОС UNIX и разрабатывается с 1993 г. на основе ОС 386BSD;
- NetBSD — версия BSD UNIX, развивается с 1993 г. из версий 4.3BSD и 386BSD;
- OpenBSD — свободная многоплатформенная ОС, развивается с 1995 г. из версии 4.4BSD; ее основным отличием от других ответвлений от 4.4BSD является ориентированность на создание наиболее безопасной, свободной и лицензионно чистой ОС;
- DragonFly BSD — ОС с открытым кодом, возникшая в середине 2003 г. на базе FreeBSD, ориентирована на платформу x86;
- Darwin — открытая POSIX-совместимая ОС, выпущенная компанией Apple Inc. в 2000 г.

Перечислим также хорошо известные коммерческие версии ОС UNIX [3]:

- AIX (от англ. Advanced Interactive eXecutive) — ОС UNIX компании IBM, первая версия которой вышла в 1986 г. на базе System V Release 2, 4.2BSD и 4.3BSD;
- HP-UX — версия ОС UNIX компании Hewlett-Packard, развивается с 1984 г. из версии System V;
- IRIX — 64-разрядная версия ОС UNIX компании Silicon Graphics, предназначенная для аппаратной платформы MIPS, совместима с SVR4, разрабатывалась с 1983 по 2006 гг.; сегодня на компьютеры данной компании устанавливается ОС Linux.
- Tru64 UNIX (до 1998 г. — Digital UNIX) — 64-разрядная версия ОС UNIX компании DEC для платформы Alpha; на данный момент при-

надлежит компании Hewlett-Packard;

- MacOS X — UNIX-совместимая ОС компании Apple, основанная на ядре Darwin.

2.4. Стандарт POSIX

В 1984 г. ряд европейских компаний сформировал некоммерческую организацию X/Open с целью разработки общего стандартизированного набора интерфейсов ОС и создания открытых систем с минимальной стоимостью переноса приложений между платформами [3].

В 1988 г. с целью разработки стандарта в сфере ОС UNIX был создан консорциум Open Software Foundation (OSF), в который вошли компании DEC, IBM, Hewlett-Packard и др. Его деятельность началась с принятия первой версии стандарта *POSIX*[1] (от англ. Portable Operating System Interface for Computing Environment — переносимый интерфейс операционной системы для вычислительной среды). Этот стандарт нашел широкое применение во многих ОС, в том числе и с отличной от UNIX архитектурой.

Спустя два года стандарт был принят как IEEE[2] Std 1003.1-1990, известный также как POSIX.1, а в 1992 г. была разработана редакция IEEE Std 1003.2-1992 (POSIX.2). В этом же году появился документ, известный под названием Portability Guide версии 3 или XPG3, который включал POSIX.1 и стандарт на графическую систему X Window. В дальнейшем интерфейсы XPG3 были расширены, включив базовые API систем BSD и System V (SVID), в том числе и архитектуру STREAMS. В результате была выпущена спецификация, ранее известная как Spec 11/70, получившая в 1994 г. название XPG4.2.

В рамках OSF велись работы по разработке ядра UNIX-совместимой ОС, по архитектуре в целом аналогичной UNIX SVR3 (монолитное ядро с поддержкой STREAMS и некоторыми особенностями BSD). К моменту завершения разработки уже был выпущен UNIX System V Release 4.2 (Destiny), достигший всех целей, заявленных в проекте UNIX System VI, и консорциум фактически распался.

[1] Название «POSIX» было предложено Ричардом Столлманом (англ. Richard Matthew Stallman) — основоположником и пропагандистом концепции свободного программного обеспечения, распространяемого вместе с исходным кодом.

[2] IEEE (англ. Institute of Electrical and Electronics Engineers) — Институт инженеров по электротехнике и радиоэлектронике — международная некоммерческая ассоциация специалистов в области техники.

В 1996 г. объединение усилий X/Open и OSF привело к созданию консорциума The Open Group, который продолжил разработку открытых стандартов и стал проводить сертификацию в сфере информационных технологий[1].

С 1996 г. слово «UNIX» (все буквы заглавные) является зарегистрированной торговой маркой[2] консорциума The Open Group. Правом называться UNIX-системой может только ОС, прошедшая платную сертификацию в The Open Group. Большинство свободных ОС с открытым исходным кодом не проходят процедуру сертификации и, строго говоря, не являются UNIX-системами, хотя по документации они могут полностью соответствовать стандартам UNIX. Таким образом, сегодня под ОС UNIX понимается некоторый собирательный термин, указывающий на ОС с UNIX-совместимыми архитектурой и программным интерфейсом.

С 1998 г. по настоящее время развивается так называемая спецификация The Single UNIX Specification Version 3, основа которой была заложена объединенной рабочей группой *Austin Group*, в состав которой входят члены комитета IEEE Portable Applications Standards Committee (PASC), члены The Open Group и ISO[3]/IEC[4] Joint Technical Committee 1. Целью работы группы Austin Group явилось объединение существующих стандартов POSIX IEEE Std 1003.1, IEEE Std 1003.2, базовых спецификаций от The Open Group, ISO/IEC 9945-1, ISO/IEC 9945-2 [18]. В 2001 г. объединенный стандарт содержал следующие четыре части:

1) основные определения (термины, концепции и интерфейсы, общие для всех частей);
2) описание API к системным сервисам на языке Си;
3) описание интерфейса к системным сервисам на уровне командного языка и служебных программ;
4) детальное разъяснение положений стандарта, обоснование принятых решений.

[1] См. Интернет-ресурс http://www.opengroup.org/
[2] См. Интернет-ресурс http://www.unix.org/
[3] ISO (англ. International Organization for Standardization) — Международная организация по стандартизации, занимающаяся выпуском стандартов в различных областях.
[4] IEC (англ. International Electrotechnical Commission) — Международная электротехническая комиссия (МЭК).

С развитием стандарта расширилась и трактовка термина «POSIX». Первоначально он относился к документу IEEE Std 1003.1-1988, описывающему только API. После стандартизации интерфейса на уровне командного языка и служебных программ под словом «POSIX» стал пониматься стандарт в целом, обозначая перечисленные выше части 2 и 3 через POSIX.1 и POSIX.2 в соответствии с нумерацией документов IEEE и ISO/IEC. В 2004 г. результатом работы группы стала редакция стандарта IEEE Std 1003.1-2004 [18, 19]. На сегодня «POSIX» является зарегистрированным товарным знаком IEEE.

Основной целью разработки стандарта POSIX было обеспечение совместимости различных версий ОС UNIX и переносимости прикладных программ на уровне исходного кода. Впоследствии стандарт также коснулся и командной оболочки. В состав системы документации `man` современных версий ОС UNIX, в дополнение к официальной документации разработчика дистрибутива ОС, стали входить альтернативные разделы, в которых описываются функции стандартной библиотеки Си и основные команды оболочки согласно спецификации стандарта POSIX.

Стандарт POSIX — весьма обширный и многогранный документ, где подробно рассматриваются спецификации различных компонентов ОС или концепций информационных систем. В табл. 2 приведены основные проекты спецификаций, входящие в состав стандарта.

Среди разрабатываемых сегодня ОС стандарту POSIX соответствуют следующие наиболее распространенные системы: BSD/OS, HP-UX, AIX, LynxOS, MacOS X, IPhone OS, Oracle Solaris, OpenSolaris, OpenVMS. Следующие ОС можно считать POSIX-совместимыми[1]: BeOS, FreeBSD, NetBSD, Nucleus RTOS, OpenBSD, Symbian OS.

Отметим также достаточно распространенную ОС QNX — систему реального времени на основе микроядерной архитектуры, разработка которой началась в 1982 г. канадской компанией Quantum Software Systems. В начале 1990-х гг. компания была переименована в QNX Software Systems (QSS). Начиная с версии QNX4 (середина 1990-х гг.) ОС стала соответствовать стандарту POSIX. С 2010 г. компания QSS стала подразделением канадской телекоммуникационной компании BlackBerry, известной своими одноименными

[1]Официально не сертифицированные как POSIX-совместимые, но по большей части соответствующие стандарту.

Таблица 2. Основные компоненты стандарта IEEE POSIX [19].

Проект	Описание
P1003.0	Руководство по окружениям открытых систем POSIX
P1003.1,.1a	Системные интерфейсы
P1003.1b,.1d	Подсистема реального времени
P1003.1c	Механизм нитей (программных потоков)
P1003.1e	API безопасности
P1003.1f	Прозрачный доступ к файлам
P1003.1g	Протоколо-независимые сетевые спецификации
P1003.2, .2b	Оболочка и утилиты
P1003.2c	Утилиты безопасности
P1003.2d	Расширения для пакетной обработки
P1003.10	Профиль для суперкомпьютерных окружений
P1003.13	Профили реального времени
P1003.14	Мультипроцессирование
P1003.16	Связывание для языка Си
P1003.18	Профиль POSIX-платформы
P1003.21	Связь распределенных систем реального времени
P1003.22	Руководство по основам безопасности окружений открытых систем POSIX
P1201.1	Унифицированный API для графического пользовательского интерфейса
P1201.2	Управляемость пользовательским интерфейсом
P1224	API для стандартов OSI, манипулирование абстрактными данными
P1238.0	OSI API для функций общей поддержки
P1327	OSI API языка Си для абстрактных манипуляций данными
P1387	Системное администрирование
P2003.n	Методы тестирования

смартфонами. Сегодня ОС продается под названием QNX Neutrino RTOS и является формально шестой версией ОС QNX. Отличительной особенностью системы является высочайшая стабильность в работе и очень компактное микроядро, способное функционировать во многих встраиваемых системах на базе аппаратных платформ Intel x86, ARM и других.

Ярким примером свободной POSIX-совместимой ОС является MINIX — микроядерная ОС, разработка которой началась в 1987 г. профессором Амстердамского свободного университета Эндрю Таненбаумом (англ. Andrew Stuart Tanenbaum) [20]. Изначально целью проекта было создание открытой учебной ОС для студентов компьютерных специальностей. Сегодня ОС Minix разрабатывается небольшим сообществом программистов и является

достаточно развитой UNIX-совместимой ОС, включающей средства разработки и отладки приложений, командные оболочки, стек протоколов TCP/IP, графическую среду X Window. ОС Minix бесплатно распространяется вместе с исходным кодом по лицензии BSD. Текущая актуальная версия ОС Minix 3.3.0 вышла в сентябре 2014 г.

2.5. Проект GNU

В 1983 г. Ричард Столлман (англ. Richard Matthew Stallman), сотрудник Массачусетского технологического института (MIT) (англ. Massachusetts Institute of Technology), основал проект по разработке свободного и независимого от существующего коммерческого исходного кода ПО. Проект предполагал создание как минимум текстового редактора, средств разработки ПО (включая компиляторы языков Си, Си++ и отладчик) и других привычных для UNIX-систем программ. Как максимум, предполагалось создание полноценной UNIX-совместимой ОС на собственном ядре. Проект получил название *GNU* как рекурсивный акроним от англ. *GNU is Not UNIX*. В 1985 г. Столлман, уже не работавший в MIT, основал Фонд свободного программного обеспечения (англ. Free Software Foundation, сокращенно FSF), который также стал содействовать проекту GNU.

Первой программой проекта GNU принято считать текстовый редактор Emacs. На начальном этапе проекта также были разработаны: компилятор языков Си и Си++ `gcc` (GNU C Compiler), ассемблер, отладчик `gdb` (GNU Debugger), другие необходимые для разработки программы, анализаторы текста, командная оболочка `bash`, архиваторы `gzip` и `tar`, совместимый с Postscript интерпретатор Ghostscript, а также программы для системы X Window. Сегодня ПО проекта GNU можно встретить практически в любой ОС, а компилятор `gcc` стал одним из стандартов компьютерной индустрии и даже поставляется с коммерческими версиями различных ОС.

Несмотря на большой набор программного обеспечения, у проекта не было самого главного — работоспособного и проверенного ядра ОС. Официальным ядром операционной системы GNU считается GNU Hurd, разработки которого ведутся с 1990 г. Тем не менее, проект по созданию ядра Hurd все еще не завершен, хотя Hurd на основе микроядра Mach уже может нормально функционировать и выполнять многие приложения.

ПО проекта распространяется по двум специальным лицензиям[1]. *Лицензии GNU GPL* (англ. General Public License — универсальная общественная лицензия) предоставляет пользователю права копировать, модифицировать и распространять (в том числе на коммерческой основе) программу, при условии, что все производные программы также будут распространяться под этой лицензией и с исходным кодом. Таким образом, например, запрещается создавать на основе свободной программы под лицензией GPL другой программный продукт, не предоставляя его исходный код пользователям.

Лицензия GNU LGPL (англ. Lesser General Public License — универсальная общественная лицензия ограниченного применения) позволяет связывать с данной библиотекой или программой авторскую программу под любой лицензией, несовместимой с GNU GPL, при условии, что такая программа не является производной от ПО, распространяемого под лицензией (L)GPL, кроме как путем связывания. Таким образом, лицензия LGPL, в отличие от GPL, позволяет связывание библиотеки с любой программой, необязательно свободной.

2.6. ОС Linux

В августе 1991 г. Линус Торвальдс (швед. Linus Benedict Torvalds), студент университета г. Хельсинки, объявил в новостной группе comp.os.minix о своей разработке новой ОС, исходный код которой вскоре был выложен в сети. Торвальдс разрабатывал свободную UNIX-совместимую ОС, независимую от существующих реализаций, в том числе и от учебной ОС Minix. Идея свободной ОС была подхвачена сообществом программистов, и система, названная *Linux*, стала стремительно набирать популярность. Вскоре Linux стала основной ОС для большого количества ПО, уже созданного к тому времени в рамках проекта GNU.

Сегодня ОС Linux является лидирующей в сфере распределенных вычислительных систем и Интернет-серверов, а также стремительно захватывает нишу офисных, домашних и мобильных компьютеров. По данным сайта TOP500 [2], собирающего статистику по программному и аппаратному обеспечению самых известных суперкомпьютерных систем в мире, на ноябрь 2014 г.

[1] http://www.gnu.org/licenses/licenses.html
[2] www.top500.org

лидером среди устанавливаемых ОС является Linux, составляя 97%. По данным сайта W3Techs[1] на февраль 2015 г. среди ОС общедоступных Интернет-серверов Linux также занимает первое место (доля 35,9%). По данным сайта StatCounter[2], определяющего статистику по ОС на основе данных веб-браузеров, доля систем, основанных на ядре ОС Linux, непрерывно растет и составила на февраль 2015 г. почти 25%.

ОС Linux, которую более корректно называть GNU/Linux, представляет собой, вообще говоря, только ядро, модули и самый минимальный набор системных утилит (например, утилиту конфигурирования ядра перед компиляцией последнего). Даже загрузчик ядра ОС в Linux не является неотъемлемой частью ОС. Однако общепринято под термином «Linux» понимать также полностью готовый к работе многофункциональный *дистрибутив* — сборку ядра Linux и ПО под конкретную аппаратную платформу, включающую, как правило, также программную и пользовательскую документацию.

Отличительной особенностью ОС Linux является фактически децентрализованная разработка ядра системы и сопутствующего системного и прикладного ПО. В настоящее время выпускается большое количество разнообразных дистрибутивов ОС Linux различными компаниями или группами программистов. По данным сайта http://www.operating-system.org насчитывается более 600 различных дистрибутивов ОС Linux. Бóльшая часть входящего в них программного обеспечения поставляется по лицензии GPL, т. е. с исходным кодом. Как правило, дистрибутивы уже содержат все необходимые для работы системные и прикладные программы, включая офисные, графические и мультимедийные приложения, средства сетевой коммуникации, веб-браузеры, почтовые программы, большой набор серверов, средства разработки и т. д.

С 2007 г. разработку ядра Linux координирует некоммерческий консорциум The Linux Foundation, в который входят более 180 крупнейших мировых IT-компаний[3], включая Fujitsu, HP, IBM, Intel, NEC, Oracle, Qualcomm, Samsung, AMD, Hitachi и др. Основной целью консорциума является развитие, продвижение и стандартизация ОС Linux.

[1] http://w3techs.com/technologies/overview/operating_system/
[2] http://gs.statcounter.com/
[3] http://www.linuxfoundation.org/about/members

ОС Linux является UNIX- и POSIX-совместимой ОС, поэтому прекрасно подходит для выполнения любых задач, для которых нужна надежная многопользовательская многозадачная сетевая ОС с самыми высокими требованиями по защищенности. Отсутствие необходимости платы за лицензию делает эту революционную ОС еще более привлекательной для всех категорий пользователей — от простых пользователей офисных ПК до системных администраторов и профессиональных программистов.

Полностью открытая архитектура превращает ОС Linux в идеальный инструмент для изучения программирования, ОС и современных сетевых технологий благодаря тому, что у пользователя всегда есть возможность получить справку по любому из компонентов ОС (команде, функции библиотеки или конфигурационному файлу). И, наконец, исчерпывающую информацию всегда можно получить из исходных кодов, распространяемых вместе с системой.

2.7. ОС семейств CP/M, DOS и Windows

В 1974 г. компания Digital Research выпустила простейшую однозадачную ОС, рассчитанную на использование в инструментальных 8-разрядных системах на базе микропроцессоров Intel 8080 и 8085. ОС поддерживала работу с накопителями на магнитной ленте, а также с накопителями на гибких магнитных дисках (НГМД). Система быстро приобрела популярность и стала фактически стандартом для микрокомпьютеров. Вскоре система была перенесена на 16- и 32-разрядные платформы, стала устанавливаться на первые персональные компьютеры (ПК), появившиеся в конце 1970-х гг. В начале 1980-х гг. была реализована многозадачная и сетевая версии CP/M [1]. CP/M представляет собой типичную дисковую ОС, работающую на процессорах с открытой моделью памяти (см. гл. 3) и имеющую следующие отличительные особенности:

- язык командного интерпретатора, схожий с DCL, но имеющий все команды с единственным вариантом написания имени;
- трехбуквенный системный идентификатор устройств ввода-вывода (например, `LPT` — строчный принтер, `CON` — консоль);
- однобуквенный метод идентификации всех дисков (носителей информа-

ции) (например, `A:` — первый дисковод в системе, `B:` — второй, `C:` — первый жесткий диск и т.д.);

- двоичные исполняемые файлы программ в формате `.COM`.

ОС CP/M имеет модульную структуру, в которой можно выделить три основные подсистемы:

1) базовую систему ввода-вывода, включающую драйверы физических устройств,
2) базовую дисковую операционную систему,
3) командный процессор (интерпретатор команд).

В 1981 г. компания IBM анонсировала ПК IBM PC на основе 16-разрядного микропроцессора Intel 8086, способного адресовать до 1 Мбайта оперативной памяти, но не поддерживающего виртуальную память и многозадачный режим работы. В силу ряда обстоятельств, ОС CP/M не стала устанавливаться на эти компьютеры. Вместо этого, контракт на разработку ОС для IBM PC получил Билл Гейтс (англ. Bill Gates), основавший к тому времени свою фирму Microsoft.

В итоге Microsoft предложила IBM ОС MS-DOS, основанную на клоне CP/M — ОС QDOS, которую Microsoft купила у компании Seattle Computer Products и адаптировала под аппаратуру IBM PC. Впоследствии Гейтс разработал для MS-DOS собственную файловую систему — FAT (англ. File Allocation Table — таблица размещения файлов), особенностью которой стала возможность создания вложенных каталогов. Также был добавлен новый формат двоичных исполняемых файлов с относительной загрузкой `.EXE` (от англ. executable — исполняемый) и была реализована динамическая загрузка драйверов устройств с указанием файлов драйверов в текстовом файле C:\CONFIG.SYS. Позже в системе появились многие черты ОС UNIX:

- API файловой системы, напоминающий API ОС UNIX;
- переменные среды;
- возможность переназначения ввода-вывода и конвейеры (см. п. 8.3, 8.4).

В конце 1980-х гг. компания Digital Research на основе CP/M разработала свою версию DOS — DR-DOS, программно совместимую с MS-DOS. Своя версия ОС под названием PC-DOS на основе исходного кода MS-DOS была

разработана и самой компанией IBM, которая поставлялась вместе с компьютерами IBM. Последняя, седьмая версия PC-DOS вышла в 1998 г., но система поставлялась со многими компьютерами и ноутбуками еще несколько лет.

ОС MS-DOS вместе со своими аналогами просуществовала более десятилетия, оставаясь основной ОС для ПК IBM PC с архитектурой x86. В начале 1990-х гг. она уже не могла удовлетворить возрастающие потребности пользователей. С появлением микропроцессора Intel 80286, а затем 80386 и разработкой на его базе следующего поколения 32-разрядных ПК IBM PC/AT, потребовалась новая ОС, в полной мере раскрывающая возможности новой архитектуры.

В конце 1980-х и начале 1990-х гг. стали появляться так называемые *расширители DOS* — специальные программы, позволяющие воспользоваться всей физически установленной в ПК оперативной памятью, объем которой уже исчислялся мегабайтами. Фактически, расширитель DOS можно считать отдельной ОС, предоставляющей свой специальный API. Загрузившись, расширитель переводил процессор в защищенный режим (см. гл. 3) и создавал для DOS задачу в режиме *виртуального процессора 8086*, так что DOS могла продолжать работу, предоставляя приложениям свою базовую подсистему ввода-вывода [1]. В 1991 г. был принят стандарт DPMI (англ. DOS Protected Mode Interface — интерфейс защищенного режима DOS), который определял требования к расширителю DOS и обеспечивал сосуществование нескольких расширителей в оперативной памяти.

В 1985 г. компания Quarterdeck разработала расширитель DESQview, создающий многооконную и многозадачную среду для DOS. Расширитель реализовывал режим невытесняющей многозадачности и переключение между задачами с применением виртуальных текстовых консолей.

С середины 1980-х гг. компания Microsoft стала разрабатывать еще один вариант надстройки над DOS — графическую среду Windows с многооконным пользовательским интерфейсом. Первые версии (1.0–3.1) были 16-разрядными, позволяли загружать в память сразу несколько приложений и переключаться между ними в режиме кооперативной многозадачности. Система Windows загружалась пользователем (после загрузки самой DOS) через исполняемый файл `WIN.EXE` и предоставляла дополнительный API гра-

фического интерфейса, при этом DOS оставалась в оперативной памяти и использовалась как подсистема ввода-вывода (включая файловую подсистему). Также особенностью Windows стала сборка программ в момент загрузки с использованием DLL-библиотек. Основные идеи архитектуры и пользовательского интерфейса Windows во многом позаимствовала от ОС MacOS персональных компьютеров фирмы Apple.

Первые версии Windows работали на 16-битном микропроцессоре Intel 80286, обеспечивающем несравненно меньшие возможности, чем 32-битный микропроцессор 80386. Последний обеспечил значительный прорыв в эксплуатационных характеристиках Windows, сделав возможным режим виртуального процессора 8086. Кроме того, в версии Windows 3.11 появилась своя 32-битная дисковая подсистема, функционирующая в защищенном режиме [1]. Однако, несмотря на все улучшения, надежность и функциональные возможности системы Windows все еще оставляли желать лучшего.

В полной мере реализовать возможности новых процессоров и защищенный многозадачный режим могла лишь принципиально новая ОС. С конца 1980-х гг. IBM совместно с Microsoft разрабатывали систему OS/2 для новой серии ПК IBM Personal System/2 (PS/2). Первые версии ОС OS/2 для микропроцессора 80286 использовали сегментную виртуальную память, а также динамическую сборку в процессе загрузки с DLL-библиотеками, формат которых был взят от Windows. Кроме того, OS/2 обеспечивала вытесняющую многозадачность, многопоточность в пределах одной задачи и некоторые средства межпроцессного взаимодействия, такие как семафоры и очереди сообщений (см. гл. 7,8). Одной из отличительных особенностей OS/2 являлся мощный механизм обработки исключений, аналогичный используемым в MVS, OS/390, z/OS и VMS [1].

С началом реализации OS/2 для микропроцессора 80386 между Microsoft и IBM возникли разногласия по поводу способа обеспечения совместимости между существующими 16-битными приложениями, использующими сегментированную память, и новой 32-битной ОС с линейным адресным пространством. В результате каждая из компаний стала развивать свою версию ОС.

Первая 32-битная версия IBM OS/2 2.0 представляла собой сочетание 16- и 32-битных подсистем. Так, подсистема ввода-вывода была полностью

16-битной и обеспечивала полную совместимость со старыми драйверами устройств. Ядро же в полной мере использовало возможности архитектуры: страничную организацию виртуальной памяти с «подкачкой» (см. гл. 3) и режим виртуального процессора 8086, что было актуально для совместимости с DOS-приложениями. В версии IBM OS/2 Warp 4.5 (1999 г.) была реализована 32-битная подсистема ввода-вывода, а также журналируемая файловая система JFS. В новой версии ОС был реализован стек сетевых протоколов TCP/IP, совместимый с 4.4BSD и поддерживающий фильтрацию IP пакетов [1].

С 1994 г. компания IBM стала развивать OS/2 под иную аппаратную платформу на основе процессоров PowerPC, имевшую хорошие перспективы в середине 1990-х гг. Однако, успех компании Intel в разработке микропроцессоров Pentium с тактовой частотой 100 МГц и более привел к провалу проекта PowerPC для IBM. Сегодня развитием OS/2 занимается американская компания Serenity Systems International, продавая систему под маркой eComStation, основанную на IBM OS/2 Warp 4.5.

Компания Microsoft воплотила наработки по OS/2 в своей новой ОС Windows NT (от англ. New Technology — новая технология), первая версия 3.1 которой вышла в 1993 г. и имела графический многооконный интерфейс, аналогичный Windows 3.x. В Windows NT изначально планировался программный интерфейс OS/2 API и затем POSIX, поддержка Windows API была добавлена в последнюю очередь. В итоге система обеспечила совместимость с приложениями для OS/2 1.x в отдельной подсистеме без возможности обращения 16-битных приложений к 32-битным DLL-библиотекам и наоборот. Многое в Windows NT было позаимствовано и из ОС VMS, в частности, страничная организация виртуальной памяти нижнего уровня с «подкачкой», идентификация пользователя на уровне процессов [1].

В ОС Windows NT появилась также журналируемая файловая система NTFS, представляющая собой гибрид файловых систем HPFS (OS/2) и FCS2 (VAX/VMS) [1]. Как и OS/2, Windows NT поддерживала сетевую файловую систему на основе протокола NetBIOS (сетевой–сеансовый уровни), разработанного в 1983 г. фирмой Sytek Corporation по заказу IBM, а также протокол прикладного уровня SMB (от англ. Server Message Block — серверные блоки сообщений), разработанный компаниями IBM, Microsoft, Intel и 3Com. Про-

токол SMB является основным сетевым протоколом для всех ОС семейства Windows и обеспечивает обмен файлами и сообщениями, а также функции сетевой печати.

Одной из особенностей всех последующих версий ОС Windows стал принцип хранения всей конфигурации системы и приложений в едином *системном реестре*, представляющим собой иерархическую базу данных. Идея системного реестра, по всей видимости, была позаимствована из архитектуры ПК Apple Macintosh [1]. ОС имеет функции API для работы с реестром на уровне операций записи, чтения и добавления, однако отсутствуют средства резервного копирования и восстановления. Практика показала, что, в случае повреждения реестра Windows по какой-либо причине, исправить положение может только полная переустановка ОС.

В первой половине 1990-х гг. зависимость пользователей от ранее разработанных DOS-приложений все еще оставалась высокой. В отличие от OS/2, обладавшей одной из лучших реализаций виртуальной DOS-машины, Windows NT не имела полноценной поддержки DOS. Особую сложность создавали приложения, применяющие непосредственный (низкоуровневый) доступ к специфическому оборудованию. Вследствие этого, компания Microsoft вынуждена была продолжать развитие системы Windows 3.x как «многозадачной DOS с защищенным (32-битным) режимом» [1] по крайней мере до тех пор, пока пользователи окончательно не откажутся от старых DOS-приложений.

Первый этап «превращения» Windows 3.x в 32-битную ОС был реализован через так называемую библиотеку Win32s, представлявшую собой набор DLL файлов и модулей ядра, поставляемых как самой компанией Microsoft, так и другими производителями в составе их собственного ПО. Библиотека Win32s позволяла исполнять загрузочные модули формата PE (от англ. Portable Executable — переносимый исполняемый модуль), предоставляя подмножество Win32 API от Windows NT.

В 1995 г. на рынок домашних и офисных ПК вышла система под названием Windows 95, получившая новый графический интерфейс с постоянно присутствовавшей на экране панелью задач и знаменитой кнопкой «Пуск». Система поддерживала технологию «Plug and Play» («подключил и играй»)

— быстрое определение и конфигурирование нового оборудования. Реально же процесс установки драйверов для нового оборудования, как правило, требовал несколько перезагрузок системы. Новая система также требовала установленную ОС MS-DOS, роль которой фактически сводилась к загрузке ядра Windows 95. ОС MS-DOS могла по-прежнему использоваться для работы старых драйверов устройств в целях совместимости, но это снижало стабильность работы системы.

Другой особенностью Windows 95 стала поддержка длинных имен файлов в файловой системе FAT32 с применением Unicode-символов, при этом DOS-приложения «видели» такие файлы с очень странными именами, оканчивающимися на символ «~» и цифру. Стоит также отметить, что в Windows 95 на уровне файловой системы отсутствовали какие-либо механизмы контроля доступа: файлы, включая системные и пользовательские, не были защищены от удаления или модификации, хотя формально система считалась многопользовательской.

Впоследствии до 2000 г. вышли версии Windows 98 и Windows ME (от англ. Millennium Edition), не имевшие никаких принципиальных архитектурных отличий от Windows 95. Указанные системы являлись переходными, имели большое количество архитектурных недостатков и низкую надежность, однако за счет агрессивной маркетинговой политики компании Microsoft они продержались на рынке достаточно долго, подготовив место для последующих ОС на базе Windows NT.

В 1996 г. вышла новая версия ОС Windows NT— 4.0[1]. Графический интерфейс был взят от Windows 95, была улучшена поддержка DOS-приложений. Система стала довольно популярной и просуществовала до 2000 г., когда ей на смену вышла версия 5.0, названная Windows 2000.

Через год компания Microsoft выпустила клиентскую ОС Windows NT 5.1 под названием Windows XP. Невысокие аппаратные требования, неплохая стабильность и скорость работы, а также достаточно высокая функциональность сделали Windows XP, пожалуй, самой популярной из всех ОС, выпущенных компанией Microsoft. Об этом также свидетельствует рекордный

[1]Выпускались две версии ОС — «серверная» и «клиентская» (для рабочих станций и ПК).

срок её эксплуатации — более 10 лет[1]. В 2003 г. вышла последняя из NT 5.x, серверная версия — Windows Server 2003.

Дальнейшие версии Windows NT не имели принципиально новых системных решений и каких-либо существенных улучшений потребительских качеств, однако стали отличаться явно завышенными аппаратными требованиями и невысоким быстродействием.

[1]Поддержка Windows XP компанией Microsoft прекращена 8 апреля 2014 г.

3. Управление оперативной памятью

3.1. Модели организация оперативной памяти

Оперативная память является важнейшим ресурсом любой современной ОС. Несмотря на то, что простейшую вычислительную систему можно реализовать вообще без оперативной памяти, ограничиваясь только внутренними регистрами микропроцессора, такая технология всерьез нигде не используется. Оперативная память необходима для решения, по крайней мере, следующих задач:

- хранение исполняемого кода программ (процессов),
- хранение обрабатываемых данных,
- организация стека.

Как будет показано далее, для многозадачной ОС оперативная память также необходима для реализации механизма *виртуальной памяти* и многозадачного режима.

Рассмотрим возможные модели организации оперативной памяти с точки зрения ОС (см. рис. 2). При этом стоит обратить внимание на то, что приведенная классификация касается именно *логической организации памяти*, которая не связана с физической реализацией оперативного запоминающего устройства (ОЗУ) в компьютере.

Рис. 2. Модели организации оперативной памяти.

Для простейших ОС типа DOS, рассмотренных в гл. 2, типична *открытая организация памяти*, не обеспечивающая никакой защиты данных. Первый тип такой памяти с чисто *линейной адресацией* свойственен большинству однозадачных восьмиразрядных систем, где в коде программы все адреса указываются явно. В этом случае логический адрес всегда в точности соответствует *физическому адресу*. Разновидностью открытой памяти является модель с применением *сегментной адресации*, когда память условно разбивается на сегменты, а логическая адресация происходит посредством уже двух компонент — *базового адреса сегмента* и *смещения* относительно него. В этом случае возникает возможность написания *позиционно-независимого кода* программы, но только в пределах одного сегмента, размеры которого всегда ограничены разрядностью процессора. Такая организация применялась, например, в 16-разрядных микропроцессорах i8086 и i8088 (ОС DOS) с целью увеличения адресного пространства выше предела в 64 кбайта.

Возможности вычислительной системы радикально возрастают с применением механизма *виртуальной памяти* (ВП). В первую очередь, при этом обеспечивается защита данных между ядром ОС и процессами. Т.е. наличие механизма ВП является необходимым условием для функционирования современной многозадачной ОС. В общем случае, в системах с ВП программы никогда не адресуются в физическое адресное пространство непосредственно. Вместо этого создается тот или иной *механизм трансляции* логического адреса в физический адрес. С точки зрения ОС, для удобства управления памятью (а это основная функция ОС) естественным способом организации такого механизма является логическое разбиение физического адресного пространства не некоторые блоки, имеющие размер порядка нескольких килобайт или более.

Трансляция логических адресов в физические происходит аппаратно, посредством специального *устройства управления памятью* (УУП) (англ. MMU — memory management unit), которое может входить в состав микропроцессора. При этом всегда происходит контроль доступа к таким блокам, а также контроль допустимых адресов в пределах установленных границ каждого блока.

Логический адрес в системах с ВП состоит из так называемого *селектора*, которым однозначно задается конкретный блок, и *смещения*, которое определяет адрес в пределах заданного блока. Принципиальное отличие открытой памяти с сегментной организацией от механизма ВП состоит в том, что в первом случае сам сегмент однозначно задается физическим адресом[1]. В случае же ВП блок не адресуется физическим адресом непосредственно, а задается селектором, который, в свою очередь, указывает на так называемый *дескриптор* блока (от англ. descriptor — дословно переводится как «описатель»). Вся информация о блоке, включая его линейный адрес, права доступа и другие атрибуты, хранится в дескрипторе. Дескрипторы в системе образуют одну или несколько таблиц, которые хранятся в оперативной памяти и обслуживаются ОС. Такие таблицы называются *таблицами дескрипторов* или *таблицами трансляции*.

Если размер всех блоков фиксирован и равен некоторой константе, реализуется механизм ВП со *страничной адресацией*, а блоки памяти именуются *страницами*. Размер страницы в зависимости от платформы варьируется от нескольких килобайт до нескольких мегабайт. В случае страничной адресации адресное пространство однозначно разбивается на фиксированные страницы, начальные физические адреса которых уже заранее определены реализацией конкретного микропроцессора или УУП.

Если размеры блоков могут различаться и задаваться отдельно для каждого блока, реализуется механизм ВП с *сегментной адресацией*, а блоки памяти именуются *сегментами*. Длина каждого сегмента в этом случае является одним из его атрибутов и хранится в дескрипторе сегмента. В случае сегментной адресации возможно появление неиспользуемых частей оперативной памяти, так называемых «дыр». Как правило, один сегмент соответствует коду или данным одного процесса [1].

Возможен третий вариант реализации ВП — *смешанная адресация*, когда применяются оба рассмотренных механизма, при этом сегментная адресация выступает более высоким уровнем, а страничная адресация — более низким, т.е. сегменты строятся из страниц.

[1] В микропроцессорах i8086 и i8088 такой 16-разрядный адрес аппаратно сдвигался влево на 4 разряда и становился в итоге 20-разрядным, что позволяло адресовать до 2^{20} = 1 Мбайт памяти. Размер сегмента оставался при этом ограничен размером 64 кбайта, поскольку смещение оставалось 16-разрядным.

3.2. Страничная организация виртуальной памяти

Страничная адресация представляет собой низкоуровневый механизм отображения виртуального адресного пространства в физическое, при котором физическая память разделяется на *страницы* фиксированного объема порядка нескольких килобайт. Отметим некоторые особенности страничной организации памяти.

1. При страничной адресации исчезает проблема фрагментации оперативной памяти, т.е. проблема неиспользуемых «дыр», поскольку в силу постоянства размера страницы любая неиспользуемая страница физической памяти может быть в любой момент «отдана» задаче, запрашивающей память. Такое сложное распределение адресного пространства, т.е. трансляция непрерывных для задачи логических адресов в различные области физической памяти, практически не сказывается на быстродействии.

2. При страничной организации виртуальной памяти возникает возможность «сбрасывать» на внешний накопитель те страницы физической памяти, которые в данный момент не используются, высвобождая дефицитный ресурс — физическую память. Файл, содержащий данные выгруженных страниц, часто называют *своп-файлом* (от англ. swap — обмен, перекачка).

3. Дескриптор страниц теоретически должен быть короче, поскольку исчезает необходимость хранения длины. Это означает возможность существенной экономии памяти на таблицах трансляции.

Как правило, диспетчер памяти имеет также кэш дескрипторов — быструю память с ассоциативным доступом, в которой хранятся дескрипторы часто используемых страниц. Записи этого кэша называются TLB (от англ. translation lookaside buffers — справочные буферы для трансляции). В 64-разрядных процессорах в силу большого объема адресного пространства объем таблиц трансляции становится слишком большим, поэтому часто УУП таких процессоров оперируют только кэш-таблицами TLB, в которых хранятся дескрипторы ранее использованных страниц [1]. Если задача обращается к странице, для которой нет записи в TLB, процессор генерирует исключение, в результате которого вызывается процедура-обработчик из ядра ОС. Эта процедура каким-либо образом должна найти дескриптор требуемой страницы согласно алгоритму работы диспетчера памяти данной ОС [1].

Рассмотрим механизм страничной адресации (см. рис. 3). УУП всегда содержит специальный регистр — указатель на таблицу дескрипторов (таблицу трансляции), которая размещается в отдельной области ОЗУ. Её элементами являются *дескрипторы страниц*. Дескриптор страницы содержит физический адрес страницы, права доступа, а также некоторые флаги, в числе которых имеется *бит присутствия* страницы в ОЗУ, используемый в алгоритме свопа. Алгоритм работы УУП при запросе логического адреса вида *страница:смещение* будет следующим [1].

1. Проверить, существует ли запрашиваемая страница вообще. Если страница не существует (номер страницы больше количества страниц), процессор генерирует исключение «ошибка сегментации» (segmentation violation).

2. Если номер страницы нормальный, попытаться найти дескриптор страницы в кэше TLB.

3. Если его нет в кэше, загрузить дескриптор из таблицы в памяти или сгенерировать запрос на поиск страницы ядру ОС.

4. Проверить, имеет ли процесс соответствующее право доступа к странице. Если не имеет, то процессор генерирует исключение «ошибка сегментации».

5. Проверить, находится ли страница в оперативной памяти. Если ее там нет, возникает особая ситуация отсутствия страницы или страничный

Рис. 3. Принцип организации страничной виртуальной памяти [1].

отказ (page fault). Как правило, реакция на нее состоит в том, что вызывается специальная программа-обработчик из ядра ОС (trap — ловушка), которая загружает требуемую страницу с диска. Тем самым реализуется механизм свопа.

6. Если страница есть в памяти, взять из ее дескриптора физический адрес.
7. Произвести доступ к памяти.

Рассмотрим принцип организации виртуальной памяти на примере 32-разрядной архитектуры Intel (IA-32) на базе микропроцессора Intel 80386. Механизм виртуальной памяти и поддержки многозадачности для данного семейства процессоров реализуется в так называемом *защищенном режиме*, когда становится возможным выполнение нескольких процессов с полной изоляцией и защитой друг от друга их адресного пространства [3, 20, 21].

Принцип организации страничной виртуальной памяти микропроцессора Intel 80386 иллюстрирует рис. 4. В данной архитектуре трансляция виртуального адреса двухуровневая, а размер страницы составляет 4 кбайта. Линейный 32-разрядный адрес, которым оперируют программы, включает три компоненты:

- 10-разрядный номер таблицы страниц в *каталоге таблиц* (биты с 22-го по 31-й),
- 10-разрядный номер страницы в выбранной *таблице страниц* (биты с 12-го по 21-й),
- 12-разрядное смещения ячейки памяти в выбранной *странице* (биты с 0-го по 11-й).

Каталог страниц является первым уровнем трансляции и представляет собой таблицу указателей на таблицу страниц. Каталог может включать максимум $2^{10} = 1024$ 32-битных указателя и сам занимает ровно одну страницу в оперативной памяти размером 4 кбайта. Формат указателя на таблицу страниц приведен в табл. 3. Старшие 20 бит указателя используются собственно для определения адреса таблицы страниц. Бит 1 (R/W) и бит 2 (U/S) используются при контроле доступа к странице и будут рассмотрены в п. 3.4; описание остальных полей можно найти в [21].

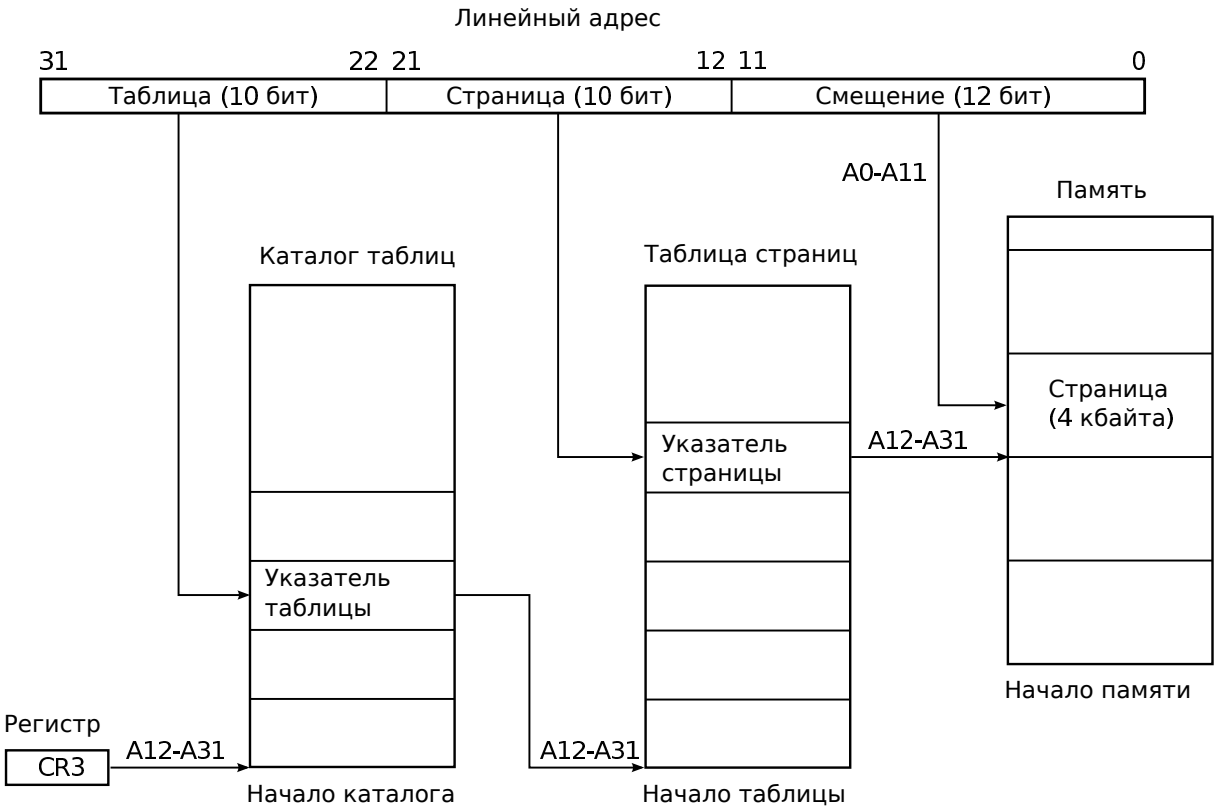

Рис. 4. Организации страничной памяти микропроцессора Intel 80386 [21].

Таблица страниц является вторым уровнем трансляции и представляет собой таблицу указателей на страницы. Таблица может включать максимум $2^{10} = 1024$ 32-битных указателя и также занимает ровно одну страницу в оперативной памяти. Формат указателя на страницу совпадает с форматом указателя на таблицу (см. рис. 3), причем в данном случае старшие 20 бит непосредственно определяют адрес страницы в физическом адресном пространстве. К этому адресу добавляется 12-разрядное смещение, и таким образом формируется итоговый 32-разрядный физический адрес ячейки памяти.

Таблица 3. Формат 32-битного указателя на таблицу страниц и на страницу для микропроцессора Intel 80386.

12–31	9–11	8	7	6	5	4	3	2	1	0
Базовый адрес (20 бит)	Резерв	0	0	D	F	0	0	U/S	R/W	P

Рассмотрим механизм страничной адресации для 64-разрядной архитектуры AMD64 (другое название x86-64) [22–24]. Линейный адрес в архитектуре AMD64 (в так называемом режиме Long) формально является 64-разрядным, но реально используются только младшие 48 бит, поскольку на данный момент отсутствует необходимость в адресации объема памяти в 2^{64} байт.

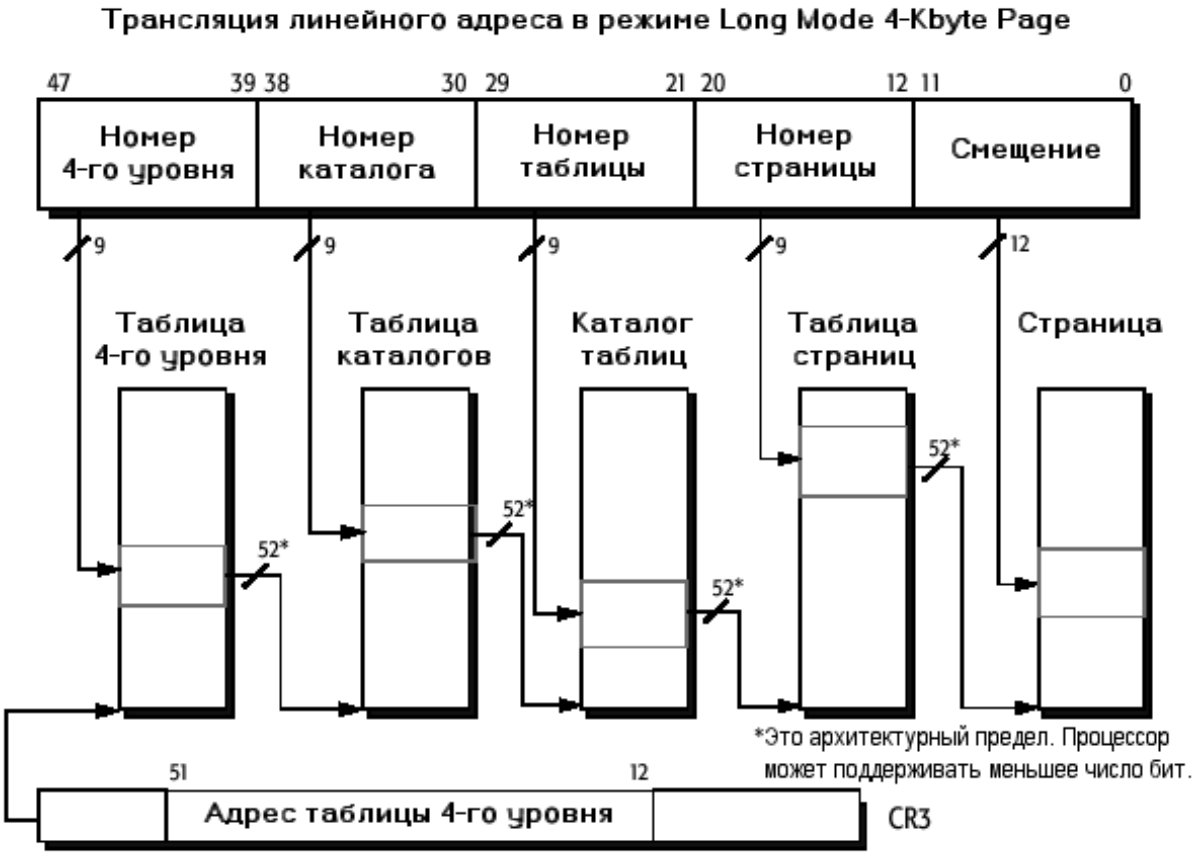

Рис. 5. Организации страничной памяти для архитектуры AMD64 в режиме Long, размер страницы 4 кбайта [22].

Размер страницы в данной архитектуре может составлять 4 кбайта, 2 Мбайта, 4 Мбайта или 1 Гбайт. УУП AMD64 поддерживает различные режимы трансляции, в том числе режимы совместимости с механизмом PAE (от англ. Physical Address Extensions — расширение физического адреса) процессоров Intel Pentium. Во всех режимах размер указателей трансляции составляет 64 бита, но их форматы различаются. При размере страницы 4 кбайта в режиме Long применяется четырехуровневая схема трансляции линейного адреса, которую иллюстрирует рис. 5. Линейный адрес состоит из четырех 9-разрядных компонент, определяющих физический адрес страницы, и 12-разрядного смещения. В табл. 4 приведены общие для всех указателей поля; конкретный формат указателя каждого уровня трансляции приведен в [24]. Базовый адрес составляет 40 бит и определяет физический адрес страницы. К этому адресу добавляется 12-разрядное смещение, и таким образом формируется итоговый 52-разрядный физический адрес ячейки памяти. Биты R/W и U/S используются при контроле доступа к страницам и будут рассмотрены в п. 3.4; описание остальных полей можно найти в [24].

Таблица 4. Обобщенный формат 64-битного указателя трансляции для режима AMD64 Long с размером страницы 4 кбайта.

63	52–62	12–51	9–11	8	7	6	5	4	3	2	1	0
NX	Резерв	Базовый адрес	Резерв	x	x	x	A	PCD	PWT	U/S	R/W	P

Примечание: x — биты, назначение которых зависит от уровня трансляции.

3.3. Сегментная организация виртуальной памяти

При сегментной организации ВП процессы используют логическую адресацию в пределах определенного *сегмента памяти*, который уже аппаратно отображается в конкретную область физической памяти (см. рис. 6). Как правило, процессу отводятся стандартные сегменты: *сегмент кода команд* (сегмент программы), *сегмент стека* и несколько *сегментов данных*.

Сегментная адресация может применяться вместе со страничной как дополнительный уровень виртуализации памяти, обеспечивая «прозрачный» доступ к памяти для прикладных программ, в которых нужно учитывать только сегментную адресацию.

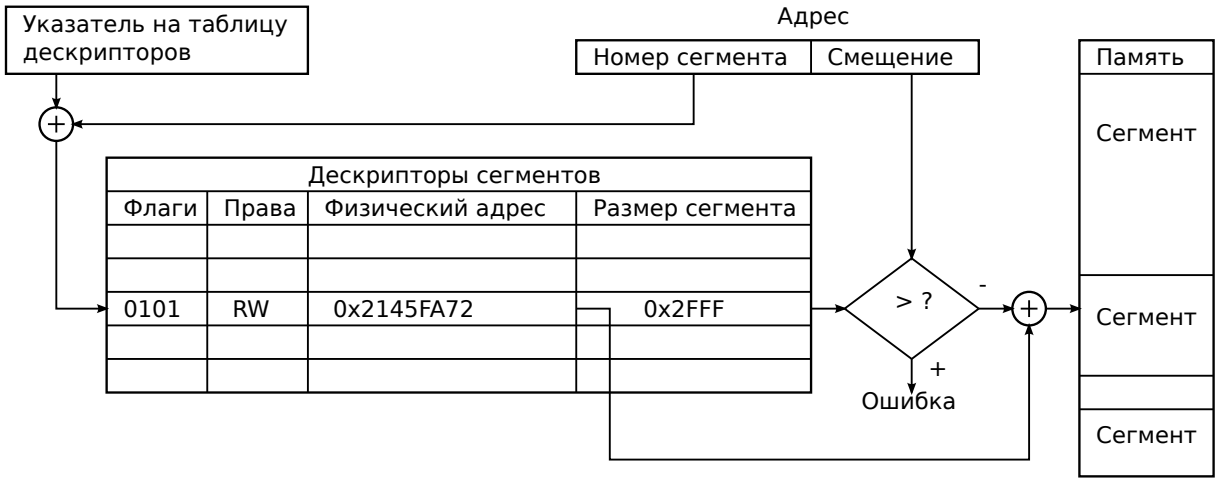

Рис. 6. Принцип организации сегментной виртуальной памяти [1].

Все механизмы адресации также предполагают организацию защиты данных и контроля доступа к программному коду процессов.

Рассмотрим защищенный режим работы микропроцессора Intel 80386 на основе механизма сегментной адресации [21]. В микропроцессоре для каждого сегмента формируется *дескриптор сегмента* — структура данных, описывающая атрибуты сегмента. В табл.5 и 6 приведен общий формат дескриптора сегмента и рассмотрены атрибуты, важнейшими из которых с точки зрения контроля доступа являются:

Таблица 5. Атрибуты 8-байтного дескриптора сегмента микропроцессора Intel 80386.

атрибут	байт 0	байт 1	байт 2	байт 3	байт 4	байт 5	байт 6	байт 7
базовый адрес			0–7	8–15	16–23			24–31
размер сегмента	0–7	8–15					16–19 (биты 0–3)	
бит дробности G							бит 7	
бит разрядности D							бит 6	
байт доступа						0–7		

Таблица 6. Формат байта доступа дескриптора сегмента микропроцессора Intel 80386 в зависимости от назначения сегмента.

назначение	бит 7	бит 6	бит 5	бит 4	бит 3	бит 2	бит 1	бит 0
системный	P	DPL		S=0	тип			
код (программа)	P	DPL		S=1	E=1	C	R	A
данные	P	DPL		S=1	E=0	ED	W	A

- 32-разрядный *базовый адрес* — адрес начала сегмента в виртуальной памяти,
- 20-разрядный *размер сегмента* — размер сегмента, указываемый в байтах при бите G=0 или в страницах объемом по 4 кбайта при G=1,
- 2-битный *уровень привилегий дескриптора сегмента* — DPL (от англ. Descriptor Privilege Level),
- отдельные биты байта доступа дескриптора: R — бит разрешения считывания сегмента кода (при R=0 код можно только выполнить), W — бит разрешения записи в сегмент данных (при W=0 данные можно только прочитать).

В системной части физической памяти организуются:

- *глобальная таблица дескрипторов* GDT (англ. Global Descriptor Table),
- несколько *локальных таблиц дескрипторов* LDT (англ. Local Descriptor Table) в зависимости от числа процессов,
- *таблица дескрипторов прерываний* IDT (англ. Interrupt Descriptor Table).

Обращение к дескриптору сегмента осуществляется с помощью 16-

разрядного *селектора сегмента*, значения которого для соответствующих сегментов хранятся в регистрах микропроцессора CS, SS, DS, ES, FS и GS [21]. Формат селектора приведен в табл. 7. Бит TI определяет, к какой таблице дескрипторов происходит обращение: GDT при TI=0, LDT при TI=1, а сам дескриптор из соответствующей таблицы выбирается по прямому 16-разрядному смещению, полученному обнулением младших 3-х бит селектора. Таким образом, с помощью 13-разрядного индекса можно обратиться к 8192 различным 8-байтным дескрипторам из таблиц GDT или LDT.

Таблица 7. Формат селектора сегмента микропроцессора Intel 80386.

биты 3–15	бит 2	биты 0–1
индекс	поле TI	поле RPL

3.4. Защита памяти и контроль доступа

Принципы защиты данных и контроля доступа на базе механизма виртуальной памяти, рассматриваемые далее, являются аппаратной основой обеспечения информационной безопасности в любой вычислительной системе.

Практически любой современный процессор обеспечивает *мандатный контроль доступа* к страницам или сегментам на основе механизма *привилегий* или так называемых *колец защиты*. Обычно в процессорах реализуют от двух до четырех уровней привилегий. Мандатный контроль доступа в процессорах семейства x86 (Intel 80386 и последующих, включая AMD64), основан на четырех уровнях привилегий от 0 до 3, причем наиболее привилегированным является уровень 0.

Рассмотрим принцип защиты в механизме страничной ВП процессоров x86. Указатели на таблицу страниц и на страницу определяют права доступа к соответствующей таблице или странице на основе битов R/W (1) (от англ. read/write) и U/S (2) (от англ. user/supervisor) (см. табл. 3,4). Если осуществляется запрос с уровнем привилегий 3, то при значении бита U/S=0 ему запрещается доступ к соответствующей таблице или странице. Если U/S=1, то при значении R/W=0 разрешается только чтение, а при R/W=1 — чтение и запись. При запросах с большими привилегиями (0–2) допускается запись и чтение при любых значениях битов U/S и R/W.

Обратимся к принципам построения механизма защиты на уровне сегментов для микропроцессора Intel 80386. Базовый адрес и размер сегмента определяют разрешенную для данного процесса (например, его сегмента данных) область памяти: при попытке процесса выйти из этого диапазона адресов генерируется *исключение*, что дает сигнал ОС завершить этот процесс, поскольку он нарушил разрешенные границы памяти и его дальнейшие действия непредсказуемы.

Мандатный контроль доступа к сегментам в микропроцессоре Intel 80386 основан на атрибуте сегмента DPL. Кроме того, микропроцессор использует также *уровень привилегий запроса* — RPL (от англ. Requested Privilege Level), соответствующий уровню привилегий инициатора запроса-обращения к сегменту. Значение RPL хранится как атрибут (младшие 2 бита) *селектора* — 16-разрядного указателя на дескриптор сегмента в таблице дескрипторов. Микропроцессор автоматически поддерживает также *текущий уровень привилегий* CPL (от англ. Current Privilege Level), равный содержимому поля DPL дескриптора сегмента кода команд. В соответствии со значениями DPL все сегменты в системе оказываются распределенными по кольцам защиты соответствующих уровней привилегий.

Микропроцессор осуществляет проверку на допустимость выполнения команд в соответствии с текущим уровнем привилегий CPL и уровнем привилегий сегментов DPL, данные из которых указаны в качестве операндов. В случае попытки обращения к более привилегированному сегменту генерируется исключение, которое дает команду ядру ОС завершить процесс, попытавшийся нарушить систему защиты. На основе рассмотренных уровней привилегий применяются следующие правила:

1) данные из сегмента с уровнем защиты DPL могут быть выбраны процессом, имеющим такой же или более высокий уровень привилегий, т. е. при условии $\max\{\text{RPL},\text{CPL}\} \leq \text{DPL}$ (численно);
2) сегмент кода (подпрограмма), имеющий уровень защиты DPL, может быть вызван процессом, имеющим такой же или более низкий уровень привилегий CPL, т. е. при условии $\text{CPL} \geq \text{DPL}$ (численно).

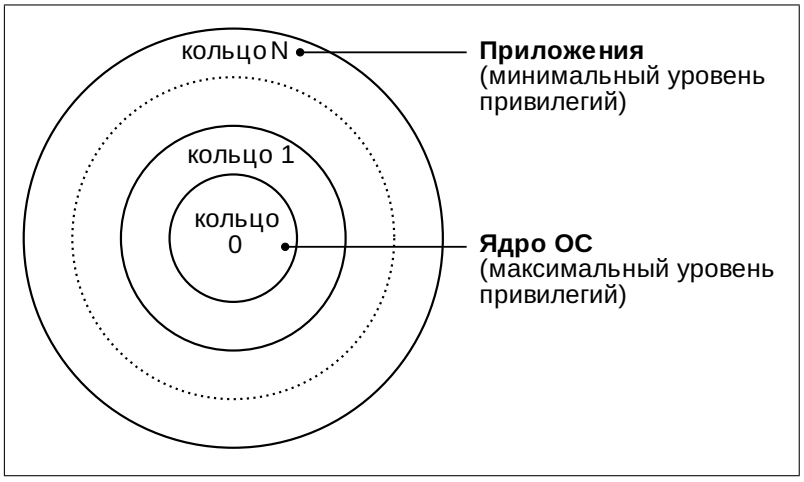

Рис. 7. Кольца защиты ОС.

Эффективное использование архитектуры колец защиты требует тесного взаимодействия между ОС и аппаратными средствами. Конкретная ОС, поддерживающая несколько аппаратных платформ, может иметь различную реализацию механизма колец защиты на каждой аппаратной платформе.

Структуру уровней привилегий можно изобразить в виде нескольких концентрических кругов — колец защиты (см. рис. 7). В этом случае ядро ОС выполняется с максимальным уровнем привилегий в кольце 0, а остальные процессы, включая прикладные задачи, «располагаются» в кольце с бóльшим номером. Системные процессы-драйверы устройств могут находиться между ними, т.е. в кольцах 1, 2 и т. д.

Большинство распространенных ОС, таких как UNIX и Windows, используют модель безопасности на основе двухуровневой схемы привилегий, даже если аппаратная платформа обеспечивает бóльшее число уровней. Для архитектуры x86 при этом используется уровень привилегий 0 для ядра ОС, включая модули и драйверы, и уровень 3 для процессов пользователя.

4. Файловая система

4.1. Организация хранения данных

Долговременное хранение данных в вычислительных системах, как правило, осуществляется на носителях информации различного физического принципа действия, обеспечивающих сохранность данных даже в отсутствии электроэнергии. Сохраняемая на носителях информация организуется на основе так называемых *файлов* (англ. file — сборник (подшивка) документов).

Файл представляет собой некоторую связанную именованную последовательность байт, отражающих информационное содержание некоторой сущности, например, текста (документа или программы), графической картинки, набора числовых данных, оцифрованного звукового или видео материала и т.д. Файл является ключевым понятием любой ОС.

За хранение данных в ОС отвечает *файловая система* (англ. file system). Под файловой системой (ФС) в зависимости от конкретного контекста может пониматься следующее:

1. *Файловая подсистема ОС*, т.е. часть системного программного обеспечения, отвечающая за хранение данных на носителях и обеспечивающая выполнение всех операций по созданию, изменению, чтению и удалению информации (см. п. 4.3).

2. Конкретный *формат хранения информации* (файлов) на носителе определенного вида, что также предполагает наличие программных средств управления конкретным типом ФС в самой ОС. Как было сказано выше, эта функция может быть реализована как в ядре ОС, так и в виде отдельного модуля ядра или драйвера данного типа ФС.

3. *Логическая ФС*, т.е. совокупность всех доступных носителей информации в данной вычислительной системе, обращение к которым осуществляется посредством *единого программного интерфейса* (API), а механизм адресация файлов в логической ФС унифицирован и не зависит от вида носителя или физического формата хранения информации.

Таким образом, файловая подсистема ОС реализует единый API для доступа ко всем подключенным носителям информации с применением унифицированного механизма адресации файлов. При этом за работу с каждым

конкретным физическим носителем информации отвечает модуль ядра или отдельный процесс, понимающий формат данного носителя (тип его ФС). Далее под ФС будет пониматься совокупность файловой подсистемы ОС и средств работы с физическими носителями информации.

ФС должна предоставлять как минимум две функции:

1) хранение данных в виде именованных файлов на внешних носителях информации (функция *идентификации* файлов);
2) хранение атрибутов файлов: размер, время создания, время последнего доступа, время последней модификации, тип файла и пр.

Большинство современных ФС предоставляют также следующие функции:

1) иерархическая организация файлов с применением *каталогов*;
2) разграничение доступа к объектам ФС между пользователями (хранение специальных дополнительных атрибутов);
3) шифрование отдельных файлов или всей ФС целиком;
4) минимизация фрагментации файлов;
5) обеспечение совместного доступа к файлам со стороны нескольких процессов.

Кроме того, к современным ФС предъявляется требование отказоустойчивости к сбоям электропитания, ошибкам аппаратных и программных средств.

4.2. Физический уровень файловой системы

Физически файловая система может быть реализована на различных внешних носителях информации, таких как жесткие магнитные диски (винчестеры), гибкие магнитные диски, внешняя электронная память flash, оптические диски CD и DVD. Здесь стоит упомянуть такие распространенные типы ФС как FAT, NTFS, UFS, EXT2/3/4, для оптических (лазерных) дисков — ISO, UDF. Кроме того, вполне возможно создание ФС в выделенной части оперативной памяти компьютера (так называемый RAM-disk, т. е. «диск» в ОЗУ), а также сетевой доступ к удаленным ресурсам другого компьютера посредством специальной *сетевой файловой системы*, типичными примерами которой являются системы NFS (от англ. Network File System) в UNIX и NetBIOS в MS Windows.

4.3. Операции в файловых системах

ОС должна предоставить в распоряжение пользователя набор операций для работы с объектами файловой системы, которые реализуются через системные вызовы [20,25]. Чаще всего при работе с файлом пользователь выполняет не одну, а несколько операций. Во-первых, нужно найти данные файла и его атрибуты по символьному имени, во-вторых, считать необходимые атрибуты файла в отведенную область оперативной памяти и проанализировать права пользователя на выполнение требуемой операции. Затем следует выполнить операцию, после чего освободить занимаемую данными файла область памяти.

Основные файловые операции можно разбить на две условные группы:

- *операции манипулирования файлами и каталогами* как объектами ФС;
- *операции доступа к данным файла*.

Рассмотрим вначале операции манипулирования файлами и каталогами:

1) *создание*, т.е. добавление файла или каталога в ФС и присвоение ему ряда атрибутов. При этом выделяется место на диске и вносится соответствующая запись в ФС;

2) *удаление* файла или каталога и освобождение занимаемого им дискового пространства;

3) *переименование*, т.е. присвоение существующему файлу или каталогу нового имени. Учитывая иерархическую структуру ФС, *операция переноса* файла или каталога в другой каталог может трактоваться как частный случай операции переименования, т.к. после нее изменяется путь к объекту, и, следовательно, его полное имя. Поэтому в ОС UNIX существует универсальная команда переименования и перемещения объектов ФС `mv` (от англ. move — передвигать);

4) *изменение атрибутов* файла или каталога. Часть атрибутов изменяется автоматически (например, время последнего доступа к объекту), а часть может изменить только пользователь (например, право файла на исполнение).

4.4. Операции доступа к данным файла

Файлы, предназначенные непосредственно для хранения информации, называются *регулярными*. Регулярный файл в ОС UNIX всегда является объектом *последовательного доступа*. Это означает, что операции записи и чтения данных с файлами выполняются в потоко-ориентированном режиме, т.е. побайтно. Как будет показано в гл. 5, в общем случае, в ОС UNIX файлы могут быть предназначены не только для хранения данных, но также могут играть роль некоторого интерфейса, через который осуществляется доступ к другим объектам или устройствам, например, к коммуникационным портам или сетевым сокетам. Если такие объекты имеют интерфейс последовательного доступа, то и соответствующий файл является объектом последовательного доступа. Объекты последовательного доступа в ОС UNIX принято называть также *символьными* (англ. character - символ, единица информации (байт)). Общим для всех символьных объектов является *потокоориентированный* доступ с *буферизацией* по чтению и по записи. Перечень операций доступа к данным символьных объектов включает:

1) *открытие файла*,

2) *закрытие файла*,

3) *запись данных*,

4) *чтение данных*,

5) *позиционирование*,

6) *синхронизация буфера*.

Операция *открытия файла* выполняется перед использованием файла с целью разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в оперативную память список адресов блоков данных файла для быстрого доступа к его содержимому. API ОС UNIX предлагает по крайней мере два различных метода работы с файлами: двоичный доступ через системные вызовы и форматированный ввод-вывод через библиотечные функции. Второй метод не исключает возможность двоичного доступа к данным. Для работы с одним заданным файлом в программе применяется, как правило, либо первый, либо второй метод, хотя и возможен вариант их сочетания.

В случае использования системных вызовов файл открывается универсальным вызовом `open()`, возвращающим *дескриптор файла* как целое число. В дальнейшем оно используется во всех остальных вызовах при работе с данным файлом. Файловый дескриптор уникален только в пределах данного процесса.

В случае использования функций форматированного ввода-вывода файл открывается функцией `fopen()`, которая возвращает указатель на специальную файловую структуру `FILE`. Она играет роль файлового дескриптора и также используется во всех остальных функциях при работе с данным файлом.

Существует несколько *режимов открытия файла*, в том числе: открытие для *чтения*, *записи*, *чтения и записи*, *добавления* (записи в конец существующего файла). Режим указывается в соответствующей функции открытия файла вместе с именем файла. Для успешного открытия файла режим открытия файла не должен противоречить *режиму доступа* к файлу, который был до этого установлен в ФС (см. гл. 6).

Операция *закрытия файла* выполняется после завершения работы с файлом с целью освобождения места во внутренних таблицах файловой системы и записи данных из буфера на физический носитель, если файл был открыт для записи. Для закрытия файла используется системный вызов `close()`, если файл был открыт через вызов `open()`, или функция `fclose()`, если файл был открыт функцией `fopen()`.

Операция *записи данных* в файл помещает новые данные в некоторую текущую позицию файла. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые таким образом теряются. После успешной записи данных текущая позиция в файле продвигается вперед на число записанных байт. Запись данных осуществляется системным вызовом `write()`, если файл был открыт вызовом `open()`, или функциями стандартной библиотеки типа `fwrite()` и `fprintf()`, если файл был открыт функцией `fopen()`.

При записи данных в файл ОС по умолчанию использует *буфер записи* — область памяти для промежуточного (временного) хранения информации.

Буфер предназначен для оптимизации операций вывода информации по затрачиваемому времени, а также для согласования асинхронной работы процесса, генерирующего данные, и устройства, принимающего данные (носителя ФС или коммуникационного объекта). Для корректного завершения всех операций записи буфер должен быть синхронизирован с файлом или устройством. Иными словами, вся накопленная в нем информация должна быть отправлена в объект назначения. Это всегда выполняется при закрытии файла, а также может быть выполнено явно соответствующими системными вызовами (см. далее).

Операция *чтение данных* осуществляет извлечение информации из файла с некоторой текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти. Каждый раз при чтении данных текущая позиция в файле продвигается вперед на число считанных байт. Чтение данных осуществляется системным вызовом `read()`, если файл был открыт через вызов `open()`, или функциями стандартной библиотеки типа `fread()`, `fscanf()`, `fgetc()` или `fgets()`, если файл был открыт функцией `fopen()`.

При чтении данных из файла ОС по умолчанию использует *буфер чтения*. Буфер чтения имеет, в принципе, такое же назначение, как и буфер записи, только работает при передаче данных в другом направлении. Соответственно, ОС обычно считывает данные из файла в большем объеме, чем это запрашивает системный вызов. В случае коммуникационных устройств буфер решает задачу временного хранения принимаемых данных до момента их востребования со стороны процесса.

Операция *позиционирование* дает возможность специфицировать место внутри файла, откуда будет производиться считывание данных или куда будут записываться новые данные, то есть явно задать текущую позицию. В API имеется вызов произвольного позиционирования `lseek()` для файла, открытого вызовом `open()`, а также функции `fseek()` и `rewind` (позиционирования в начало файла) для файла, открытого функцией `fopen()`. Стандартная библиотека предоставляет также функцию `ftell()`, которая возвращает текущую позицию (в байтах) для файлов, открытых функцией `fopen()`.

Операция *синхронизации буфера* обычно применяется для синхронизации буфера записи после операций записи в файл. Синхронизация (или другими словами опустошение, сброс) буфера гарантирует завершение фактической передачи данных от процесса на конечный носитель информации или коммуникационное устройство. Синхронизация выполняется всегда при закрытии файла. Необходимость явно вызывать функции синхронизации возникает, например, при реализации системы журналирования событий для процесса. В этом случае после записи каждого сообщения в журнал нужно вызывать синхронизацию, иначе при аварийном завершении процесса существует вероятность потери информации, временно хранящейся в буфере.

API ОС UNIX предоставляет системные вызовы синхронизации `fsync()` и `sync()` для файла, открытого вызовом `open()`, а также функцию `fflush()`, для файла, открытого функцией `fopen()`. Также имеется команда `sync`, которая синхронизирует все буферы всех процессов. Команда может быть вызвана от имени любого пользователя.

Часто бывает необходимо выяснить, какие процессы используют указанный файл. Для это существует специальная команда `fuser`, возвращающая список процессов, которые в данный момент работают с этим файлом. Аналогично работает команда `lsof`, но при этом она также позволяет выполнить обратную задачу — узнать, какие файлы открыты указанным процессом.

4.5. Файловые дескрипторы и потоки

Файл, который может быть открыт через библиотечную функцию `fopen()` с последующей его идентификацией через указатель на структуру `FILE`, в стандартной библиотеке часто именуется *потоком* (англ. stream). Соответствующий набор библиотечных функций с префиксом `f*()` реализует более высокоуровневый (форматированный) доступ к данным, используя, в свою очередь, системные вызовы типа `open()`, `read()`, `write()`, `close()`.

Стандартная библиотека предоставляет функцию `fileno()`, которая в качестве аргумента принимает указатель на структуру `FILE` и возвращает соответствующий целочисленный файловый дескриптор. Таким образом, применяя высокоуровневый метод работы с файлами (потоками), у программиста остается возможность обратиться к низкоуровневым системным вызовам через дескриптор для решения каких-либо специфических задач.

API ОС UNIX предоставляет системные вызовы для создания новых файловых дескрипторов к уже открытым в данном процессе файлам:

- `dup()` — создает новый файловый дескриптор (как минимальное свободное число) к указанному дескриптору в качестве дубликата, после чего с файлом можно продолжить работать через любой из дескрипторов; при этом для существующего и нового дескрипторов применяется один и тот же указатель текущей позиции чтения или записи, однако возможно независимое закрытие файла через каждый из дескрипторов (через вызов `close()`);
- `dup2()` — создает новый заданный файловый дескриптор, указанный в аргументах вместе с существующим дескриптором, после чего существующий дескриптор закрывается;
- `fcntl()` — может выполнять разные манипуляции с дескриптором, в том числе создавать дубликат аналогично `dup2()`, устанавливать режим закрытия дескриптора при порождении нового процесса (см. п. 7.3).

Эти вызовы часто используются при создании новых процессов, пример их применения будет разобран в п. 8.4.

4.6. Структура файловой системы ОС UNIX

Файловая система ОС UNIX, в отличие от распространенных офисных ОС (DOS, Windows или OS/2), имеет четко заданную структуру каталогов корневого уровня, а также одного или более последующих уровней в зависимости от назначения каталога. Основной принцип организации каталогов описывается в стандарте FHS (англ. Filesystem Hierarchy Standard — стандарт иерархии файловой системы), принятом для унификации местонахождения файлов и директорий с общим назначением в файловой системе ОС UNIX [26]. На данный момент большинство UNIX-подобных систем в той или иной степени следует этим правилам. Например, исполняемые файлы пользовательских программ всегда хранятся в каталогах `/bin` и `/usr/bin`, а конфигурационные файлы — в каталоге `/etc`.

Логически ФС в ОС UNIX представляет собой древовидную структуру, начинающуюся с *корневого уровня*, или *корневого каталога*, обозначаемого

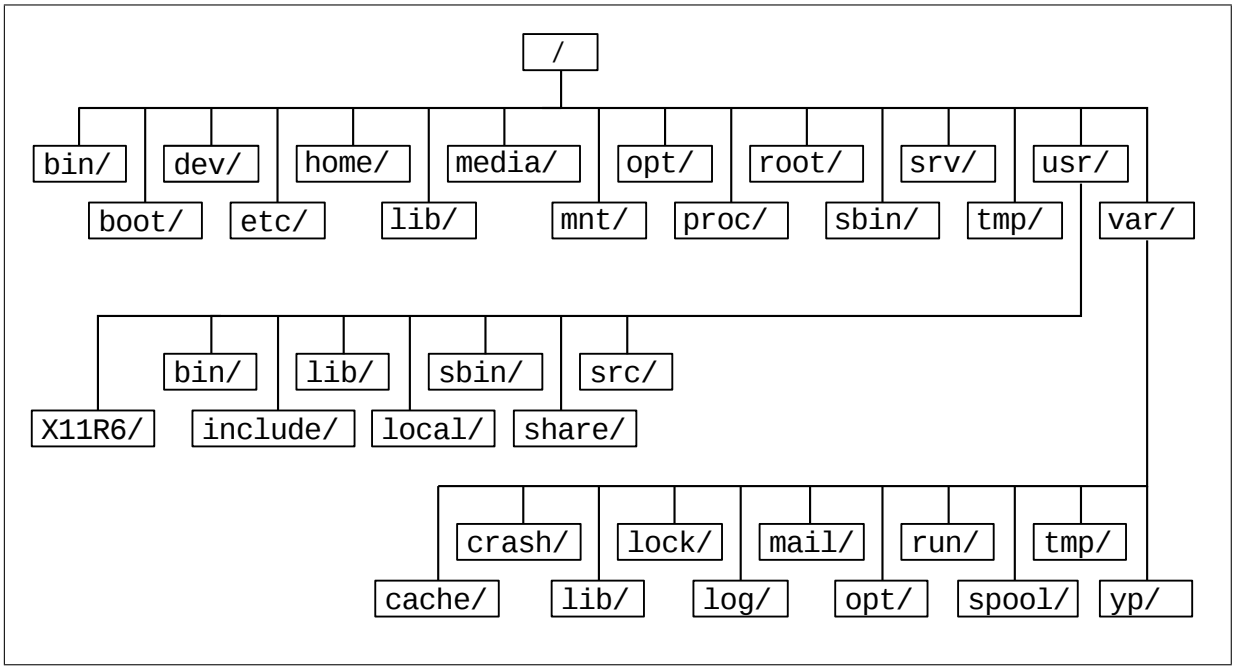

Рис. 8. Структура ФС ОС UNIX по стандарту FHS (до второго уровня включительно).

символом *прямого слеша* «/». На корневом уровне располагаются основные каталоги ФС, содержащие другие файлы и каталоги, т. е. продолжения древовидной структуры и т. д. Положение любого объекта ФС, именуемое также *путем к объекту*, обозначается через указание имен всех вышестоящих каталогов, начиная с корневого, и имени самого объекта, с разделением каждого уровня символом «/». Такое указание положения объекта называется *абсолютным* или *полным путем*. Примеры абсолютного пути: `/bin` — каталог корневого уровня `bin` или `/usr/local/lib` — каталог `lib`, находящийся в каталоге `local` каталога корневого уровня `/usr`. Структура ФС ОС UNIX по стандарту FHS вплоть до второго уровня изображена на рис. 8. Рассмотрим назначение основных каталогов стандарта.

`/bin/` — исполняемые файлы основных программ ОС, включая командные оболочки, необходимые всем пользователям;

`/boot/` — загрузочные файлы, в том числе файлы загрузчика ОС, образ ядра ОС и другие данные, необходимые для начальной загрузки;

`/dev/` — основные файлы устройств (например, `/dev/null`, `/dev/sda`);

`/etc/` — каталог общесистемных конфигурационных файлов (от лат. et cetera), содержит подкаталоги для конфигурирования подсистем;

`/home/` — содержит домашние каталоги пользователей, которые, в свою оче-

редь, содержат персональные настройки и данные пользователя (часто размещается на отдельном разделе);

`/lib/` — основные библиотеки, необходимые для работы ОС, в частности, программ из каталогов `/bin/` и `/sbin/`, а также модули ядра;

`/media/` — каталог содержит точки монтирования для сменных носителей, таких как CDROM или внешних USB-устройств хранения данных;

`/mnt/` — каталог содержит точки монтирования других файловых систем;

`/opt/` — дополнительное программное обеспечение, как правило, имеющее свою иерархию каталогов в стиле FHS;

`/proc/` — виртуальная файловая система, представляющая состояние ядра ОС, устройств и запущенных процессов в виде файлов;

`/root/` — домашний каталог суперпользователя root;

`/sbin/` — исполняемые файлы основных системных программы для администрирования и настройки ОС (например, `init`, `poweroff`);

`/srv/` — данные, необходимые для работы системных и сетевых серверов, (например, `/srv/www/` содержит файлы веб-сервера);

`/tmp/` — общесистемный каталог временных файлов, обычно очищается при перезагрузке ОС;

`/usr/` — вторичная иерархия каталогов для пользовательских программ; содержит большинство приложений и утилит, используемых в многопользовательском режиме (каталог может быть смонтирован по сети и быть общим для нескольких машин);

`/usr/bin/` — исполняемые файлы дополнительных программ для всех пользователей;

`/usr/include/` — стандартные заголовочные файлы для компилятора Си;

`/usr/lib/` — библиотеки для программ, находящихся в каталогах `/usr/bin/` и `/usr/sbin/`, а также библиотеки языка Си для статической сборки программ;

`/usr/sbin/` — исполняемые файлы дополнительных системных программ (например, демоны различных сетевых сервисов);

`/usr/share/` — платформенно-независимые общие данные для пользовательских приложений;

`/usr/src/` — исходные коды программ, в том числе ядра ОС;

`/usr/X11R6/` — отдельный каталог файлов графической подсистемы X Window версии 11, релиз 6 (см. п.9.4) со своей иерархией;

`/usr/local/` — третичная иерархия каталогов, специфичных для данного компьютера; обычно содержит подкаталоги `bin/`, `lib/`, `share/`;

`/var/` — каталог, объединяющий все изменяемые в системе файлы, такие как файлы журналов (log-файлы), временные почтовые файлы, файлы очереди печати принтера и т. д.;

`/var/lib/` — данные, изменяемые программами в процессе работы (например, базы данных, метаданные пакетного менеджера, база данных DNS-сервера и др.);

`/var/lock/` — лок-файлы, указывающие на занятость некоторого ресурса;

`/var/log/` — файлы системных журналов (лог-файлы) (от англ. log);

`/var/mail/` — каталог содержит почтовые ящики пользователей;

`/var/run/` — информация о запущенных системных программах;

`/var/spool/` — данные системных задач, ожидающих обработки (например, очереди печати, непрочитанные или неотправленные письма);

`/var/tmp/` — временные файлы, которые должны быть сохранены между перезагрузками.

4.7. Идентификация объектов и ссылки

Идентификация объектов ФС — файлов и каталогов — осуществляется посредством указания пути к объекту. Абсолютный путь всегда однозначно определяет расположения объекта в ФС, поскольку начинается с абсолютного корневого уровня. Часто требуется указать положение объекта относительно какого-либо каталога, тогда используют *относительный путь*, который

никогда не начинается с символа «/», вместо этого сразу указывается имя каталога или имя самого объекта. Относительный путь имеет смысл только в случаях, когда подразумевается конкретный путь к каталогу, относительно которого и нужно определять путь. Таким каталогом обычно выступает текущий каталог процесса (см. гл. 7), но возможны и другие ситуации.

В ОС UNIX путь относительно текущего каталога иногда указывается явно с помощью символа точка «.», за которым следует слеш «/». Обычно это используется при задании имени исполняемого файла, если он находится в текущем каталоге (см. п. 9.2), в остальных случаях обычно этого не требуется. Пример такого пути: `./program` — вызов исполняемого файла `program` из текущего каталога процесса.

Указание двух точек «..» в пути к объекту обозначает переход на уровень выше в иерархии каталогов, т. е. к родительскому каталогу. Например, команда оболочки `cd ..` сменит текущий каталог на уровень выше, а путь `../abc` определяет каталог `abc`, который находится на одном уровне с текущим каталогом. Команда `cd /bin/../` будет эквивалентна команде `cd /`, т. е. в обоих случаях указан корневой уровень.

Абсолютный путь к объекту всегда начинается со знака «/», за исключением случая, когда первым символом указывается знак тильда «~», означающий путь к домашнему каталогу пользователя. В этом случае можно считать, что путь `~/tmp/file`, указывающий на файл `file` в каталоге `tmp/` домашнего каталога пользователя, является абсолютным, но только в отношении данного пользователя; такой символ не применяется в общесистемных настройках.

ОС UNIX имеет богатые возможности по указанию альтернативных путей к объектам ФС, реализуемые с помощью *жестких* и *символьных ссылок*. Для понимания этих механизмов рассмотрим суть организации типовой UNIX-совместимой ФС[1] на физическом уровне.

Файловая система ОС UNIX имеет три основных компоненты:

- *суперблок*,
- *массив индексных дескрипторов*,
- *блоки хранения данных*.

Суперблок — наиболее ответственная область ФС, содержащая необхо-

[1] Пример таких ФС: UFS (Solaris, BSD) и ext2 (Linux).

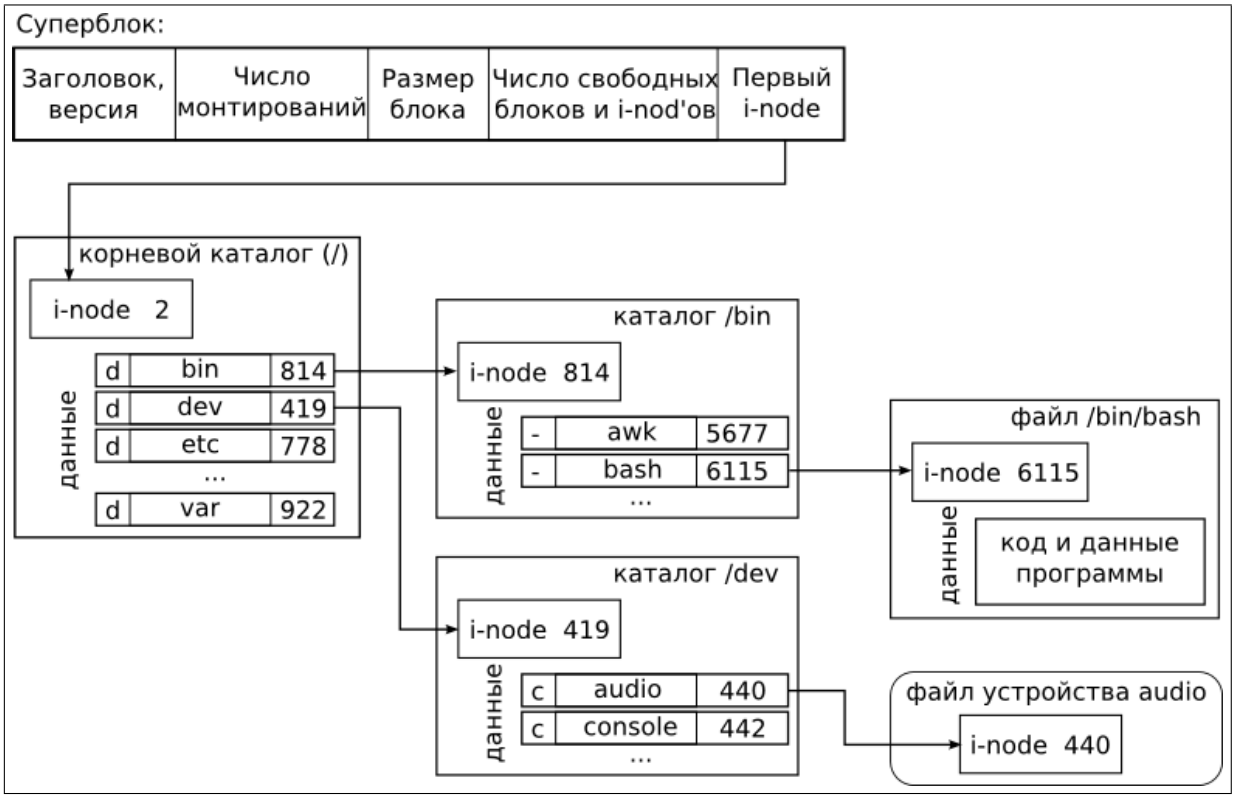

Рис. 9. Индексная файловая система ОС UNIX [5].

димую для работы ФС информацию, а также идентификатор типа ФС. В суперблоке хранится число монтирований раздела, список свободных блоков для хранения данных, список свободных индексных дескрипторов (см. далее) и некоторая другая служебная информация. Суперблок считывается в память при монтировании ФС и находится там до ее отключения.

Каждый файл в ФС имеет закрепленный за ним набор метаданных, называемый *индексным дескриптором* и содержащий информацию о размере файла, владельце файла, правах доступа, времени последнего доступа и модификации, а также информацию о расположении самих данных файла в *блоке хранения данных*. Индексный дескриптор получил название *i-node*[1], произносимое на русском языке как *инод* или *айнод*. Индексные дескрипторы i-node хранятся в *массиве индексных дескрипторов* и не несут информации об имени файла или его расположении в структуре каталогов. Вместо этого организуется таблица соответствия «имя файла — i-node», которая и хранит содержимое каталогов. Каждый i-node имеет свой уникальный номер, используемый для организации таких таблиц. Пример построения иерархической индексной структуры ФС показан на рис. 9.

[1]Возможная версия происхождения этого термина связана со словами *index node* — индексный узел (элемент) или *information node* — информационный элемент.

Отсутствие имени файла как атрибута i-node приводит к возможности создания нескольких имен для одного и того же файла (точнее, индексного дескриптора). *Жесткой ссылкой* (англ. hard link, или hardlink) в ОС UNIX называется имя файла, привязанное к уникальному индексному дескриптору файла. Таким образом, понятия «жесткая ссылка на файл» и «имя файла» являются синонимами.

В i-node хранится число жестких ссылок, т. е. количество имен, на него ссылающихся. Пока это число больше нуля, т. е. существует хотя бы одно имя для i-node, связанные с ним данные хранятся в разделе. Файл вместе с данными считается удаленным после удаления последней ссылки на соответствующий i-node, однако фактически место освобождается, когда его индексный дескриптор перестает использоваться. Жесткие ссылки определены только для файлов и в пределах одного физического раздела. При создании нового файла автоматически создается единственная жесткая ссылка — заданное имя файла.

Символьная ссылка[1] (англ. Symbolic link, или symlink) представляет собой специальный файл, хранящий единственную текстовую строку, которая трактуется как путь к целевому объекту ФС. Во время файловых операций второй группы (см. п.4.3) над символьной ссылкой все действия фактически производятся с объектом, на который она указывает. Целью символьной ссылки может быть любой объект — файл или каталог, независимо от их местоположения в иерархии ФС, а также другая ссылка. Символьная ссылка может указывать и на несуществующий объект, в этом случае попытка выполнить какие-либо действия с ним приведет к ошибке. После появления объекта ссылки файловые операции становятся вновь доступны.

Содержимое символьной ссылки (текстовая строка) может представлять собой как абсолютный, так и относительный путь к объектам. В последнем случае путь отсчитывается от каталога, в котором расположена сама ссылка. В случае задания абсолютного пути к целевому объекту ссылку можно свободно перемещать по ФС, не нарушая ее работоспособности. В случае задания относительного пути ссылка и объект могут перемещаться только вместе, оставаясь на заданном уровне взаимного расположения в иерархии. В пути к объекту ссылки возможно использовании символов «`..`».

[1] В литературе также встречается термин *символическая ссылка*.

Практически символьные ссылки используются для более удобной организации структуры файлов, так как позволяют одному файлу или каталогу иметь несколько имен. Символьные ссылки свободны от некоторых ограничений, присущих жестким ссылкам (последние действуют только в пределах одного раздела и не могут ссылаться на каталоги).

Для создания жестких и символьных ссылок в ОС UNIX существует команда оболочки `ln`, в которой указываются путь к целевому объекту и путь к самой ссылке (имя ссылки). При создании символьной ссылки указывается опция `-s`.

Стандартный API ОС UNIX предоставляет следующие вызовы для работы с ссылками:

- `link()` — создание новой жесткой ссылки на существующий файл,
- `unlink()` — удаление жесткой ссылки и файла, если указанное имя файла было последним,
- `symlink()` — создание символьной ссылки для указанного объекта.

5. Устройства

5.1. Обобщение понятия файла

Файлы играют важнейшую роль в ОС UNIX. В гл. 4 было показано, что типовые операции с некоторыми устройствами и операции с регулярными файлами по своей сути очень близки. Учитывая это, в архитектуре ОС UNIX была заложена концепция, в которой практически все доступные в системе физические устройства (последовательные порты, средства печати, терминалы), а также многие средства межпроцессного взаимодействия (каналы, сокеты, некоторые типы сетевых соединений) и некоторые виртуальные устройства рассматриваются через *файловый интерфейс*.

Согласно этой концепции, практически каждое физическое устройство в ОС UNIX представлено специальным *файлом устройства* в каталоге `/dev` и его подкаталогах при необходимости объединения устройств по дополнительным признакам (см. п. 4.6). Принцип наименования файлов устройств зависит от конкретной реализации ОС. Таким образом, для работы со всеми файлами применяется единый программный интерфейс в виде набора унифицированных системных вызовов. Такая унификация программного интерфейса обеспечивается как *драйверами устройств*, так и средствами самого ядра ОС.

Понятно, что любое устройство имеет свои особенности, которые необходимо учитывать при его использовании. Поэтому для ряда устройств может потребоваться введение дополнительных операций, например, операции изменения режима работы устройства или настройки специфичных для данного устройства параметров. Такие функции выполняет, например, системный вызов `ioctl()`.

В ОС UNIX принята специальная схема кодирования типа объекта ФС в виде одного ASCII-символа, приведенная в табл. 8. Так, регулярный файл кодируется символом «`-`», каталог — символом «`d`» (от англ. directory), а символьная ссылка — символом «`l`». Все остальные символы обозначают специальные файлы устройств или средств межпроцессного взаимодействия и будут рассмотрены далее. Типы файлов «`p`» и «`s`» будут рассмотрены в п. 8.5 и 8.6 соответственно.

Таблица 8. Типы файлов в ОС UNIX.

символ	тип файла
-	регулярный файл
b	файл блочного устройства (Block device)
c	файл символьного устройства (Character device)
d	каталог (Directory)
l	символьная ссылка (symbolic Link)
p	именованный канал FIFO (Pipe)
s	UNIX (IPC) сокет (Socket)

5.2. Символьные и блочные устройства

В ОС UNIX существует два типа специальных файлов устройств:

- *файлы символьных устройств*,
- *файлы блочных устройств*.

Файлы символьных устройств используются для доступа к устройствам, драйверы которых обеспечивают собственную буферизацию и *побайтную передачу данных*, т.е. *последовательный доступ*. Примерами таких устройств могут быть терминалы, принтеры, стримеры (накопители на магнитной ленте), последовательные коммуникационные порты и т.д.

Файлы блочных устройств служат интерфейсом к устройствам, обмен данными с которыми происходит больши́ми фрагментами, называемыми *блоками*[1]. При этом ядро ОС обеспечивает необходимую буферизацию. Классическим примером физических устройств, соответствующих этому типу файлов, являются накопители на магнитных и оптических дисках.

В ОС UNIX существует возможность представления регулярного файла, являющегося объектом последовательного доступа (аналогично символьному устройству), в виде блочного устройства. Это дает возможность, например, представить файл как виртуальный носитель информации (винчестер или CD/DVD диск), отформатировать его под конкретный тип ФС и использовать его впоследствии при монтировании как обычный носитель информации. Для этого существуют специальные «петлевые» виртуальные устройства с общим

[1]Типовой размер одного блока жесткого диска составляет 512 байт.

названием `loop`[1], которые можно указывать в команде монтирования вместо реальных блочных устройств. Файл, содержащий данные ФС, часто называют *образом носителя* и также указывают в команде монтирования. Типичным примером файла-образа носителя является так называемый ISO-образ[2] CD или DVD диска.

5.3. Идентификация и монтирование дисковых разделов

При конфигурировании ФС ОС UNIX необходима возможность идентификации дисков и расположенных на них разделов. И диски, и их разделы представлены в системе файлами блочных устройств, расположенных в общесистемном каталоге `/dev`. Рассмотрим принцип обозначения дисков и разделов в ОС Solaris и Linux.

В ОС Solaris файлы дисковых устройств располагаются в каталоге `/dev/dsk`, формат имени файла имеет вид:

```
cXtYdZsN
```

где `X` — Channel number, номер контроллера (или канала контроллера для IDE дисков), `Y` — Target number, дополнительный номер адресации диска (SCSI ID для SCSI дисков), `Z` — Disk number, номер диска, `N` — номер раздела на диске. Все указанные компоненты нумеруются с нуля, номер раздела может быть от 0 до 7 включительно (всего 8). Раздел с номером 2 имеет особый смысл: он представляет собой весь диск, т. е. является репрезентацией всего диска в целом, со всеми его разделами. Следовательно, на диске на самом деле может быть создано до 7 разделов, а восьмой (раздел 2) всегда адресует сам диск как устройство, на разделе 2 нельзя создать ФС. Раздел 2 используется при указании устройства для утилиты fdisk, в нем хранятся сведения о диске в целом: реальный размер, число цилиндров и т.п. Рассмотрим примеры:

- `/dev/dsk/c0t1d0s1` — второй раздел (s1, нумерация от 0) первого диска (d0) на первом канале (c0), причем этот IDE-диск подключен как Slave (ведомый), поскольку указано t1,

[1] Реально в конкретной реализации ОС UNIX можно ожидать наличие нескольких устройств типа `/dev/loop0`, `/dev/loop1`, и т.д.

[2] Название ISO-образа происходит от стандарта файловой системы для оптических дисков ISO 9660.

- `/dev/dsk/c0t0d0s2` — первый диск (d0) (раздела нет, т.к. указано s2) на первом (t0, возможно единственном для SAS) канале первого (c0) контроллера дисков SCSI (или SAS).

В ОС Linux файлы дисковых устройств располагаются в каталоге `/dev`, формат имени файла имеет вид:

- `hdXY` (для IDE/ATA винчестеров в ядре версий до 2.6.x)
- `sdXY` (для всех видов винчестеров в новых ядрах начиная с 2.6.x)

где `X` — строчная латинская буква (начиная с `a`), обозначающая диск в порядке следования номеров контроллеров, каналов и функции диска на канале, `Y` — цифра, обозначающая номер раздела: от 1 до 4 для *основных разделов* (primary partition), от 5 и выше — для *логических разделов* (logical partition) внутри *расширенного раздела* (extended partition). Для SCSI или SATA винчестеров буква `X` означает канал, к которому подключен диск. Для IDE/ATA винчестеров порядок присвоения имен следующий:

- для версий ядра до 2.6.x (независимо от наличия других винчестеров):
  `hda` — «master» на первом канале IDE,
  `hdb` — «slave» на первом канале IDE,
  `hdc` — «master» на втором канале IDE,
  `hdd` — «slave» на втором канале IDE, и т. д.
- для версий ядра начиная с 2.6.x, в общем случае, порядок зависит от наличия других винчестеров:
  `sda` — «master» на первом канале IDE,
  `sdb` — «master» на втором канале IDE (нумерация в порядке обнаружения дисков независимо от функции устройства «master»/«slave»),
  `sdc` — «slave» на втором канале IDE, и т. д.

Приводы оптических накопителей CD/DVD обозначаются как `srX`, где `X` — номер привода (начиная с 0) в порядке его обнаружения в системе.

Рассмотрим примеры:

- `/dev/sda2` — второй раздел первого обнаруженного диска (IDE/ATA или SATA),
- `/dev/sdc5` — первый логический раздел расширенного раздела третьего обнаруженного диска (IDE/ATA или SATA).

Чтобы данные с раздела диска (т. е. данные в физической ФС) стали доступны ОС, требуется выполнить операцию *монтирования*, т. е. сообщить ОС, в какой каталог существующей ФС следует *отобразить* содержимое нового раздела. Этот каталог называется *точкой монтирования*. После монтирования ФС приобретает целостный вид и для пользователя продолжает выглядеть «монолитно», хотя на самом деле ФС может состоять из нескольких «ветвей» (разделов с различными ФС), каждая из которых присоединена в свою точку монтирования [11].

В системе UNIX информация о статически монтируемых ФС традиционно хранится в файле `/etc/fstab`, представляющем собой текстовый файл в табличном формате, где строки разделяются символами новой строки (ASCII код 10), а столбцы — пробельными символами (пробел или табуляция). Отсюда появился суффикс tab в названии файла.

Рассмотрим фрагмент типового файла `/etc/fstab` для ОС Linux 2.6.x:

```
/dev/sda1   /        ext3     acl,user_xattr        1 1
/dev/sda2   swap     swap     defaults              0 0
/dev/sdb1   /mnt/c   ntfs-3g  fmask=133,dmask=022  0 0
proc        /proc    proc     defaults              0 0
sysfs       /sys     sysfs    noauto                0 0
```

Здесь в первом столбце указывается раздел (имя блочного устройства или сетевой путь для сетевых ФС) или специальное обозначение типа *виртуальной* ФС. Примерами таких виртуальных ФС являются: `proc` — ФС, используемая в ОС UNIX для получения доступа к информации о конфигурации системы, аппаратных ресурсах, выполняемых процессах ядра и пр.; `sysfs` — ФС в ОС Linux, начиная с версии 2.6.x, экспортирующая в пространство пользователя информацию ядра Linux о присутствующих в системе устройствах и драйверах.

Во втором столбце файла `/etc/fstab` указывается точка монтирования (для swap-раздела указывается просто ключевое слово `swap`); в третьем столбце — тип физической ФС согласно принятому обозначению в Linux. В четвертом столбце через запятую указываются дополнительные опции монтирования, например: `acl` — использование системы контроля доступа ACL [6], `defaults` — использование опций по умолчанию, `fmask` — использование

заданной маски *umask* для регулярных файлов данного раздела (см. п. 6.6), `dmask` — использование заданной маски *umask* для каталогов данного раздела (см. п. 6.6), `noauto` — отключение автомонтирования на стадии загрузки ОС. В пятом столбце файла `fstab` указывается частота выполнения операции снятия дампа с ФС [27, 28]. В последнем, шестом столбце указывается порядок проверки файловых систем программой `fsck`. Файловые системы с одинаковым значением этого поля проверяются, если это возможно, параллельно.

В ОС UNIX существуют следующие команды управления разделами:

- `mount` — монтирование разделов с указанием опций монтирования;
- `umount` — отсоединение примонтированных разделов;
- `fdisk` — управление разметкой всего жесткого диска (или другого носителя информации), включая создание и удаление разделов,
- семейство команд `mkfs.*` для создания различных физических ФС на разделах (команды форматирования),
- семейство команд `fsck.*` для проверки целостности физических ФС.

Информацию о примонтированных в данный момент разделах можно узнать с помощью команды `mount` (без аргументов), из текстового файла `/etc/mtab` или через функцию `getmntent()`.

5.4. Виртуальные устройства

Помимо файлов, соответствующих внешним устройствам, в ОС UNIX существует несколько стандартных файлов *виртуальных устройств* [5]. Эти файлы могут передавать и принимать от пользовательских процессов специальные данные, например, из символьного устройства `/dev/zero` можно прочитать только нули, сколько бы процесс не читал данные из этого файла. Вот список наиболее распространенных виртуальных устройств:

`/dev/console` — устройство соответствует активной в данный момент терминальной линии (виртуальной консоли) (см. гл. 9);

`/dev/null` — «черная дыра» — любая информация, записанная в этот файл, пропадает безвозвратно; обычно используется для поглощения ненужного вывода программ;

`/dev/random`, `/dev/urandom` — устройства, генерирующие соответственно случайные и псевдослучайные данные;

`/dev/stdin`, `/dev/stdout`, `/dev/stderr` — устройства, соответствующие трем стандартным потокам ввода-вывода для каждого из процессов системы, которые подробно будут рассмотрены в гл. 8;

`/dev/zero` — устройство генерирует нулевые байты;

`/dev/loopN` (`N` — целое число) — устройства, позволяющие организовать доступ к файлу как к блочному устройству. Применяются, в частности, при монтировании файла-образа ФС или файла, содержащего сразу несколько разделов.

6. Многопользовательская среда

6.1. Пользователи и группы

ОС UNIX изначально проектировалась как многопользовательская система, предоставляющая свои ресурсы одновременно или поочередно нескольким пользователям, подключающимся к системе непосредственно через терминалы или удаленно через сетевые соединения. Для такой ОС требовалась организация совершенного и гибкого механизма контроля доступа на уровне самой ОС, способного разграничивать доступ пользователей к ресурсам файловой системы и аппаратным устройствам. В результате была разработана достаточно простая модель доступа, основанная на *субъектно-объектной модели безопасности* и *статической авторизации*, сохранившаяся практически без изменений во всех современных версиях ОС UNIX [5].

В системах контроля доступа используются три основных понятия: *объект*, *субъект* и *режим доступа* [5]. *Объектом* системы называется любой ее идентифицируемый ресурс (например, файл, процесс или устройство). *Режим доступа* к объекту системы — это перечисление допустимых операций над этим объектом (например, чтение или запись). *Действительным субъектом* системы называется любая сущность, способная выполнять действия над объектами (имеющая к ним доступ). Действительному субъекту системы соответствует некоторая абстракция, на основании которой принимается решение о предоставлении доступа к объекту или об отказе в доступе. Такая абстракция называется *номинальным субъектом*. Например, сотрудник организации — действительный субъект, а его пропуск в здание — номинальный. Другим примером может служить злоумышленник, прокравшийся в секретную лабораторию с украденной картой доступа — он является действительным субъектом, а карта — номинальным [5].

В ОС UNIX роль номинального субъекта безопасности играет *пользователь*. Каждому пользователю выдается уникальное входное имя (*login* или *username*). Каждому входному имени соответствует уникальный идентификатор пользователя UID (от англ. User Identifier) — целое неотрицательное число, которое в итоге и используется системой в качестве идентификатора номинального субъекта для определения прав доступа.

Каждый пользователь ОС UNIX входит в одну или несколько групп. *Груп-*

па — это список пользователей системы, имеющий собственный уникальный идентификатор GID (от англ. Group Identifier) — целое неотрицательное число. Поскольку группа объединяет несколько пользователей системы, в терминах политики безопасности она соответствует понятию *множественный субъект*. Идентификатор GID — это идентификатор множественного субъекта. Таких идентификаторов (групп) у номинального субъекта (пользователя) может несколько. Таким образом, одному UID соответствует список из одного или более GID.

Роль действительного (работающего с объектами) субъекта играет процесс. Каждый процесс в ОС UNIX имеет следующие атрибуты, связанные с системой контроля доступа:

- UID — идентификатор пользователя, запустившего процесс;
- EUID — *эффективный идентификатор пользователя* (от англ. Effective), значение которого используется при определении прав доступа процесса, будет рассмотрен в п. 6.5;
- GID — идентификатор основной группы пользователя, запустившего процесс;
- EGID — *эффективный идентификатор группы*, который используется при определении прав доступа процесса, будет рассмотрен в п. 6.5.

Процесс, порожденный некоторым процессом пользователя, наследует его атрибуты UID и GID. Подробно процессы будут рассмотрены в гл. 7.

6.2. Суперпользователь

В ОС UNIX существует особый пользователь с именем root, именуемый также *суперпользователь*; значение его идентификатора UID всегда равно нулю. Пользователь root играет роль особого доверенного субъекта ОС и имеет право на выполнение всех без исключения операций. Уровень его привилегий соответствует уровню привилегий ядра ОС, т. е. самому высокому. Система никогда не проверяет права доступа процесса, запущенного от имени суперпользователя. Бóльшая часть системных ресурсов (файлы и каталоги ФС, некоторые устройства) доступна для модификации только суперпользователю.

Системный администратор ОС UNIX должен использовать права суперпользователя (т. е. работать от имени root) только при выполнении операций,

связанных с *администрированием ОС*, т. е. управлением важнейшими характеристиками системы, включая конфигурирование аппаратных ресурсов и системного программного обеспечения, формирование политики контроля доступа. В остальных случаях необходимо использовать учетную запись уровня обычного пользователя. Работа с повышенными привилегиями требует особого внимания: выполнение неверной команды может привести к выходу системы из строя или утрате информации.

Для смены пользователя в текущем сеансе работы используется команда `su`, которая без аргументов выполняет переключение на суперпользователя root, запрашивая его пароль. Для выполнения однократных действий с уровнем привилегий пользователя root в ОС UNIX существует команда `sudo`, которая в качестве аргумента принимает соответствующую командную строку. Программа `sudo` запрашивает пароль суперпользователя и, в случае его успешного ввода, выполняет указанные действия. Перечень пользователей, авторизованных на выполнение команды `sudo`, определяется в системном файле `/etc/sudoers` с применением специальных синтаксических правил. В графических оболочках системы X Window (см. гл. 9.4) могут применяться альтернативные программы, например, `gnomesu` для графических сред Gnome и XFCE.

6.3. Учетные записи

С каждым зарегистрированным в системе пользователем связано понятие *учетной записи*[1] — блока данных, включающего следующие основные атрибуты пользователя: имя и идентификатор UID, основную группу, пароль (хранится всегда в зашифрованном виде) и путь к домашнему каталогу пользователя. Информация об учетных записях хранится в специализированной системной базе данных, функцию которой в ОС UNIX традиционно выполняет файл `/etc/passwd`, представляющий собой текстовый файл, в котором каждому пользователю соответствует одна строка с семью полями, разделяемыми двоеточиями в следующем формате:

```
login:x:UID:GID:user_info:home_dir:shell
```

Здесь `user_info` — дополнительная информация, например, фамилия и имя пользователя, `home_dir` — полный путь к домашнему каталогу пользова-

[1] В зарубежной литературе используется английский термин *user account*.

теля, `shell` — полный путь к командной оболочке, запускаемой при входе пользователя в систему. В качестве примера приведем фрагмент файла `/etc/passwd`:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
ivan:x:101:101:Иван Петров:/home/ivan:/bin/bash
```

Группа GID, указанная в файле `/etc/passwd`, является *основной*, или *первичной группой* данного пользователя. Пользователь может входить в состав других групп, которые для него будут считаться *вторичными*. Все новые файлы и каталоги, создаваемые процессами этого пользователя, в качестве группы-владельца будут получать основную группу данного пользователя[1].

С целью исключения возможности входа в систему конкретного пользователя в его учетной записи указывается «неправильная» командная оболочка, такая как `/bin/false` (см. пример выше), работа в которой невозможна[2]. Такой приём применяется для учетных записей так называемых *псевдопользователей*, от имени которых выполняются некоторые системные задачи и которыми не пользуются реальные люди.

Пароли на вход в систему пользователей в современных версиях ОС UNIX не хранятся в открытом виде, хранятся только их *хэш-значения*[3]. Даже если злоумышленник получит это хэш-значения, ему придется подбирать пароль, применяя данную одностороннюю функцию к различным словам и сравнивая полученный результат с исходным хэш-значением. Обычно хэш-значения хранятся в специальном файле `/etc/shadow`, доступ к которому разрешен только системе, так что перебор вообще не возможен.

[1] Такое правило используется в ветви систем UNIX System V, но его можно переопределить, см. п.6.5.

[2] Команда false немедленно завершается и возвращает ошибку.

[3] Хэширование (англ. hashing) — одностороннее преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования выполняются при помощи хэш-функций или функций свертки. Их результаты также называются хэшем, хэш-кодом или дайджестом сообщения (англ. message digest).

Аналогичным образом информация о группах пользователей хранится в файле `/etc/group`. Каждой строке файла соответствует информация о группе: ее имя, числовой идентификатор и список пользователей, входящих в эту группу. Как было сказано выше, пользователь может иметь и вторичные группы. Таким образом, факт членства пользователя во вторичных группах отражается в файле `/etc/group`.

Добавление и удаление пользователей, а также изменение информации о пользователях разрешено только суперпользователю и может производиться прямым редактированием этих файлов, однако более корректным способом является использование специальных системных команд. Во-первых, такой способ, как правило, будет более простым и универсальным. Во-вторых, нет никаких гарантий, что в конкретной ОС информация о пользователях и группах не будет храниться еще где-то кроме этих файлов.

В ОС UNIX существуют следующие системные команды управления учетными записями пользователей и групп:

- `useradd` — создание новой учетной записи пользователя с указанием необходимых атрибутов, а также создание домашнего каталога[1];
- `usermod` — изменение атрибутов учетной записи пользователя, включая первичную и вторичные группы;
- `userdel` — удаление учетной записи пользователя, а также удаление пользовательских файлов;
- `passwd` — изменение собственного пароля пользователя или пароля произвольного пользователя в случае запуска от имени root;
- `groupadd` — создание новой группы с указанием необходимых атрибутов;
- `groupmod` — изменение атрибутов группы с возможностью изменения состава пользователей;
- `groupdel` — удаление записи о группе.

[1] В ОС Solaris 10 одной команды `useradd` недостаточно для создания полноценной учетной записи пользователя.

6.4. Дискреционная система контроля доступа

ОС UNIX поддерживает единообразный механизм контроля доступа к объектам ФС. Любой процесс, как *субъект* такой системы контроля, может получить доступ к некоторому *объекту* ФС в том и только в том случае, если права доступа, описанные для объекта, соответствуют возможностям данного процесса. Система контроля доступа, в которой для каждой пары «субъект-объект» задается *режим доступа*, т. е. явное перечисление допустимых типов доступа (на чтение, запись и т. д.), называется *избирательной* или *дискреционной* (от англ. discretionary).

За каждым объектом ФС (файлом или каталогом) закрепляется один *владелец-пользователь* данного ресурса и один *владелец-группа*, соответственно объект получает атрибуты UID и GID. В рамках системы контроля доступа для каждого объекта организуются три категории субъектов:

1) *владельцы-пользователи*, т. е. процессы, EUID которых совпадает с UID владельца объекта;
2) *члены владельца-группы*, т. е. процессы, пользователь которых (согласно EUID) входит в группу GID объекта;
3) *остальные пользователи*, не попавшие в первые две категории.

На уровне ФС определяются три права доступа:

- *чтение*, обозначается буквой «`r`» (от англ. *read*),
- *запись*, обозначается буквой «`w`» (от англ. *write*),
- *исполнение*, обозначается буквой «`x`» (от англ. *execution*).

Для файлов право на чтение («`r`») дает доступ к содержащейся в файле информации, а право записи («`w`») — возможность модификации содержимого файла. Право исполнения («`x`») дает возможность запускать данный файл как программу (бинарный файл или сценарий) из командной оболочки или через системный вызов `exec()` (см. п. 7.3). Для исполнения сценария также необходимо иметь право чтения («`r`»).

Для каталогов право на чтение («`r`») дает доступ к перечню объектов этого каталога (а не к самим объектам !), а право записи («`w`») — возможность модификации этого перечня, т. е. возможность операций *добавления*, *удаления* или *переименования* объектов каталога. Право исполнения («`x`»)

для каталога означает в принципе возможность доступа к его объектам и их метаданным, т. е. доступ к дескрипторам i-node. Процесс должен иметь права «`x`», если он хочет сделать данный каталог текущим (выполнить команду `cd`). Это право также необходимо для модификации перечня объектов каталога и обычно устанавливается совместно с правом записи («`w`»).

Наличие или отсутствие права физически кодируется одним информационным битом («0» — право отсутствует, «1» — право установлено), а в символьной записи — соответствующей латинской буквой при установленном праве или символом «`-`» — при отсутствии права. В зависимости от типа объекта (файла или каталога) эти права определяют возможность разных операций. Права устанавливаются независимо друг от друга, но для выполнения некоторых операций требуется совместное задание нескольких прав.

С каждой из трех категорий субъектов связан набор из трех рассмотренных прав доступа. В итоге, для каждого объекта ФС формируется режим доступа, состоящий из 9 независимо устанавливаемых битов, по три на каждую категорию (см. рис. 10). При определении возможности доступа к объекту ОС проверяет соответствие субъекта одной из трех категорий *последовательно в указанном выше порядке*. Решающей становится первая подходящая категория с ее тремя правами доступа, а *права доступа в остальных двух категориях игнорируются*.

В ОС UNIX приняты строковая и числовая восьмеричная формы записи режима доступа. В строковом формате права записываются последовательно в виде указанных выше букв в порядке `r`,`w` и `x`. Порядок следования категорий соответствует выше рассмотренному: владелец-пользователь, владелец-группа и остальные. В числовой форме режим доступа записывается в виде трехзначного восьмеричного числа, в котором права для каждой категории кодируются цифрой согласно табл. 9, при этом категории следуют в порядке убывания веса разряда.

Строковое представление режима доступа применяется, например, в выводе команды `ls -l`, отображающей содержимое каталога (см. рис. 10). Дополнительно к 9 символам строки слева добавляется однобуквенный код типа файла из возможных значений, указанных в табл. 8.

Наличие режима доступа у файлов устройств (см. гл. 5) позволяет систем-

Рис. 10. Режим доступа для трех категорий.

Таблица 9. Восьмеричное кодирование режима доступа.

Представление режима доступа		
строковое	двоичное	восьмеричное
`---`	000	0
`--x`	001	1
`-w-`	010	2
`-wx`	011	3
`r--`	100	4
`r-x`	101	5
`rw-`	110	6
`rwx`	111	7

ному администратору таким образом контролировать доступ пользователей к аппаратному обеспечению. Например, чтобы пользовательский процесс имел возможность ввода информации из устройства (например, последовательного порта), необходимо иметь право чтения (`r`) для соответствующего файла, а для вывода информации через устройство (например, звука через аудиоплату) — право записи (`w`). Это еще раз демонстрирует простоту и универсальность важнейшей концепции ОС UNIX— «каждое устройство — файл».

В ОС UNIX существуют следующие системные команды для установки владельцев и режима доступа объектов ФС:

- `chown` — установка владельца-пользователя данного объекта, может быть выполнена только суперпользователем root;
- `chgrp` — установка владельца-группы данного объекта, может быть выполнена суперпользователем root, а также обычным пользователем в слу-

чае, когда он является владельцем объекта и входит в состав указываемой группы;

- `chmod` — установка режима доступа к объекту, может быть выполнена суперпользователем или владельцем-пользователем данного объекта.

Отметим основные особенности дискреционной системы контроля доступа в ОС UNIX:

1) для доступа к любому объекту (файлу или каталогу) необходимо наличие права доступа (**x**) *на всех вышестоящих каталогах*, начиная с корневого уровня;
2) наличие права записи в файл не дает права на его удаление или переименование, поскольку для этого необходимо право записи на каталог, однако возможно уничтожение или модификация информации в файле;
3) отсутствие права записи в файл не дает защиты от его удаления, поскольку для этого требуется отсутствие права записи или доступа на каталог, в котором расположен файл[1];

[1]Исключение составляет случай для ОС Solaris, когда для данного каталога установлен Sticky-бит, см. п. 6.5.

4) установка права доступа и отмена права чтения на каталог (режим **--x** или **-wx**) приводит к эффекту «спрятанных» объектов каталога, доступ к которым возможен только при наличии у пользователя информации об их именах;

5) для возможности манипулирования объектами каталога (создания, переименования и удаления) необходимо кроме права записи (**w**) иметь также и право доступа (**x**) в каталог, поскольку при этих операциях изменяется не только перечень объектов каталога, но и атрибуты индексных дескрипторов самих объектов;

6) классическая дискреционная система доступа не дает возможности «спрятать» *отдельные* объекты каталога, равно как и защитить их от удаления.

6.5. Дополнительные атрибуты доступа

Кроме основных девяти битов режима доступа, у каждого объекта ФС также имеются 3 дополнительных бита:

- *Set UID* или *SUID* — бит установки EUID процесса,
- *Set GID* или *SGID* — бит установки EGID процесса,
- *Sticky bit* — бит сохранения образа исполняемого бинарного файла в памяти.

Эти биты образуют дополнительную тройку атрибутов, которая также кодируется восьмеричной цифрой и ставится в старший значащий разряд (записывается слева) результирующего режима доступа. Порядок бит следующий: SUID — старший, SGID — средний, Sticky bit — младший. Таким образом, полное числовое представление режима доступа составляет четырехзначное восьмеричное число. Реакция системы на установку дополнительных атрибутов зависит от типа объекта (файл или каталог).

Для файлов установка дополнительных атрибутов имеет смысл только при установленном праве исполнения (**x**). Установка Sticky-бита для исполняемых файлов в ранних версиях ОС UNIX применялась для сохранения образа исполняемого файла в оперативной памяти с целью уменьшения времени его последующей загрузки. Такой приём применялся для наиболее часто используемых программ [3]. В большинстве современных версий ОС UNIX на файлах этот бит будет игнорироваться.

Наличие установленных битов SUID и SGID на *исполняемых бинарных файлах* влечет возможные изменения привилегий порождаемого процесса. Запуск бинарного исполняемого файла с установленным битом SUID порождает процесс, EUID которого будет заменен на UID владельца-пользователя данного файла. Если же бит SUID сброшен, числовые атрибуты процесса EUID и UID будут равны. Запуск бинарного исполняемого файла с установленным битом SGID порождает процесс, EGID которого будет заменен на GID владельца-группы данного файла. Значение атрибута EGID, отличное от GID данного процесса, будет означать возможность получения процессом прав доступа к объекту ФС по категории группы, если владелец-группа объекта совпадёт с EGID. Если бит SGID для исполняемого файла сброшен, числовые атрибуты процесса EGID и GID будут равны.

Установка SUID бита используется для некоторых системных программ, которые должны запускаться обычными пользователями, но при этом иметь привилегии root с целью временного и контролируемого доступа к важным системным ресурсам. Классическим примером является утилита `passwd`, имеющая путь `/bin/passwd`. На данном файле, владельцем которого является root, устанавливается бит SUID с целью дать всем запустившим утилиту пользователям доступ по записи к системным файлам `/etc/passwd` и `/etc/shadow`, поскольку это необходимо для изменения собственного пароля пользователя. Понятно, что требования по надежности и безопасности для таких программ крайне высокие.

Следует отметить, что установка битов SUID и SGID на *исполняемые файлы-сценарии* не приводит к изменению атрибутов EUID и EGID процесса, поскольку последний порождается загрузкой *исполняемого бинарного файла* интерпретатора сценария (например, `/bin/bash`), а сам файл сценария, в любом случае, представляет лишь текстовый файл, который прочитывается этим интерпретатором.

Для каталогов бит SGID влияет на правило установки владельца-группы для вновь создаваемых объектов внутри каталога. По умолчанию, для ОС UNIX ветви System V владельцем-группой создаваемых объектов становится основная группа пользователя, запустившего процесс. Для систем BSD существует иное правило: владелец-группа наследуется от родительско-

го каталога. Установка бита SGID на каталог в ОС Solaris и Linux позволяет «включить» правило наследования владельца-группы, т. е. имитировать поведение систем ветви BSD для данного каталога. Бит SUID для каталогов не играет никакой роли.

Установка Sticky-бита одновременно с правом записи и доступа на каталог позволяет создать дополнительную защиту: из такого каталога пользователь может удалять только те объекты, владельцем которых он является [3]. Однако полный эффект от установки Sticky-бита в различных версиях ОС UNIX может различаться. Так, в ОС Linux в каталоге с установленным Sticky-битом (а также правом на запись и доступ) право удаления объекта имеет:

1) владелец каталога,
2) владелец объекта,
3) суперпользователь root.

В ОС Solaris к этому списку добавляется также *процесс, имеющий право записи (`w`) на сам удаляемый объект*. Очевидно, что Sticky-бит имеет смысл устанавливать на каталог, в котором согласно политике безопасности необходимо разрешить его модификацию всем, т. е. установить для всех категорий право записи (`w`). Классическим примером такого каталога является `/tmp` — общесистемный каталог временных файлов.

6.6. Режим доступа по умолчанию

В ОС UNIX существует определенное правило формирования режима доступа для вновь создаваемых объектов ФС. В принципе, по умолчанию имеются следующие стандартные режимы *полного доступа*:

- 0666 (`rw-rw-rw-`) — для файлов,
- 0777 (`rwxrwxrwx`) — для каталогов.

Эти режимы модифицируются при помощи параметра *umask* (от англ. User file creation mode Mask — маска режима создания пользовательских файлов), изменяющего права доступа, которые присваиваются новым объектам по умолчанию. Результирующий режим доступа `M` вычисляется при помощи следующих побитовых операций:

M = M' and (not umask),

где M' — полный режим доступа, and — побитовая операция «И», not — побитовая операция инверсии «НЕ». Фактически, *umask* указывает, какие биты следует сбросить в выставляемых правах — каждый установленный бит *umask* запрещает выставление соответствующего бита в режиме полного доступа.

Для получения или установки параметра *umask* применяется команда оболочки `umask`. Без параметров команда выдает текущее значение маски, а при указании параметра — устанавливает маску, при этом используется восьмеричное представление. Команда `umask` влияет на все дочерние процессы, исполняемые в данной оболочке. Стандартным значением маски является 0022, при этом результирующие режимы будут следующие:

- 0666 and (not 0022) = 0644 — для файлов,
- 0777 and (not 0022) = 0755 — для каталогов.

Параметры, аналогичные *umask*, применяются также в качестве опций монтирования файловых систем, часто отдельно для файлов (*fmask*) и каталогов (*dmask*), в этом случае их действие распространяется на весь монтируемый раздел (см п. 5.3).

7. Процессы

7.1. Режимы и состояния процесса

Процесс является важнейшим понятием в ОС UNIX и основным объектом любой многозадачной ОС. В гл. 1 было введено понятие процесса, а в гл. 3 рассмотрены аппаратные средства организации защищенного многозадачного режима. Данный параграф посвящен более детальному рассмотрению процесса как объекта ОС.

Существует два основных режима (или уровня) выполнения процесса [3]:

- *режим задачи (пользователя)*,
- *режим ядра*.

В *режиме задачи* процесс выполняет инструкции прикладной программы, допустимые на непривилегированном уровне защиты процессора, при этом процессу недоступны системные структуры данных. Когда процессу требуется получение каких-либо услуг ядра ОС, он делает *системный вызов*, который выполняет инструкции ядра, находящиеся на привилегированном уровне. Несмотря на то, что выполняются инструкции ядра, это происходит от имени процесса, сделавшего системный вызов. Выполнение процесса при этом переходит в *режим ядра*. Таким образом, ядро системы защищает собственное адресное пространство от доступа прикладного процесса, который может нарушить целостность структур данных ядра и привести к разрушению ОС. Более того, часть процессорных инструкций, например, изменение регистров, связанных с управлением памятью, могут быть выполнены только в режиме ядра [3].

В режиме задачи процессы имеют доступ только к своим собственным инструкциям и данным, но не к инструкциям и данным ядра (либо других процессов). Однако в режиме ядра процессам уже доступны адресные пространства ядра и пользователей. Например, виртуальное адресное пространство процесса может быть поделено на адреса, доступные только в режиме ядра, и на адреса, доступные в любом режиме [29, 30].

Некоторые машинные команды являются привилегированными и вызывают возникновение ошибок при попытке их использования в режиме задачи. Например, в машинном языке может быть команда, управляющая реги-

стром состояния процессора; процессам, выполняющимся в режиме задачи, она недоступна.

Кроме основных режимов задачи и ядра, процессы могут находиться в других логических состояниях и переходить между ними в соответствии с установленными правилами перехода [3, 29, 30], при этом информация о состоянии сохраняется в таблице процессов и в адресном пространстве процесса. В табл. 10 перечислены возможные состояния процесса согласно [3]. Необходимо отметить, что не все процессы проходят через все приведенное множество состояний.

Таблица 10. Перечень возможных состояний процесса согласно [3].

№	*Состояние процесса*
1	Процесс выполняется в режиме задачи. При этом процессором выполняются прикладные инструкции данного процесса.
2	Процесс выполняется в режиме ядра. При этом процессором выполняются системные инструкции ядра операционной системы от имени процесса.
3	Процесс не выполняется, но готов к запуску, как только планировщик выберет его (состояние runnable). Процесс находится в очереди на выполнение и обладает всеми необходимыми ему ресурсами, кроме вычислительных.
4	Процесс находится в состоянии сна (состояние asleep), ожидая недоступного в данный момент ресурса, например завершения операции ввода/вывода.
5	Процесс возвращается из режима ядра в режим задачи, но ядро прерывает его и производит переключение контекста для запуска более приоритетного процесса.
6	Процесс только что создан вызовом `fork()` и находится в переходном состоянии: он существует, но не готов к запуску и не находится в состоянии сна.
7	Процесс выполнил системный вызов `exit()` и перешел в состояние *зомби* (zombie, defunct). Как такового процесса не существует, но остаются записи, содержащие код возврата и временную статистику его выполнения, доступную для родительского процесса. Это состояние является конечным в жизненном цикле процесса.

7.2. Контекст процесса

Каждому процессу соответствует *контекст* или *образ*, включающий в себя следующие три компонента [29, 30]:

1) *пользовательский контекст* — содержимое виртуального адресного пространства, сегментов программного кода, данных, стека, разделяемых сегментов и сегментов файлов, отображаемых в виртуальную память;
2) *регистровый контекст* — содержимое аппаратных регистров (регистр

счетчика команд, регистр состояния процессора, регистр указателя стека и регистры общего назначения), на платформе x86-IA32 регистровый контекст сохраняется в так называемом *сегменте состояния задачи* (англ. TSS — task state segment) [21];

3) *системный контекст* — структуры данных ядра ОС, связанные с процессом.

Системный контекст процесса состоит из двух частей: *статической* и *динамической*. Для каждого процесса имеется одна статическая часть системного контекста и переменное число динамических частей. Статическая часть контекста включает следующие атрибуты процесса [5]:

Идентификатор процесса PID (от англ. Process Identifier) — уникальный номер, идентифицирующий процесс в системе. По сути, это номер строки в таблице процессов — специальной внутренней структуре ядра ОС, хранящей информацию о процессах. В любой момент времени ни у каких двух процессов номера не могут совпадать, однако после завершения процесса его номер освобождается и может быть в дальнейшем использован для идентификации любого вновь запущенного процесса.

Идентификатор родительского процесса PPID (от англ. Parent Process Identifier). В ОС UNIX процессы выстраиваются в иерархию — новый процесс может быть создан только одним из уже существующих процессов, который выступает для него родительским. Очевидно, что в такой схеме должен присутствовать один процесс с особым статусом: он должен быть порожден ядром операционной системы и будет являться родительским для всех остальных процессов в системе. В ОС UNIX такой процесс имеет собственное имя — `init`. Подробнее этот процесс рассмотрен в гл. 10.

Идентификаторы пользователя и группы. Идентификаторы пользователя UID и группы GID, от имени которых запущен процесс — наследуются от родительского процесса; а также эффективный идентификатор пользователя EUID и эффективный идентификатор группы EGID, которые фактически используются ядром ОС для определения прав доступа процесса в системе. В общем случае, для процесса значения UID и GID могут не совпадать с EUID и EGID соответственно. Атрибуты EUID и

EGID были рассмотрены в гл. 6.

Идентификаторы группы процессов (PGID) и сеанса (SID), рассматриваемые в п. 10.3.

Приоритет процесса — число, используемое при планировании исполнения процесса в операционной системе. Подробнее о приоритете будет рассказано в п. 7.6.

Таблица дескрипторов открытых файлов — список структур ядра, описывающий все файлы, открытые этим процессом для ввода-вывода.

Состояние процесса. Каждый процесс в любой момент времени находится в одном из нескольких определенных состояний, перечень которых приведен в табл. 10.

Терминальная линия (TTY). Терминал или псевдотерминал (см. п. 9.1), связанный с процессом. С этим терминалом по умолчанию связаны стандартные потоки: входной, выходной и поток сообщений об ошибках. Стандартные потоки будут рассмотрены в гл. 8.

Временны́е параметры процесса, включая *время выполнения в режиме задачи*, *время выполнения в режиме ядра*.

Переменные окружения — набор строковых переменных, определяющих окружение процесса (см. п. 7.4).

Другие системные параметры, включая сигналы, ожидающие доставки; размер адресного пространства процесса; указатель на локальную таблицу дескрипторов сегментов; список областей памяти процесса; код возврата, передаваемый родительскому процессу и пр.

Динамическая часть контекста процесса — это один или несколько стеков, которые используются процессом при выполнении в режиме пользователя и в режиме ядра (в процессе прерываний и системных вызовов) [31].

В ОС UNIX получить информацию о процессах можно с помощью следующих команд:

- `ps` — выводит в терминал информацию о процессах данного терминала, данного пользователя или сведения по всем процессам в системе в заданном формате, при этом возможно выбрать какие именно атрибуты процесса будут отображаться;

- `top` — выводит в терминал информацию о процессах системы в интерактивном режиме, включая возможность сортировки по степени нагрузки на процессор (режим работы по умолчанию), размеру занимаемой памяти (физической или виртуальной) и другим атрибутам;
- `pgrep` — команда аналогична `ps`, но позволяет отображать только те процессы, командная строка вызова которых соответствует указываемому *регулярному выражению*[1]; в простейшем случае с помощью данной команды можно найти PID процесса по имени исполняемого файла;
- `pstree` — выводит в терминал информацию о «родственных» связях процессов в системе в виде дерева с применением псевдографических символов.

Стандартный API ОС UNIX предоставляет следующие вызовы для получения информации о процессах:

- `getpid()` — возвращает PID текущего процесса;
- `getppid()` — возвращает PID родительского процесса (PPID).

Виртуальная файловая система `/proc` (см. п.4.6) предоставляет подробную информацию о каждом процессе в текстовом формате, при этом каждому процессу соответствует вложенный каталог, имя которого соответствует PID процесса в десятичной системе. Внутри такого каталога располагаются несколько десятков файлов, хранящих различную информацию. Например, файл `comm` содержит команду вызова процесса, файл `environ` — переменные окружения, `loginuid` — UID пользователя, а подкаталог `fd` хранит все используемые процессом файловые дескрипторы. Таким образом, стандартный API ОС UNIX для получения информации о процессе предполагает чтение соответствующих текстовых файлов.

7.3. Создание и завершение процесса

Каждый процесс в ОС UNIX, за исключением первоначального (нулевого), является объектом, создаваемым в результате выполнения системного вызова `fork`. Процесс, инициировавший вызов `fork()`, называется *родительским* (процесс-родитель), а вновь создаваемый процесс называется *порожденным*

[1] Регулярные выражения (англ. regular expressions) — формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов.

или дочерним (процесс-потомок). Каждый процесс имеет одного родителя, но может породить множество дочерних процессов [29, 30].

Синтаксис вызова функции `fork()` следующий:

```
pid = fork();
```

В результате выполнения функции `fork()` контекст и того, и другого процессов совпадает во всем, кроме возвращаемого значения переменной `pid` и значений PID и PPID. *Для родительского процесса в* `pid` *возвращается идентификатор порожденного процесса, для порожденного — нулевое значение.*

В ходе выполнения функции ядро производит следующие действия:

1) отводит место в таблице процессов под новый процесс;
2) присваивает порождаемому процессу уникальный код идентификации;
3) делает логическую копию контекста родительского процесса. Поскольку те или иные составляющие процесса, такие как область команд, могут разделяться другими процессами, ядро может иногда вместо копирования области в новый физический участок памяти просто увеличить значение счетчика ссылок на область;
4) увеличивает значения счетчика числа файлов, связанных с процессом, как в таблице файлов, так и в таблице индексов;
5) возвращает родительскому процессу код идентификации порожденного процесса, а порожденному процессу — нулевое значение.

Процесс-потомок наследует все атрибуты своего родителя, включая UID и GID, следовательно, его права доступа в системе будут такими же, как и у процесса-родителя. В ряде случаев некоторым системным программам (например, программам `login`, `su` или серверу удаленного доступа `sshd`) бывает необходимо изменить значения UID, EUID, GID и EGID порождаемых процессов. Для этого используются системные вызовы `setuid()`, `seteuid()`, `setgid()`, `setegid()`, соответственно. Выполнение этих вызовов для изменения значений атрибутов, отличных от атрибутов вызывающего их процесса, возможно только для процессов с привилегиями суперпользователя (EUID=0).

Завершение выполнения процесса инициирует системный вызов `exit()`,

аргументом которого является целое число типа `int`, передаваемое в родительский процесс как *код завершения*[1] процесса-потомка. В ОС UNIX принято, что в случае успешного завершения процесс возвращает код 0, иные значения трактуются как признак завершения процесса с какой-либо ошибкой.

Процессы могут вызывать функцию `exit()` как в явном, так и в неявном виде. Так, во всех программах на языке Си компилятором автоматически создается специальная процедура startup, вызывающая функцию `main()`. При завершении выполнения программы, т.е. на выходе из функции `main()`, в процедуре startup выполняется вызов `exit()` с кодом завершения, который был указан в операторе `return` функции `main()`.

Процесс-родитель может синхронизировать продолжение своего выполнения с моментом завершения процесса-потомка, если воспользуется системной функцией `wait()`. Синтаксис вызова функции:

```
pid = wait(status_addr);
```

где `pid` — значение PID прекратившего свое существование потомка, `status_addr` — адрес переменной целого типа, в которую будет помещено возвращаемое функцией `exit()` значение. Функция `wait()` приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не прекратит выполнение или до появления сигнала[2], который либо завершает текущий процесс, либо требует вызвать функцию-обработчик.

Возможны ситуации, когда дочерний процесс к моменту вызова функции `wait()` уже завершился. Такой дочерний процесс называется *процессом-зомби*, или просто *зомби* (англ. zombie process, а также англ. defunct process); он еще присутствует в списке процессов ОС, чтобы дать родительскому процессу возможность получить код его завершения. В таком случае функция `wait()` немедленно возвращается, а системные ресурсы, связанные с дочерним процессом, освобождаются. Следующая программа показывает, каким образом может быть создан процесс-зомби:

[1] Иногда также используется термин *код возврата* или *статус завершения* процесса.

[2] Подробнее о сигналах см. п.8.2.

```
1 int main() {
2   pid_t p;
3   p = fork();
4   if (p > 0) sleep (10);
5   else exit (0);
6   return 0;
7 }
```

В этом примере после выполнения вызова `fork()` родительский процесс останавливается на 10 секунд, а дочерний процесс завершается немедленно вызовом `exit(0)`. Поскольку родительский процесс не предпринимает никаких действий, в системе существуют оба процесса, причем дочерний — в состоянии зомби. После завершения родительского процесса автоматически снимается и дочерний процесс.

Системная функция `exec()`[1] дает возможность процессу запускать другую программу, при этом соответствующий этой программе исполняемый файл будет загружен в пространство памяти данного процесса. Содержимое пользовательского контекста после вызова функции становится недоступным, за исключением передаваемых функции параметров, которые переписываются ядром из старого адресного пространства в новое. Распространенный вариант вызова функции имеет вид:

```
execvp(filename,argv,envp);
```

где `filename` — имя исполняемого файла, `argv` — указатель на массив параметров, которые передаются вызываемой программе, а `envp` — указатель на массив параметров, составляющих среду выполнения вызываемой программы. С помощью системных вызовов семейства exec() в дочернем процессе запускаются на выполнение все системные и пользовательские программы.

На рис. 11 показана схема создания нового процесса в ОС UNIX посредством вызовов `fork()` и `exec()` с применением синхронизации в родительском процессе через вызов `wait()`.

[1] В действительности существует семейство функций с общим названием `exec`, см. описание системной библиотеки `libc`.

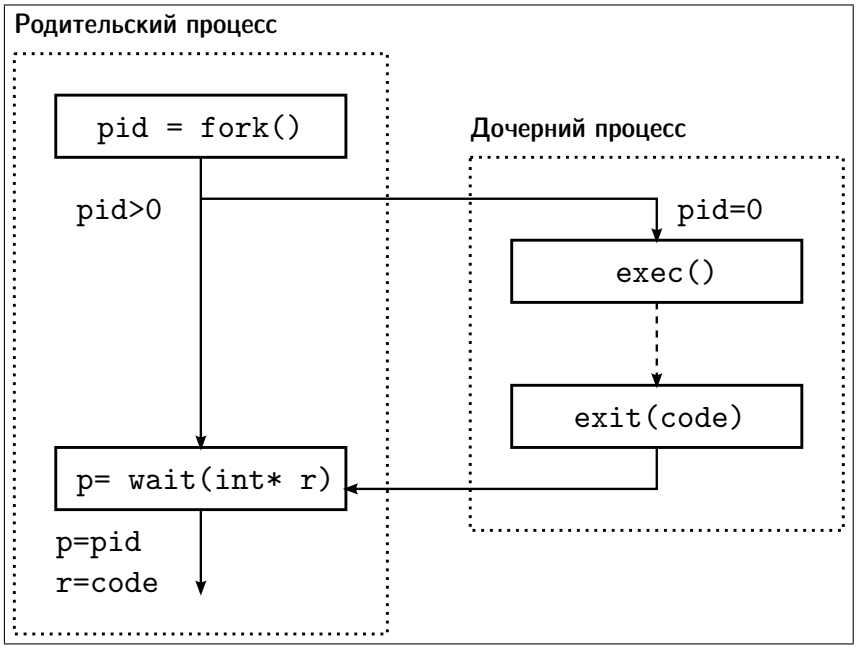

Рис. 11. Создание нового процесса в ОС UNIX.

7.4. Переменные окружения

Каждый запускаемый в ОС процесс содержит некое информационное пространство, именуемое *окружением*, в котором можно задавать именованные хранилища данных — *переменные окружения* (другой термин — *переменные среды*). Для переменных окружения определены операции *записи* любой текстовой информации (присвоение значения переменной окружения) и *чтения* этой информации.

При создании нового процесса окружение процесса-потомка создается как точная копия окружения процесса-родителя. Переменные окружения дочернего процесса копируются из родительского, поэтому дочерний процесс не может непосредственно повлиять на переменные окружения родительского процесса. Родительский процесс может заранее установить список или модифицировать значения наследуемых переменных для дочернего процесса.

Переменные окружения устанавливаются пользователем или сценариями оболочки. Начальный набор переменных инициализируется стартовыми сценариями ОС и сценариями, запускаемыми при регистрации пользователя в системе (см. гл. 10). Переменные окружения имеют большое значение в ОС UNIX, так как хранят множество настроек как системы в целом, так и отдельных программ.

Получить информацию о переменных окружения можно с помощью коман-

ды `env`, либо вывести значение определенной переменной через команду

```
echo $VARNAME
```

где `VARNAME` — имя переменной. Установить значение переменной для текущего сеанса командной оболочки можно с помощью команды присвоения

```
VARNAME=VALUE
```

где `VALUE` — значения переменной.

Если необходимо сделать переменную доступной для всех процессов, запускаемых из данной оболочки, нужно выполнить конструкцию

```
export VARNAME="VALUE"
```

В библиотеке Си значение переменной окружения можно получить с помощью функции `getenv()`, а установить — функцией `setenv()`. Удаляется переменная из окружения процесса функцией `unsetenv()`. Наиболее важные и часто используемые переменные окружения в ОС UNIX приведены в табл. 11.

Таблица 11. Основные переменные окружения процесса в ОС UNIX.

Переменная	*Пример значения*	*Описание*
`DISPLAY`	`:0.0`	Переменная используется графической подсистемой X11 и указывает на адрес X-сервера и номер используемого экрана (см. гл. 9)
`HOME`	`/home/user`	Переменная содержит имя домашнего каталога пользователя
`LANG` `LC_...`	`ru_RU.UTF-8`	Переменные, задающие язык интерфейса и другие параметры локализации программы
`PATH`	`/bin:/usr/bin:` `/usr/local/bin`	Содержит список каталогов (разделитель «:»), в которых производится поиск исполняемых файлов с соответствующим команде именем
`PWD`	`/tmp`	Текущий каталог процесса (можно узнать командой `pwd`)
`SHELL`	`/bin/bash`	Имя текущей программы оболочки
`TERM`	`xterm`	Тип терминала текущего процесса
`USER`	`user`	Имя текущего пользователя

7.5. Типы процессов

Процессы в ОС UNIX возможно классифицировать по различным признакам. С точки зрения спектра основных выполняемых задач и уровня привилегий процессы разделяются на *системные* и *пользовательские*.

Системные процессы выполняются в режиме суперпользователя и ориентированы на выполнение системных функций. Среди таких процессов выделяют особую разновидность — *процессы-демоны* (от англ. daemon[1]). Процессы-демоны обычно запускаются во время загрузки системы (см. гл. 10) и имеют следующие типичные задачи: серверы сетевых протоколов (например, HTTP, FTP, электронная почта и др.), сервер базы данных, управление оборудованием, поддержка очередей печати для принтера, управление выполнением заданий по расписанию и т. д.

Процессы-демоны имеют следующие особенности:

1) не связаны ни с одним из пользователей и не имеют ассоциированного с ними терминала;
2) не входят в группу процессов, а соответственно, не являются членами сеансов (см. п. 10.3);
3) игнорируют сигналы `SIGINT`, `SIGHUP` и `SIGQUIT` (см. п. 8.2).

Бóльшую часть времени демоны ожидают, пока тот или иной процесс запросит определенную услугу через установленные средства межпроцессных взаимодействий (см. гл. 8). Пользователь (системный администратор) может получить информацию о функционировании такого процесса из системных журналов — *логов* (от англ. *log* — журнал регистраций).

К *пользовательским процессам* относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы. Важнейшим пользовательским процессом является начальный командный интерпретатор, который обеспечивает выполнение команд пользователя в системе UNIX.

Пользовательские процессы могут выполняться как в *интерактивном (приоритетном)*, так и в *фоновом* режимах. Интерактивные процессы мо-

[1]Иногда слово *daemon* интерпретируют как акроним англ. *Disk and execution monitor*, но само слово возникло на заре ОС UNIX гораздо раньше этой расшифровки.

нопольно владеют терминалом, и пока такой процесс не завершит свое выполнение, пользователь не имеет доступа к командной строке.

В современных версиях ОС UNIX помимо процессов существует понятие *программного потока* или *нити* (англ. *thread*)[1]. В рамках процесса может существовать несколько потоков, каждый из которых исполняется независимо, но все они объединены общим виртуальным адресным пространством. Можно сказать, что все процессы исполняются с единственным потоком по умолчанию, но при необходимости могут быть созданы новые потоки.

Потоки отсутствовали в оригинальной архитектуре ОС UNIX и были добавлены под влиянием современных архитектур персональных компьютеров, в которых переключение контекста исполнения между процессами занимает значительно большее время, чем переключение контекста исполнения между потоками. Однако ценой эффективного переключения между потоками является сильное влияние потоков в рамках одного процесса друг на друга: критическая ошибка в одном потоке приводит к аварийному завершению всего процесса [5].

7.6. Приоритет процессов

Процессы в ОС UNIX имеют различный приоритет выполнения, определяемый в каждый момент планировщиком процессов [3, 29, 30]. Так, системные процессы, как правило, имеют изначально более высокий приоритет по сравнению с пользовательскими. В ОС UNIX традиционно используются динамически изменяющиеся приоритеты. При образовании процесса ему назначается некоторый устанавливаемый системой *статический* или *относительный приоритет* — параметр *nice number (NI)*, учитываемый планировщиком процессов при определении очередности их запуска.

Реальным критерием планирования выступает *динамический приоритет* или *приоритет выполнения* — параметр *priority* (PRI), а статический приоритет составляет основу начального значения динамического приоритета процесса и не изменяется ядром на всем протяжении жизни процесса.

Диапазон значений приоритета различен, в зависимости от версии ОС UNIX и используемого планировщика. В ОС Linux и Solaris параметр

[1] Существует также термин *Light-Weight Process* — дословно, «легковесный» процесс.

nice number может принимать значения от -20 до 20, при этом наивысшему приоритету соответствует минимальное значение *nice number*, низшему — максимальное значение *nice number*. Пользовательские процессы (выполняемые от имен обычного пользователя) могут принимать значения *nice number* от 0 до 20, системные процессы (выполняемые от имени суперпользователя root) обычно запускаются с бóльшим приоритетом (с отрицательным значением *nice number*). Диапазон значений приоритета выполнения PRI может отличаться от диапазона *nice number* (например, в ОС Solaris диапазон PRI составляет от 0 до 159).

Отдельно стоит выделить так называемый *приоритет реального времени*, который получают наиболее важные системные процессы, критичные к каким-либо временны́м задержкам. Такой приоритет обозначается как RT (от англ. Real Time).

Для изменения стартового значения относительного приоритета процесса при запуске программы используется команда `nice`, а для изменения относительного приоритета выполняемого процесса — команда `renice`. В приложениях приоритет может быть изменен системным вызовом `nice()`.

8. Средства межпроцессного взаимодействия

8.1. Обзор средств взаимодействия процессов

Возможности многозадачной ОС были бы сильно ограничены, если бы в ней не имелись развитые средства взаимодействия и обмена данными между выполняемыми процессами. Поэтому одна из важнейших задач ОС UNIX — обеспечивать контролируемые взаимодействия процессов, в том числе за счет возможности разделения одного сегмента памяти между виртуальной памятью нескольких процессов. Взаимодействие между процессами необходимо для решения следующих задач:

- *Передача данных.* Один процесс передает данные другому процессу, при этом их объем может варьироваться от десятков байт до сотен мегабайт.
- *Совместное использование данных.* Вместо копирования информации от одного процесса к другому, процессы могут совместно использовать одну копию данных, причем изменения, сделанные одним процессом, будут сразу же заметны для другого. Количество взаимодействующих процессов может быть больше двух.
- *Извещение.* Процесс может известить другой процесс или группу процессов о наступлении некоторого события. Это может понадобиться, например, для синхронизации выполнения нескольких процессов.
- *Управление.* Один процесс может послать какие-либо сигналы управления другому процессу, например, для приостановления или завершения последнего.

Все средства межпроцессного взаимодействия в литературе, как правило, обозначаются собирательным термином *IPC* (от англ. Inter-Process Communication). Перечислим основные средства IPC в ОС UNIX и их функции:

1) *сигналы* (извещение, управление),

2) *каналы* (передача данных),

3) *именованные каналы FIFO* (передача данных),

4) *сокеты* (передача данных) (включая сетевые соединения и сокеты BSD UNIX),

5) *семафоры* (извещение, совместное использование данных),

6) *очереди сообщений* (извещение, передача данных),

7) *разделяемая память* (совместное использование данных).

Далее будут рассмотрены основные средства межпроцессного взаимодействия в ОС UNIX.

8.2. Механизм сигналов

Сигналы — одно из традиционных средств межпроцессного взаимодействия в ОС UNIX. Сигнал может быть отправлен процессу операционной системой или другим процессом. ОС использует сигналы для доставки процессу уведомлений об ошибках или неправильном поведении.

При получении сигнала исполнение процесса приостанавливается и запускается специальная подпрограмма — *обработчик сигнала*. Обработчики сигналов могут быть явно определены в исполняемой программе. Если же они отсутствуют, а также в некоторых специальных случаях используется *стандартный обработчик*, определенный ОС.

У сигнала есть только одна характеристика, несущая информацию — его номер (целое положительное число). Иначе говоря, сигналы — это заранее определенный и пронумерованный список сообщений. Для удобства использования каждый сигнал имеет сокращенное буквенное имя. Список сигналов и их имен стандартизован и практически не отличается в различных версиях ОС UNIX. Перечень основных сигналов, традиционно присутствующих в ОС UNIX, с указанием их номеров в ОС Solaris и Linux приведен в табл. 12.

Рассмотрим подробнее назначение основных сигналов [3, 5].

`SIGABRT`

Сигнал, посылаемый процессом самому себе при выполнении вызова `abort()`, для аварийного останова в случае невозможности дальнейшего продолжения программы. Обработчик по умолчанию завершает процесс.

`SIGALRM`

Процесс может с помощью специального системного вызова `alarm()`

Таблица 12. Перечень основных сигналов в ОС Solaris и Linux

Номер в Solaris/ Linux	Название	Описание	Тип*
6/6	SIGABRT	Сигнал, посылаемый системным вызовом abort()	упр.
14/14	SIGALRM	Сигнал истечения времени, заданного функцией alarm()	увед.
10/7	SIGBUS	Неправильное обращение в физическую память	искл.
18/17	SIGCHLD	Дочерний процесс завершен или остановлен	увед.
25/18	SIGCONT	Продолжить выполнение ранее остановленного процесса	упр.
8/8	SIGFPE	Ошибочная арифметическая операция	искл.
1/1	SIGHUP	Закрытие терминала	увед.
4/4	SIGILL	Недопустимая инструкция процессора	искл.
2/2	SIGINT	Сигнал прерывания (Ctrl-C) с терминала	упр.
9/9	SIGKILL	Безусловное завершение	упр.
13/13	SIGPIPE	Запись в разорванное соединение (пайп, сокет)	увед.
3/3	SIGQUIT	Сигнал «Quit» с терминала	упр.
11/11	SIGSEGV	Нарушение при обращении в память	искл.
23/19	SIGSTOP	Остановка выполнения процесса	упр.
12/31	SIGSYS	Неправильный системный вызов	искл.
15/15	SIGTERM	Сигнал завершения процесса	упр.

* Типы сигналов: упр. — управление, увед. — уведомление, искл. — исключение.

задать время, через которое ему необходимо отправить сигнал. Через указанный промежуток времени ОС доставит процессу уведомляющий сигнал SIGALRM. Обычно этот прием применяется для задания таймаутов (англ. timeout). Если процесс не разрегистрировал обработчик этого сигнала, то обработчик по умолчанию завершает процесс.

SIGBUS

Сигнал отправляется в случае какой-либо аппаратной ошибки, например, при попытке обращения к виртуальному адресу, для которого отсутствует соответствующая физическая страница. Обработчик по умолчанию завершает процесс.

SIGCHLD

Уведомляющий сигнал отправляется родительскому процессу в случае завершения его дочернего процесса. По умолчанию сигнал игнорируется.

SIGCONT

Управляющий сигнал продолжения исполнения программы после остановки (по сигналу `SIGSTOP`). Обработчика по умолчанию нет.

`SIGFPE`

Сигнал исключения, вызванного ошибкой в вычислениях с плавающей точкой. Отправляется ОС при некорректном исполнении программы. Обработчик по умолчанию завершает процесс.

`SIGHUP`

Сигнал уведомления о закрытия терминала, к которому привязан данный процесс. Обычно отправляется операционной системой всем процессам, запущенным из командной строки при завершении сеанса пользователя. Обработчик по умолчанию завершает процесс.

`SIGILL`

Сигнал исключения, вызванного некорректной инструкцией процессора. Отправляется ОС процессу в случае, если в исполнении программы встретилась некорректная инструкция процессора. Обработчик по умолчанию завершает процесс.

`SIGINT`

Сигнал уведомления посылается ядром всем процессам текущей группы при нажатии клавиш прерывания `Ctrl-C`. Обработчик по умолчанию завершает процесс.

`SIGKILL`

Управляющий сигнал аварийного завершения процесса. По этому сигналу процесс завершается немедленно — без освобождения ресурсов. Этот сигнал не может быть перехвачен, заблокирован или переопределен самим процессом, всегда используется стандартный обработчик ОС. Сигнал используется для гарантированного завершения процесса, например, в случае его «зависания».

`SIGPIPE`

Сигнал уведомления отправляется процессу, который пытается отправить данные в канал, закрытый с противоположной стороны (о каналах см. п. 8.4). Такая ситуация может возникнуть в случае, если один из взаимодей-

ствующих процессов был аварийно завершён. Обработчик по умолчанию завершает процесс.

`SIGQUIT`

Сигнал посылается ядром всем процессам текущей группы при нажатии клавиш `Ctrl+\`. Обработчик по умолчанию завершает процесс.

`SIGSEGV`

Сигнал отправляется процессу, если возникло исключение, вызванное неверной операцией с памятью (обращение по несуществующему или защищенному адресу). Обработчик по умолчанию завершает процесс.

`SIGSTOP`

Управляющий сигнал приостановки работы процесса отправляется процессу при нажатии пользователем клавиш `Ctrl-Z`. Этот сигнал не может быть перехвачен, заблокирован или переопределен. Используется для гарантированной приостановки работы процесса с полным сохранением его состояния и возможностью возобновления.

`SIGSYS`

Сигнал посылается процессу при попытке передать неправильный аргумент в системный вызов.

`SIGTERM`

Управляющий сигнал завершения процесса, как правило используемый для корректного завершения его работы.

Для отправки сигналов процессам используется специальный системный вызов `kill()` и одноименная пользовательская команда, при этом необходимо задать номер PID целевого процесса. Команда `pkill` позволяет отправить сигнал процессу, заданному по имени соответствующей программы[1]. По умолчанию, если сигнал не задан, команды `kill` и `pkill` отправляют сигнал `SIGTERM`. В общем случае пользовательскому процессу разрешается посылать сигналы только «своим» процессам, т. е. процессам с таким же UID.

[1] Точнее, выполняется поиск заданной подстроки в списке процессов системы и сигнал отправляется всем найденным процессам.

Процесс может выбрать одно из трех возможных действий при получении сигнала:

1) игнорировать сигнал,

2) перехватить и самостоятельно обработать сигнал,

3) позволить действие по умолчанию.

В первых двух случаях необходимо использовать системный вызов `signal()`[1], где в качестве первого аргумента указывается идентификатор сигнала или его номер, а вторым аргументом — константа `SIG_IGN` в случае (1) или адрес функции-обработчика сигнала в случае (2). Текущее действие при получении сигнала называется *диспозицией сигнала*. Стоит еще раз отметить, что сигналы `SIGKILL` и `SIGSTOP` нельзя игнорировать или перехватить. Более подробно механизм сигналов изложен в [3, 29, 30].

8.3. Стандартные потоки ввода-вывода

Каждый запущенный из командного интерпретатора процесс получает три открытых стандартных потока ввода-вывода: 1) поток ввода данных, 2) поток вывода данных, 3) поток вывода сообщений об ошибках. Потоки имеют стандартные дескрипторы 0, 1 и 2 соответственно. В табл. 13 указаны обозначения стандартных потоков в языках программирования Си и Си++.

Таблица 13. Стандартные потоки ввода-вывода.

Дескриптор	*Описание*	*Си*	*Си++*
0	поток ввода данных	stdin	std::cin
1	поток вывода данных	stdout	std::cout
2	поток вывода сообщений об ошибках	stderr	std::cerr

По умолчанию все эти потоки ассоциированы с терминалом. То есть любая программа, читающая данные из потока `stdin` и выводящая в поток `stdout`, будет ожидать ввода информации с клавиатуры терминала, а выводить информацию, включая сообщения об ошибках, на экран терминала. Бóльшая часть системных и пользовательских утилит ОС UNIX использует только стандартные потоки ввода-вывода. Для таких программ командная

[1] Стандарт POSIX.1 рекомендует вместо функции `signal()` использовать функцию `sigaction()` и новый механизм управления сигналами, основанный на интерфейсе 4.2BSD UNIX [3].

оболочка shell позволяет независимо перенаправлять потоки ввода-вывода. Например, можно отправить вывод одного процесса на ввод другого, подавить вывод сообщений об ошибках или же установить для процесса ввод (вывод) из файла (в файл).

Переназначение ввода из файла в оболочке shell осуществляется добавлением в конец команды символа < и следующим за ним имени файла — источника данных для ввода:

```
команда < файл
```

Перенаправление вывода в файл от команды осуществляется добавлением символа > и следующим за ним имени файла — приемника данных:

```
команда > файл
```

В этом случае файл будет создан, если до этого отсутствовал, или перезаписан заново, если файл уже существовал (старая информация в файле будет уничтожена). Если необходимо добавить данные в конец существующего файла, используется комбинация символов >>:

```
команда >> файл
```

Если при этом файл не существовал, он будет создан автоматически. Все варианты можно комбинировать, например:

```
команда < файл_источник >> файл_приемник
```

В данном примере переназначены и ввод и вывод (в разные файлы).

При перенаправлении потока `stderr` необходимо явно указывать его номер (2) следующим образом:

```
команда 2> файл
```

Возможно объединение потоков `stdout` и `stderr` следующим образом:

```
команда 2>&1
```

Последовательность операций перенаправления потоков данных имеет значение. К примеру, команда

```
команда > файл 2>&1
```

позволяет перенаправить как данные из потока `stdout`, так и данные из потока `stderr` в файл, в то время, как команда

```
команда 2>&1 > файл
```

позволяет перенаправить только данные из потока `stdout` в файл, так как с помощью данной команды осуществляется копирование дескриптора потока `stdout` в дескриптор потока `stderr` перед тем, как поток `stdout` перенаправляется в файл. Далее механизм перенаправления потоков будет рассмотрен подробнее.

8.4. Неименованные каналы

Организовать обмена данными между процессами на уровне перенаправления потоков станет возможным, если, например, один процесс будет записывать данные в файл, а другой — читать из него данные. Но очевидны неудобство и громоздкость такого решения: необходимо создавать файл, т.е. выделять под него место в ФС, а также запускать два процесса отдельно. В таком случае удобнее воспользоваться другим механизмом обмена данными — *конвейерами*[1] на основе *неименованных каналов*.

С точки зрения пользователя оболочки shell, конвейер представляет собой последовательность команд, разделенных символом «|» (вертикальная черта):

```
command1 | command2 | command3 ...
```

Такая конструкция означает, что стандартный поток вывода команды `command1` будет сразу перенаправлен на стандартный поток ввода команды `command2`, а ее поток вывода — на поток ввода команды `command3`. При этом ввод информации для команды `command1` будет осуществляться с клавиатуры, а вывод для команды `command3` — на экран терминала, если после нее не указаны другие команды. Схема организации конвейера для трех процессов показана на рис. 12. Конвейерные конструкции можно объединять с рассмотренным выше перенаправлением в файлы (из файлов).

При вводе конвейерных конструкций реализуется механизм межпроцессного взаимодействия — *неименованный канал*, при этом запускаются параллельно все указанные команды, т. е. в системе порождается столько процессов, сколько команд указано в конвейере (с общим процессом-родителем).

[1] В отечественной литературе можно также встретить термин *канал* или *пайп* от англ. *pipe* или *pipeline* — трубопровод.

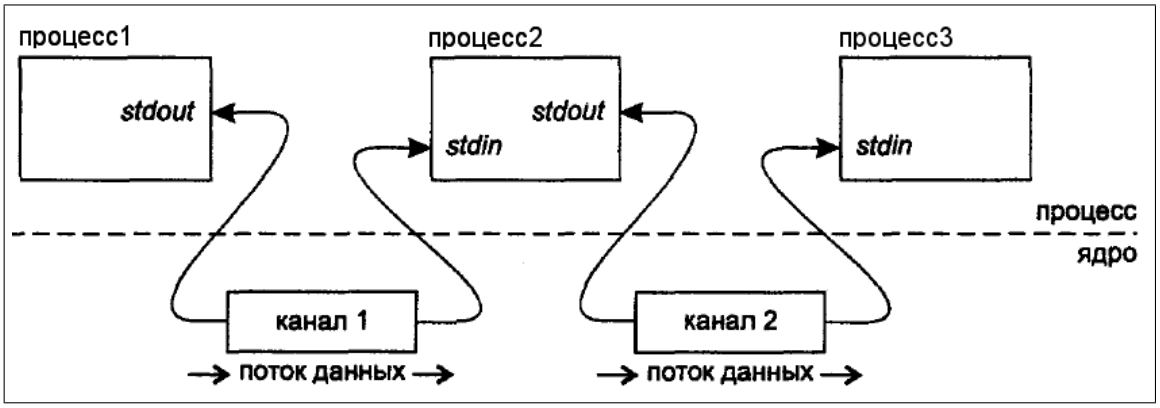

Рис. 12. Каналы между процессами при конвейерной обработке [32].

Неименованный канал обеспечивает *однонаправленную* передачу данных между двумя процессами.

Для создания канала используется системный вызов `pipe()`:

```
int pipe(int *fd);
```

который возвращает два файловых дескриптора — `fd[0]` для чтения из канала и `fd[1]` для записи в канал. После создания канала один процесс может записывает данные в файл с дескриптором `fd[1]`, а другой процесс — читать эти данные из файла с дескриптором `fd[0]`. Доступ к этим дескрипторам кроме самого процесса, создавшего канал, могут получить только дочерние процессы при наследовании контекста родительского процесса. Таким образом, неименованный канал может быть использован для передачи данных только между процессами с общим «родителем».

Рассмотрим подробнее механизм создания неименованного канала на примере программы, запускающей два процесса и создающей между ними канал, причем вывод второго процесса направляется в терминал. Исходный код программы приведен ниже:

```
1  #include <unistd.h>
2
3  int main(int argc, char* argv[]) {
4    int fd[2];
5
6    pipe(fd);
7    if (fork()==0) {
8      close(fd[0]);
9      dup2(fd[1], STDOUT_FILENO);
10     execlp(argv[1],argv[1],NULL);
11   }
12   if (fork()==0) {
13     close(fd[1]);
14     dup2(fd[0], STDIN_FILENO);
15     execlp(argv[2],argv[2],NULL);
16   }
17   return 0;
18 }
```

В строке 1 подключается библиотечный заголовочный файл, в котором определены используемые функции. В строке 4 создается локальный массив `fd` из двух переменных типа `int`. В строке 6 создается неименованный канал и полученные дескрипторы помещаются в массив `fd`. Далее выполняется первый вызов `fork()` (строка 7) и создается первый дочерний процесс с контекстом основного (родительского) процесса.

Управление на строки 8–10 переходит только в дочернем процессе. Здесь закрывается дескриптор для чтения из канала, поскольку первый дочерний процесс использует канал только для записи (строка 8). Затем происходит собственно перенаправление потока `stdout` первого дочернего процесса в неименованный канал[1] (строка 9). В строке 10 происходит замещение контекста первого дочернего процесса и фактически запуск первой программы, указанной в командной строке первым аргументом. Передача дополнительных аргументов в этот процесс в данной программе не предусмотрена.

Далее выполняется второй вызов `fork()` (строка 12) и создается второй дочерний процесс. Управление на строки 13–15 переходит только в дочер-

[1] Макрос `STDOUT_FILENO` определяет дескриптор потока `stdout` и равен значению функции `fileno(stdout)`, т.е. фактически это число 1.

нем процессе. Здесь закрывается дескриптор для записи в канал, поскольку второй дочерний процесс использует канал только для чтения (строка 13). Затем происходит собственно перенаправление потока `stdin` второго дочернего процесса в неименованный канал[1] (строка 14). В строке 15 происходит замещение контекста второго дочернего процесса и запуск второй программы, указанной в командной строке вторым аргументом.

Конвейер из нескольких процессов можно создать и с помощью библиотечной функции `popen()`, указав в качестве строкового аргумента конструкцию из одной или более команд. При этом функция возвращает указатель на обычный файловый поток, но в отличие от регулярного файла, конвейер можно открыть либо только в режиме чтения, либо только в режиме записи. Если конвейер открыт в режиме записи, данные записываются в начало конвейера (в поток `stdin` первого процесса). Если конвейер открыт в режиме чтения, данные считываются из конца конвейера (из потока `stdout` последнего процесса). Поток, созданный функцией `popen()`, закрывается функцией `pclose()`. Таким образом, неименованный канал является временным: когда все процессы заканчивают работу с каналом, ядро ОС его удаляет.

8.5. Именованные каналы

Именованные каналы представляют собой особый тип файлов, которые располагаются в ФС и могут быть открыты любым процессом (если это позволяется правами доступа). Файлы именованных каналов в символьном представлении (например, при выводе командой `ls`) помечаются буквой `p` (от англ. pipe) (см. табл. 8). Одни процессы записывают данные в канал, другие — читают из него, данные продвигаются по каналу в порядке очереди FIFO[2]. Механизм именованных каналов изображен на рис. 13.

С точки зрения как пользователя, так и программиста, работа с именованными каналами ничем не отличается от работы с файлами — канал открывается системным вызовом `open()`, затем обмен данными происходит

[1] Макрос `STDIN_FILENO` определяет дескриптор потока `stdin` и равен значению функции `fileno(stdin)`, т.е. фактически это число 0.

[2] от англ. «First In — First Out» — «первый пришел — первый вышел» — тип очереди, в которой добавление элемента возможно лишь в конец очереди, а выборка — только из начала очереди.

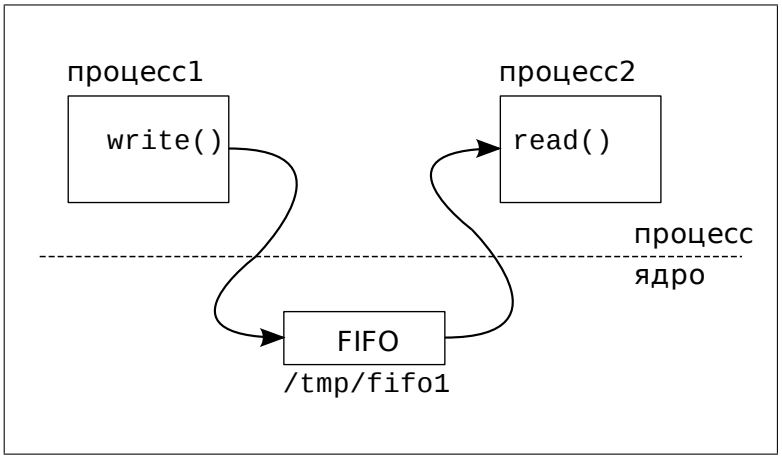

Рис. 13. Именованный канал FIFO между процессами.

посредством вызовов `read()` или `write()`, и закрытие канала для данного процесса выполняет системный вызов `close()`. В оболочке shell при перенаправлении стандартных потоков ввода-вывода именованные каналы используются точно так же, как и обычные файлы. В отличие от неименованных каналов, описанных выше, именованные каналы постоянно присутствуют в иерархии ФС до их явного удаления (как файлов).

Процесс, открывающий именованный канал для чтения, приостановит свое выполнение до тех пор, пока другой процесс не откроет именованный канал для записи, и наоборот. Не имеет смысла открывать канал для чтения, если процесс не надеется получить данные; то же самое касается записи. В зависимости от того, открывает ли процесс именованный канал для записи или для чтения, ядро возобновляет выполнение тех процессов, которые были приостановлены в ожидании записывающего (считывающего) в канал (из канала) процесса [29, 30].

В оболочке shell именованный канал создается командой `mkfifo`, а удаляется стандартной командой удаления файлов `rm`. С точки зрения программиста, именованные каналы создаются с помощью системного вызова `mkfifo()`, а удаляются с помощью системного вызова `unlink()`.

8.6. Сокеты

Общие сведения

Сокеты (от англ. *socket* — гнездо, розетка, место соединения) являются наиболее универсальным и часто применяемым средством IPC, предоставляя интерфейс обмена данными как в пределах одного компьютера, так и меж-

ду процессами, запущенными на разных узлах в компьютерной сети. Интерфейс сокетов традиционно строится на основе сетевой архитектуры *«клиент-сервер»*, выделяя тем самым во взаимодействии двух процессов *клиентскую* и *серверную* части. Серверный процесс создает сокет и переходит в состояние ожидания входящих соединений от других процессов. Само соединение инициирует клиентский процесс, подключаясь к сокету серверного процесса. Существует два принципиально разных типа сокетов: *потоковые сокеты* и *дейтаграммные сокеты*; оба типа сокетов будут рассмотрены далее.

Интерфейс сокетов впервые появился в ОС BSD UNIX и использовался для связи компьютеров через сеть Internet по транспортным протоколам TCP (потоковые сокеты) и UDP (дейтаграммные сокеты). Такое применение сокетов в настоящее время является основным и стандартным средством взаимодействия процессов в сети.

Кроме того, существует локальный вариант взаимодействия через сокеты, в котором обмен данными происходит через специальные файлы, расположенные в файловой системе (фактически, это аналог именованных каналов, но с интерфейсом сокетов). Файлы локальных сокетов, именуемых также *UNIX-сокетами*, в символьном представлении (например, при выводе командой `ls`) помечаются буквой `s` (см. табл. 8).

Сокеты с общими коммуникационными свойствами, такими как способ именования и протокольный формат адреса, группируются в *домены*. Наиболее часто используются «домен системы UNIX» для локальных процессов в пределах одного компьютера и «домен Internet» для соединения процессов, выполняемых на разных компьютерах в сети на основе тех или иных протоколов сетевого и транспортного уровней, например, IP и TCP.

Сокет создается с помощью системного вызова `socket()`:

```
sd = socket(domain,type,protocol);
```

где

- `domain` — домен коммуникаций: для локального сокета указывается макрос `PF_UNIX`, для сетевых соединений на основе протокола IPv4 указывается макрос `PF_INET`, для протокола IPv6 — макрос `PF_INET6`, кроме того, возможно указание и других сетевых протоколов, специфичных для конкретной реализации ОС;

- `type` — тип сокета: для потокового соединения указывается макрос `SOCK_STREAM`, для дейтаграммного — макрос `SOCK_DGRAM`, кроме того, в зависимости от реализации ОС, возможно указать и другие типы сокета, например `SOCK_RAW` для низкоуровнего формирования сетевых пакетов;
- `protocol` определяет протокол коммуникаций, который, как правило, уже неявно задан типом сокета.

Дескриптор сокета `sd`, возвращаемый функцией `socket()`, используется другими системными функциями для дальнейшего обращения к сокету. Закрытие сокета выполняет системный вызов `close()`.

Системный вызов `bind()` связывает дескриптор сокета с локальным адресом, тип которого зависит от домена коммуникаций. Так, для обычного локального UNIX-сокета (домен `PF_UNIX`) адрес будет задавать имя файла. В случае же сетевого взаимодействия адрес нужно задавать специальной структурой типа `sockaddr`, включающей сетевой адрес узла[1] и номер UDP или TCP порта.

Потоковые сокеты

Потоковые сокеты используются при необходимости создания надежного соединения для непрерывной последовательной передачи данных и имеют следующие особенности:

- обмен данными осуществляется в потоковом режиме аналогично режиму работы с регулярными файлами и символьными устройствами посредством системных вызовов `write()` и `read()`;
- обеспечивается *буферизация* передаваемых и принимаемых данных *в режиме FIFO*,
- обеспечивается полностью дуплексный (двунаправленный) канал обмена данными;
- *гарантируется доставка данных* за конечное время (для сетевого применения — при исправности физического соединения) с обеспечением

[1] В данном случае сетевой адрес необходимо для привязки сокета к определенному сетевому интерфейсу.

целостности информации, т.е. без потерь, без повторений и в оригинальной последовательности;

- перед обменом данными оба процесса должны провести процедуру *установления соединения*, а после завершения обмена — процедуру *закрытия соединения*.

Взаимодействие процессов через потоковые сокеты иллюстрирует рис. 14. Перед началом работы *процесс-сервер* должен создать специальный сокет для входящих соединений посредством системного вызова `socket()`, в котором указывается его тип и область действия. В случае успешного создания сокета вызов возвращает *дескриптор сокета* — целочисленный идентификатор, уникальный для данного процесса (аналогично файловому дескриптору). Созданный сокет необходимо связать с каким-либо локальным адресом при помощи системного вызова `bind()`.

Далее необходимо установить данный сокет в режим приёма входящих соединений посредством системного вызова `listen()`. После этого процесс-сервер может непрерывно запрашивать систему о наличии входящего соединения через системный вызов `accept()`. В случае получения входящего запроса на соединение со стороны процесса-клиента вызов `accept()` возвращает всю доступную информацию о сокете последнего, а также дескриптор нового сокета, созданного системой для непосредственного обмена данными. Если процесс-сервер решает принять соединение, то он начинает обслуживание данного соединения и обмен данными с процессом-клиентом через полученный новый сокет. В случае принятия решения об отказе обслуживания данного клиента сокет может быть сразу закрыт.

Алгоритм работы *процесса-клиента* несколько проще. Процесс-клиент также создает сокет посредством системного вызова `socket()` и связывает его с локальным адресом вызовом `bind()`. Далее он делает попытку подключиться к сокету процесса-сервера посредством системного вызова `connect()`. В случае успешного соединения оба процесса могут сразу начать обмен данными. Со стороны процесса-клиента используется первоначально созданный сокет. После использования потоковые сокеты должны быть закрыты со стороны клиента или сервера посредством системного вызова `close()`.

Дейтаграммные сокеты

Дейтаграммные сокеты используются в случае отсутствия необходимости установления постоянного соединения и предназначены для обмена информацией посредством небольших блоков данных — сообщений, длина которых, как правило, не превышает 64 кбайт. Такие сообщения принято называть *дейтаграммами* (англ. datagram). Дейтаграммные сокеты имеют следующие особенности:

- обмен данными осуществляется посредством передачи дейтаграмм через системные вызовы `sendto()` или `sendmsg()` и приёма дейтаграмм через системные вызовы `recvfrom()` или `recvmsg()`;
- обеспечивается *буферизация* передаваемых и принимаемых данных *на уровне целых дейтаграмм*, т.е. за раз можно передать или принять только всю дейтаграмму целиком;
- не гарантируется: 1) доставка всех отправленных дейтаграмм, 2) отсутствие повторения дейтаграмм, 3) доставка дейтаграмм в оригинальной последовательности;
- отсутствует необходимость установления соединения перед отправкой дейтаграмм.

Дейтаграммный метод используется при передаче либо очень коротких сообщений, либо при передаче информации в режиме вещания, например, для сетевого радио- или видеовещания, когда пропадание небольших фрагментов информации не является критическим. Кроме того, дейтаграммный метод не

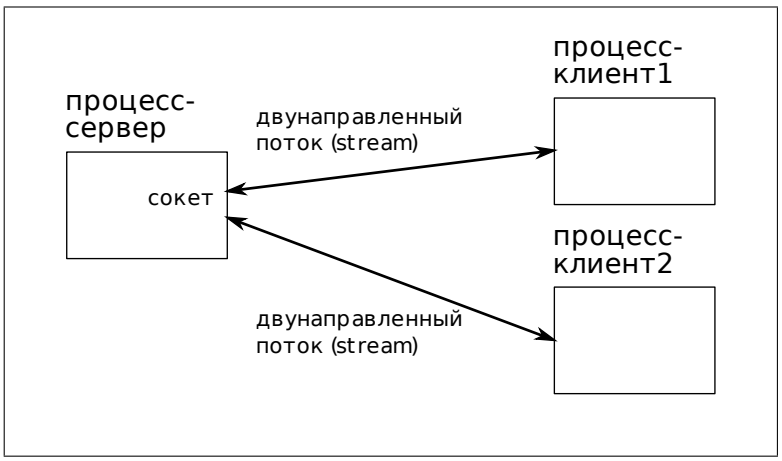

Рис. 14. Взаимодействие процессов посредством потоковых соединений.

требует временны́х затрат на установления постоянного соединения и не создает некоторой избыточности в трафике, присущей потоковым соединениям.

Взаимодействие процессов через дейтаграммные сокеты иллюстрирует рис. 15. Как и в случае потокового сокета, перед началом работы оба взаимодействующих процесса должны создать сокеты посредством системного вызова `socket()` с указанием дейтаграммного типа. Кроме того, необходимо связать каждый сокет с каким-либо локальным адресом при помощи системного вызова `bind()`.

Поскольку при дейтаграммных взаимодействиях соединение как таковое не создается, далее процесс-клиент может сразу же отправлять дейтаграммы на сокет процесса-сервера посредством системного вызова `sendto()` с указанием адреса. Процесс-сервер периодически может обращаться к системному вызову `recvfrom()` для определения наличия очередной принятой дейтаграммы.

Как видно, в данном случае роль «клиента» или «сервера» назначается процессу достаточно условно. Клиентом можно считать процесс, который первым отправит дейтаграмму, а дальнейшее поведение процессов, в том числе и необходимость ответа, определяется уже прикладной задачей.

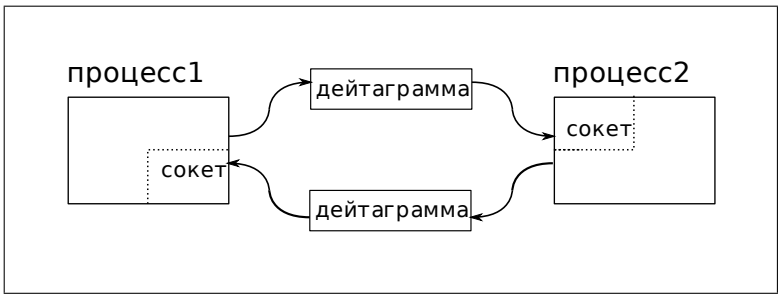

Рис. 15. Взаимодействие процессов посредством дейтаграмм.

8.7. Семафоры

Для синхронизации доступа нескольких процессов к разделяемым ресурсам используется средство IPC *семафоры*. Семафоры выполняют функцию извещения одного процесса о состоянии другого процесса или ресурса, с которым он работает, путем изменения своего *целочисленного значения*. В принципе, возможна различная реализация семафоров. В ОС UNIX для семафоров определены следующие операции [1, 25, 33]:

- $A(S, n)$ — увеличить значение семафора S на величину n;

- $D(S, n)$ — ожидание значения семафора $S \geq n$ (процесс блокируется до тех пор, пока $S < n$), далее $S = S - n$;
- $Z(S)$ — ожидание нулевого значения семафора (процесс блокируется до тех пор, пока $S \neq 0$).

Изначально семафоры инициализируются нулевым значением.

Допустим, имеется некий разделяемый ресурс, для которого необходимо исключить работу с более чем одним процессом одновременно. В этом случае можно создать семафор S, у которого значение «1» сигнализирует о доступности ресурса, «0» — о недоступности. После инициализации ресурса семафор операцией $A(S, 1)$ устанавливается в состояние «1». Затем перед обращением к ресурсу каждый процесс должен выполнить операцию $D(S, 1)$: если значение S больше нуля, выполняется программный код процесса по работе с ресурсом, иначе процесс попадает в состояние ожидания освобождения ресурса (единичного состояния семафора). После завершения работы с ресурсом процесс обязан выполнить операцию $A(S, 1)$, чтобы просигнализировать другим процессам об освобождении ресурса.

В данном примере семафор принимает только два значения, характеризующих состояния «свободен» и «занят». Такие семафоры называются *двоичными*, или *мьютексами* (англ. mutex — сокращение от Mutual Exclusion). Во многих ОС мьютексы также реализованы как средство синхронизации выполнения нескольких вычислительных потоков (тредов) в пределах одного процесса (см. п. 7.5).

Значение семафора должно быть доступно более чем одному процессу, соответственно оно должно храниться в пространстве оперативной памяти ядра. Кроме того, любая операция с семафором по определению должна быть *атомарной*, т.е. неделимой с точки зрения любого процесса. Только в этом случае процесс, изменяющий значение семафора, гарантирован от того, что в этот же момент времени это не попытается сделать другой процесс. Таким образом, операции с семафором могут быть реализованы только в пространстве ядра через системные вызовы.

Семафор в ОС UNIX состоит из следующих элементов:

1) значение семафора;

2) число процессов, ожидающих нулевого значения семафора;

3) число процессов, ожидающих увеличения значения семафора;

4) идентификатор процесса (PID), который хронологически последним работал с семафором.

Для работы с семафорами существуют три системных вызова:

- `semget()` для создания и получения доступа к набору семафоров;
- `semop()` для выполнения основных операций над семафорами A, D и Z;
- `semctl()` для выполнения разнообразных управляющих операций над набором семафоров.

8.8. Очереди сообщений

Очереди сообщений предназначены для асинхронной передачи сообщений между процессами. Обмен сообщениями проходит следующим образом: один процесс помещает сообщения в очередь, а другой процесс может прочитать их из указанной очереди, при этом возможно задать порядок и критерии выборки сообщений. Процесс-источник и процесс-приемник должны использовать один и тот же идентификатор очереди. Очереди сообщений располагаются в адресном пространстве ядра ОС в виде *однонаправленных списков* и имеют ограничение по объему информации, хранящейся в каждой очереди. Каждый элемент списка представляет собой отдельное сообщение.

Каждая очередь сообщений имеет свой уникальный идентификатор. Процессы могут записывать и считывать сообщения из различных очередей. Процесс, пославший сообщение в очередь, может не ожидать чтения этого сообщения каким-либо другим процессом. Он может закончить свое выполнение, оставив в очереди сообщение, которое будет прочитано другим процессом позже. Данная возможность позволяет процессам обмениваться структурированными данными, имеющими следующие атрибуты:

1) тип сообщения (позволяет мультиплексировать сообщения в одной очереди),

2) длина данных сообщения в байтах (может быть нулевой),

3) собственно данные (если длина ненулевая, могут быть структурированными).

Выборка сообщений из очереди может осуществляться тремя способами:

1) в порядке FIFO, независимо от типа сообщения;

2) в порядке FIFO для сообщений конкретного типа;

3) первым выбирается сообщение с минимальным типом, не превышающим некоторого заданного значения, пришедшее раньше других сообщений с тем же типом.

Механизм очередей сообщений обеспечивается следующими системными вызовами:

- `msgget()` для образования новой очереди сообщений или получения дескриптора существующей очереди,
- `msgsnd()` для постановки сообщения в указанную очередь,
- `msgrcv()` для выборки сообщения из очереди,
- `msgctl()` для выполнения ряда управляющих действий.

Более подробно с механизмом очередей сообщений можно ознакомиться в [3, 29–32].

8.9. Разделяемая память

Интенсивный обмен данными между процессами с использованием рассмотренных механизмов IPC может вызвать падение производительности системы, поскольку в обмене постоянно участвует ядро, тратится время на системные вызовы, тратится время и оперативная память на буферизацию потоков данных и т.д. В ОС UNIX процессы могут взаимодействовать друг с другом непосредственно путем разделения (совместного использования) участков виртуального адресного пространства и обмена данными через *разделяемую память*. После присоединения к виртуальному адресному пространству процесса область разделяемой памяти становится доступна так же, как и любой участок виртуальной памяти. Для обращения в эту область памяти не нужно использовать какие-либо системные вызовы — процессы используют обычные машинные команды для доступа в ОЗУ, что избавляет от накладных расходов на передачу данных через ядро. На рис. 16 показана схема организации разделяемой памяти между двумя процессами.

Приведем основные системные вызовы для работы с разделяемой памятью:

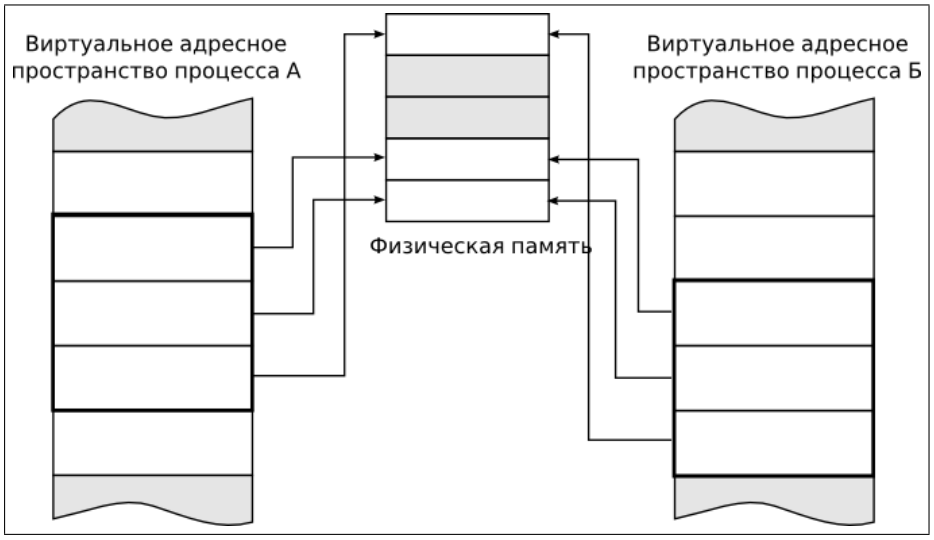

Рис. 16. Организация разделяемой памяти между двумя процессами [5].

- `shmget()` — создает новую область разделяемой памяти или возвращает адрес уже существующей области,
- `shmat()` — логически присоединяет область к виртуальному адресному пространству процесса и возвращает адрес начала области,
- `shmdt()` — отсоединяет область,
- `shmctl()` — осуществляет операции управления разделяемой областью памяти.

При использовании разделяемой памяти взаимодействующие процессы должны обеспечить механизм синхронизации доступа: один процесс перед чтением данных должен быть проинформирован о том, что другой процесс завершил их модификацию. Это легко достигается применением семафоров. Особенности организации и применения механизма разделяемой памяти детально рассмотрены в [3, 25, 29–32].

8.10. Идентификация средств IPC

Рассмотренные средства взаимодействия процессов имеют различные *пространства имен*, в которых они идентифицируется с точки зрения других процессов во всей вычислительной системе. Также средства IPC имеют различные способы *идентификации* с точки зрения API, т.е. в пределах одного процесса (программы). В табл. 14 приведены указанные характеристики рассмотренных средств IPC.

Семафоры, очереди сообщений и разделяемая память имеют общий принцип организации API: их идентификация в системе осуществляется с помощью так называемого *ключа*, представляющего собой целое число (библиотечный тип `key_t`). Функция `ftok()` стандартной библиотеки позволяет сгенерировать уникальный ключ для дальнейшей работы с данными видами средств IPC на основании двух аргументов — некоторой строки и целого числа. В качестве строки обычно выбирается имя какого-либо существующего файла. Важно, чтобы все взаимодействующие с данным средством IPC процессы одинаковым образом сформировали ключ (указали одни и те же аргументы в функции `ftok()`. Для каждого типа IPC пространство значений уникальных ключей независимо. После генерации ключа возможно уже обращение к системным вызовам с общим суффиксом `*get()` (см. выше) для получения идентификатора средства IPC, используемого в дальнейшей работе.

Таблица 14. Сводная таблица средств IPC в ОС UNIX

Средство IPC	*Пространство имен в ОС*	*Дескриптор в программе*
Неименованный канал	—	Дескриптор
Именованный канал FIFO	Имя файла в ФС	Дескриптор
Сокет локальный	Имя файла в ФС	Дескриптор
Сокет сетевой (IP)	IP адрес и порт	Дескриптор
Семафор	Ключ	Идентификатор
Очередь сообщений	Ключ	Идентификатор
Разделяемая память	Ключ	Идентификатор

9. Интерфейс пользователя

9.1. Алфавитно-цифровые терминалы

К началу 1970-х гг. *текстовый* или *алфавитно-цифровой терминал* оставался основным средством общения человека с ЭВМ. ОС UNIX, как и другие ОС того времени, использовала текстовые терминалы для организации интерфейса *командной строки*. Суть интерфейса состоит в обмене текстовыми сообщениями: пользователь набирает команды, а ОС возвращает ему результат в текстовом виде. Программную часть интерфейса со стороны ОС при этом реализует та или иная командная оболочка. Командная строка остается основным интерфейсом и сегодня, несмотря на то, что в современных UNIX-системах широко используется также и графический интерфейс.

Прототипом терминала был *телетайп*[1], который использовался в качестве средства общения с ЭВМ еще в 1950-е гг. Ввод информации в ЭВМ осуществлялся посредством клавиатуры телетайпа, выводилась же информация на бумажную ленту печатным способом. Впоследствии стали выпускаться терминалы с мониторами на основе электронно-лучевых трубок. Классическим примером является терминал VT100 производства компании DEC, ставший фактически индустриальным стандартом тех лет (см. рис. 17).

Рис. 17. Алфавитно-цифровой терминал VT100 производства компании DEC, 1978 г.

[1]Телетайп (англ. teletype, TTY) — электромеханическая печатная машина, используемая для передачи текстовых сообщений между двумя абонентами по простейшему электрическому каналу.

В современных компьютерах для организации интерфейса командной строки используется клавиатура и монитор, работающий в текстовом (алфавитно-цифровом) режиме. Терминалом может являться и отдельная программа, эмулирующая его функции в оконной графической среде. Для ОС UNIX широко известны эмуляторы терминалов `xterm` и `rxvt`. Роль терминала может играть программа, выполняющаяся на удаленном компьютере, связанным с данной UNIX-машиной посредством компьютерной сети. Сегодня такой способ удаленного доступа очень широко практикуется при администрировании серверов, физический доступ к которым затруднен или просто невозможен.

Можно также отметить еще один тип терминалов — *виртуальные консоли*, представляющие собой несколько параллельно выполняемых ОС программ (обычно именуемых `mingetty`), предоставляющих пользователю возможность зарегистрироваться в системе в текстовом режиме и получить доступ к командной строке. Например, в ОС Linux обычно создается шесть таких консолей, переключение между которыми в текстовом режиме производится с помощью комбинаций клавиш `Alt-F1`, `Alt-F2` и т.д.

Терминалы должны предоставлять возможность отображать и передавать не только алфавитно-цифровые символы, но также и *управляющие символы*, кодирующие операции редактирования текста, перемещения курсора, передачи введенного текста системе, визуального выделения отображаемого текста и т.д. Часть управляющих символов вводится специальными клавишами, предусмотренными для управления вводом текста, такими как `Backspace`, `Delete` и `Enter`. Управляющих символов может быть больше, чем управляющих клавиш на клавиатуре терминала, в этом случае их извлекают с помощью какого-либо клавиатурного модификатора, например, клавиши `Ctrl` или `Shift`. Так, выше уже были рассмотрены управляющие сочетания клавиш `Ctrl-C` и `Ctrl-Z`, используемые для передачи сигналов процессам.

Каждое устройство (или программа), выполняющее функции терминала, имеет свои возможности по вводу и выводу информации. Примерами таких возможностей являются: число отображаемых цветов, способность перемещать курсор и изменять размер экрана, набор псевдографических и управляющих символов. При этом для ОС важна возможность работать с любыми терминалами одинаковым способом.

В ОС UNIX существует понятие *тип терминала*: каждый тип сводится к набору возможностей, регламентированных в специальном конфигурационном файле. Информация о всех существующих типах терминалов хранится в специализированной базе данных (terminfo или termcap). Примерами типов являются: `tty` (телетайп) или `xterm` (графический терминал).

Все терминалы в системе имеют соответствующие файлы устройств, общим для которых, как правило, будет наличие букв `tty` в имени файла (названии устройства). Аппаратные терминальные устройства, подключаемые через последовательные порты, имеют соответствующие файлы `/dev/ttySN`, где `N` — номер последовательного порта. Терминальные программы в графической системе X Window получают файлы *псевдотерминалов* (англ. pseudo terminal) `/dev/pts/N`, где `N` — номер псевдотерминала. Виртуальные консоли в ОС Linux обозначаются как `/dev/ttyN`, где `N` — номер виртуальной консоли. Имя терминального устройства входит в контекст каждого процесса (атрибут TTY, см. п. 7.2), которое можно узнать, например, с помощью команды оболочки `ps` или переменной окружения `TTY`.

9.2. Командная оболочка

Командная оболочка позволяет управлять системой, создавая при этом минимальный поток информации между терминалом и ОС по сравнению с другими интерфейсами пользователя, например, графической системой X Window (см. п. 9.4). Это немаловажно, например, в случае удаленного доступа к системе посредством медленного канала связи. Командная строка, как основной интерфейс ОС UNIX, позволяет решать большинство пользовательских задач (явно не связанных с графикой) и 100% задач администрирования системы. Команды могут легко объединяться в самые сложные *сценарии*[1] или *командные файлы* — последовательности действий (на основе команд) по автоматическому обслуживанию системы.

Командная оболочка ОС UNIX обрабатывает команды трех типов:

- в качестве имени команды может быть указано имя двоичного исполняемого файла, полученного в результате компиляции исходного текста программы (например, программы на языке Си);

[1] Используется также термин *скрипт* от англ. script — сценарий.

- именем команды может быть имя сценария, содержащего набор командных строк, обрабатываемых какой-либо командной оболочкой;
- команда может быть внутренней командой оболочки.

Наличие внутренних команд для организации условных ветвлений (типа `if` или `case`) и циклов (типа `for`, `while` и `do-while`) делает оболочку незаменимым средством высокоуровнего программирования для быстрого решения повседневных задач по автоматизации обработки информации или администрированию системы.

Пользователям, запускающим команды, нет необходимости знать, какого они типа. Если команда является внутренней, она будет сразу исполнена оболочкой. Иначе будет предпринята попытка поиска соответствующего исполняемого файла (двоичного файла или файла сценария). Внешние команды оболочка ищет как имена файлов, расположенных только в тех системных каталогах ФС, которые перечислены в переменной окружения `PATH` (см. п. 7.4) данного процесса. При этом рабочий каталог процесса (значение переменной окружения `PWD`) не просматривается[1], если это явно не задано в переменной `PATH` как каталог «`.`». Это связано с необходимостью обеспечения общей безопасности системы и гарантирует, что при работе в системе пользователя root от его имени будут запускаться только стандартные команды системы, файлы которых расположены в обозначенных каталогах. Другими словами, злоумышленник не сможет «подложить» суперпользователю в его рабочий каталог на выполнение свой файл, названный так же, как какая-либо часто используемая команда.

Оболочка обычно исполняет команды синхронно, с ожиданием завершения выполнения команды прежде, чем считать следующую командную строку. Тем не менее, допускается и асинхронное исполнение, когда очередная командная строка считывается и исполняется, не дожидаясь завершения выполнения предыдущей команды. В этом случае о командах (процессах) говорят, что они выполняются в *фоновом режиме*. Процессы, запущенные в оболочке, часто именуют *задачами* (англ. jobs). Для управления активными и фоновыми задачами существуют специальные средства:

[1] В отличие от других ОС, таких как DOS и MS Windows, где рабочий каталог всегда просматривается в первую очередь.

- указание символа `&` в конце команды сразу запускает соответствующую задачу в фоновом режиме,
- команда `fg` для запуска задачи (приостановленной или фоновой) в активном режиме,
- команда `bg` для запуска задачи (приостановленной) в фоновом режиме,
- команда `jobs` выводит перечень задач данной командной оболочки,
- команда `kill` для посылки сигналов фоновым и приостановленным задачам данной командной оболочки,
- комбинация клавиш `Ctrl-C` прерывает работу активной задачи (отправляется сигнал `SIGINT`),
- комбинация клавиш `Ctrl-Z` приостанавливает работу активной задачи (отправляется сигнал `SIGSTOP`).

Командная оболочка — первый процесс, который запускает ОС для пользователя при его входе в систему[1]. Как правило, в комплект поставки ОС UNIX входит сразу несколько различных командных оболочек. То, какая именно оболочка будет запускаться для каждого конкретного пользователя, указывается в файле учетных записей `/etc/passwd` (см. п. 6.3).

Любую командную оболочку можно рассматривать как интерпретатор команд (инструкций), реализующий определенный язык программирования, но также способный непосредственно запустить внешнюю программу по имени ее исполняемого файла. Ассортимент разработанных для ОС UNIX оболочек достаточно большой. Отметим самые распространенные из них:

- `sh` — стандартная командная оболочка, называемая также Bourne-Shell; разработана для UNIX System V Стивом Борном; является фактически обязательной в любой POSIX-совместимой реализации ОС UNIX;
- `csh` — оболочка C-Shell, разработанная Биллом Джоем; является основной в BSD UNIX и имеет синтаксис команд, напоминающий язык Си;
- `ksh` — оболочка Korn-Shell; разработана Дэвидом Корном; имеет полную обратную совместимость с Bourne-Shell и включает в себя возможности C-Shell;

[1] Если не используется вход в систему на основе графического интерфейса.

- `bash` — оболочка Bourne-Again Shell — усовершенствованная и модернизированная вариация Bourne-Shell, одна из наиболее популярных современных разновидностей командной оболочки ОС UNIX; является фактически стандартом для ОС Linux;
- `tclsh` — интерпретатор команд языка Tcl, являющийся одним из мощнейших средств высокоуровнего программирования сценариев в ОС UNIX с обширной библиотекой по работе с текстом, файлами, сетевыми сокетами и др.; имеет расширение для создания графического интерфейса пользователя — интерпретатор `wish`; реализован под многие ОС, включая MacOS X, Windows и Android.

Подробнее командные оболочки ОС UNIX и особенности программирования в них рассматриваются в [4–6, 8, 31, 34].

9.3. Удаленный сетевой доступ

Интерфейс командной строки позволяет сравнительно легко и удобно организовать удаленный сетевой доступ к UNIX-машине. Для этого необходимо иметь программу-сервер для обслуживания таких сетевых соединений в ОС UNIX и программу-клиент для подключения к серверу, эмулирующую тот или иной тип терминала. При этом совсем не обязательно, чтобы на компьютере клиента работала именно ОС UNIX. Рассмотрим основные протоколы прикладного уровня, разработанные для удаленного терминального доступа посредством сетевого соединения.

Telnet (от англ. Telecommunication Network) — протокол передачи текстовых данных по сети для эмуляции удаленного терминального доступа к компьютеру. Основная программа Telnet-клиента традиционно называется `telnet`. Система Telnet существует и для ОС, отличных от UNIX, например, MS Windows.

Rlogin (от англ. Remote Login — удаленный вход в систему) — протокол, в целом аналогичный протоколу Telnet; он позволяет пользователям подключаться к UNIX-машине с других компьютеров сети. В протоколе явно регламентируется формат передачи логина и пароля пользователя, а также типа терминала и скорости соединения. Подключение к UNIX-машине по этому протоколу осуществляется командой `rlogin`. В отличие от протокола Telnet, Rlogin обычно используется только в ОС UNIX.

SSH (от англ. Secure Shell — безопасная оболочка) — протокол, позволяющий производить удаленное управление ОС и туннелирование TCP-соединений (например, для передачи файлов). Сходен по функциональности с протоколами Telnet и Rlogin, но, в отличие от них, обеспечивает *шифрование* всего потока данных, включая и передаваемые пароли. SSH допускает выбор различных алгоритмов шифрования, включая RSA, DSA, AES, Blowfish и 3DES. Целостность переданных данных проверяется с помощью алгоритма CRC32 в SSH версии 1 или HMAC-SHA1/HMAC-MD5 в SSH версии 2. SSH-клиенты и SSH-серверы имеются для большинства сетевых ОС. Для подключения по этому протоколу в ОС UNIX используется команда `ssh`, а для MS Windows существует удобная клиентская программа удаленного доступа по всем рассмотренным протоколам — `putty`.

9.4. Графическая система X Window

Интерфейс командной строки не позволяет создавать графические приложения и реализовывать весь потенциал современных компьютеров, поэтому с развитием аппаратного оснащения возникла необходимость создания в ОС UNIX подсистемы графического интерфейса пользователя (англ. Graphical User Interface — GUI). В 1984 г. в MIT была разработана оконная графическая система для ОС UNIX, получившая название *X Window*. В настоящее время проект по разработке реализации системы X Window с открытым исходным кодом координируется организацией X.Org Foundation.

Особенностью системы является клиент-серверная архитектура, где в качестве сервера выступает программно-аппаратная часть, реализующая базовые функции графической среды: прорисовку графических примитивов, перемещение окон на экране монитора, взаимодействие с мышью и клавиатурой и т.п. Клиентом выступает любая прикладная программа, соединяющаяся с *X-сервером* и отправляющая команды на формирование тех или иных графических объектов, а также команды на ввод данных с клавиатуры или мыши. Таким образом, любая программа, рассчитанная на графический интерфейс с пользователем, будет являться клиентом в системе X Window.

Система X Window не определяет деталей интерфейса пользователя — этим занимаются специальные программы — *менеджеры окон*. По этой причине внешний вид программ в среде X Window может очень сильно раз-

личаться в зависимости от возможностей и настроек конкретного оконного менеджера. Оконный менеджер является клиентом системы X Window.

Клиент и сервер в системе X Window могут взаимодействовать как локально, так и с использованием сетевого соединения. Так, возможна ситуация, когда клиентский процесс выполняется в ОС UNIX, а в качестве сервера используется программа *X-сервер* на другом компьютере под управлением другой ОС, например, MS Windows, предоставляя своему пользователю удаленный доступ к UNIX-машине. При этом стоит еще раз отметить, что UNIX-машина с выполняющейся пользовательской программой играет роль клиента по отношению к удаленному компьютеру с запущенным X-сервером, посредством которого пользователь управляет программой.

Обмен данными между клиентом и сервером строится на основе стандартизированного *X-протокола*, независимо от вида соединения — локального или сетевого, чем достигается высокая гибкость и универсальность системы X Window, а также возможность ее работы на различных программно-аппаратных вычислительных платформах.

Чтобы запустить удаленную клиентскую программу, выводящую графику на локальный X-сервер, пользователь обычно открывает эмулятор терминала и подключается к удаленной машине при помощи Telnet или SSH. Затем для данного сеанса определяется переменная `DISPLAY`, указывающая сетевой адрес X-сервера и номер дисплея, на который следует выводить графику:

```
export DISPLAY=[имя компьютера]:0
```

Далее пользователь запускает удаленную клиентскую программу, которая подключается к локальному X-серверу и начинает отображать графику на локальный экран и принимать данные от локальных устройств ввода.

В современных версиях SSH имеется удобный механизм организации таких соединений без необходимости определения переменной `DISPLAY`. Если запустить программу `ssh` с опцией `-X`, то автоматически организуется так называемое *туннелирование* данных X-протокола внутри SSH-соединения, при этом пользователь может сразу запускать X-клиентов, не устанавливая переменную `DISPLAY`. С применением системы SSH увеличивается степень защищенности соединения за счет шифрования потока, поскольку сам X-протокол не предусматривает шифрования передаваемых данных.

9.5. Терминалы типа «тонкий клиент»

В последние годы получили широкое распространение специализированные устройства удаленного доступа — *графические терминалы*, получившие также название *«тонкие клиенты»* (англ. *thin client*). Такие устройства, как правило, представляют собой специализированные бездисковые компьютеры небольшой мощности для сетей с клиент-серверной или терминальной архитектурой, которые переносят собственно вычислительные задачи по обработке информации на сервер. Тонкие клиенты не выполняют программы пользователей и не хранят какие-либо данные: они лишь представляют собой аппаратную реализацию удаленного графического терминала с возможностью подключения разнообразных периферийных устройств от клавиатуры и мыши (минимальный вариант) до внешних носителей с интерфейсом USB, принтеров, сканеров. Часто такие терминалы также способны воспроизводить и оцифровывать звук, поддерживать аппаратные методы аутентификации на основе смарт-карт и пр.

Типичным «тонким клиентом» можно считать *X-терминал* — программно-аппаратное устройство, на котором выполняется программа X-сервер. Эта архитектура завоевала популярность при построении недорогих терминальных парков, в которых множество пользователей одновременно используют один большой сервер приложений. Такое применение системы X Window хорошо соответствует изначальным намерениям разработчиков создать универсальную сетевую систему графического интерфейса ОС UNIX.

X-терминалы могут «изучать сеть» (в пределах локального широковещательного домена) с использованием протокола XDMCP с целью составления списка узлов сети, с которых они могут взаимодействовать (запускать на них клиентские программы). На изначальном узле должна выполняться специальная программа — *дисплейный менеджер X Window* (англ. X Display Manager), позволяющая авторизовать пользователя и создать для него *сеанс работы* в графической среде.

Производители современных «тонких клиентов» не ограничиваются реализацией протокола системы X Window. Например, компания Sun Microsystems разработала свою архитектуру графических терминалов семейства *«Sun Ray»*, включающую собственно аппаратные устройства терминала (см.

Рис. 18. «Тонкие клиенты» Sun Ray компании Sun Microsystems.

рис. 18) и ПО поддержки терминальных соединений по специализированному протоколу обмена данными. Такую терминальную систему можно организовать во многих современных ОС, включая Solaris, Linux, FreeBSD, MS Windows и др., используя одну и ту же сеть «тонких клиентов».

С другой стороны, компания Sun Microsystems разработала ПО, осуществляющее эмуляцию терминалов семейства «Sun Ray» для обычных офисных компьютеров, что позволяет динамично расширять общее число клиентов и организовывать удаленный доступ на серверы в рамках уже существующей локальной сети «тонких клиентов».

10. Инициализации и функционирование ОС

10.1. Загрузка и инициализация ядра ОС

Загрузка ОС после включения компьютера представляет собой поэтапный процесс. Несмотря на наличие большого числа реализаций ОС UNIX и аппаратных платформ, можно выделить основные этапы загрузки [5]:

1) Досистемная загрузка. После включения питания компьютера программа ПЗУ BIOS[1] проводит тестирование оборудования, определяет подключенные дисковые накопители, а затем запускается *досистемный загрузчик*. Задача этого этапа — определить, с какого дискового устройства будет идти дальнейшая загрузка, затем загрузить с него специальную программу — *загрузочный код*, и запустить ее на выполнение процессором.

2) Выполнение загрузочного кода с диска. На вычислительных платформах архитектуры x86 загрузочный код хранится в первом секторе загрузочного диска, называемом *главной загрузочной записью* или *MBR* (от англ. Master Boot Record), где также хранится таблица разделов диска. Задача этой программы — определить, где (на каком разделе диска) находится *загрузчик операционной системы*, загрузить его в память и передать ему управление.

3) Загрузка ядра ОС. Загрузчик ядра ОС представляет собой уже более сложную программу, часто имеющую интерфейс пользователя, который дает возможность выбирать конкретную ОС или параметры загрузки ядра. Чтобы продолжить загрузку, необходимо иметь доступ к *образу ядра ОС*, поэтому зачастую в код загрузчика включается поддержка ФС. Более простые загрузчики в процессе предварительной установки сохраняют адреса всех блоков диска, в которых располагается файл с образом ядра. Загрузчик читает образ ядра в определенный адрес памяти и передает туда управление. Большинство ОС имеют собственные загрузчики ядра, однако существуют и универсальные варианты, например GRUB — загрузчик, применяемый в последних версиях ОС Solaris и Linux.

4) Инициализация ядра ОС. С этого этапа фактически происходит старт ОС. В ОС Linux процесс инициализации ядра сопровождается выводом

[1] англ. Basic Input-Output System — базовая система ввода-вывода.

в консоль некоторой отладочной информация, в ОС Solaris этот процесс проходит более скрытно. В начале инициализации ядро определяет особенности данной вычислительной системы: выясняет тип и быстродействие центрального процессора, объем оперативной памяти, объем и структуру кэш-памяти; делает предположение об архитектуре компьютера в целом и многое другое. На следующем шаге определяется состав аппаратного обеспечения компьютера: тип и параметры шин передачи данных и устройств управления ими (контроллеров), список внешних устройств, доступных по шинам, настройки этих устройств (диапазон портов ввода-вывода, адреса в ПЗУ, занимаемые аппаратные прерывания, номера каналов прямого доступа к памяти и т. п.). Ядро на основании параметров, переданных ему загрузчиком, выбирает корневой раздел — файловую систему, содержащую будущий каталог `/` и его подкаталоги (для системной начальной загрузки важны каталоги `/bin`, `/etc`, `lib` и `/sbin`). Далее корневой раздел монтируется и ядро запускает первый процесс — `init` (исполняемый файл `/sbin/init`).

5) *Запуск процесса init.* Процесс `init` производит дальнейшую инициализацию системы вплоть до запуска программ-контроллеров терминалов, подготавливая систему к работе с пользователями.

10.2. Процесс init и уровни выполнения

Процесс `init` представляет собой обычный процесс ОС UNIX, выполняемый в режиме демона, но имеющий некоторые особенности: 1) процесс существует все время, пока работает система, 2) его PID всегда равен 1. Процесс `init` решает две важные задачи:

- производит инициализацию системы после загрузки ядра: монтирует все файловые системы (указанные в файле `/etc/fstab`), загружает дополнительные драйверы устройств (в виде отдельных процессов), настраивает сетевые интерфейсы, запускает служебные процессы — демоны;
- являясь родительским процессом по отношению ко всем остальным процессам в системе, «усыновляет» процессы, «родитель» которых завершился до их завершения; это гарантирует то, что для любого процесса в любой момент времени будет существовать родительский процесс.

В ОС Solaris и в большинстве дистрибутивов ОС Linux традиционно используется принцип инициализации системы в стиле UNIX System V[1]. В основу принципа инициализации ОС UNIX System V положено понятие *уровня выполнения*[2] (англ. Run Level) — режима работы ОС, определяющего набор системных процессов и предоставляемых сервисов, которые будут выполняться в данном режиме. Кроме того, переключение на тот или иной уровень выполнения может означать процедуру завершения работы ОС или перезагрузки компьютера. Перечень уровней выполнения для ОС Solaris и Linux приведен в табл. 15.

Принцип инициализации ОС в стиле UNIX System V предполагает также наличие особого унифицированного механизма сценариев инициализации и управления всеми системными службами, запускаемыми процессом `init`. При этом конфигурация списка служб (демонов) хранится отдельно для каждого из уровней выполнения.

Таблица 15. Уровни выполнения в системе UNIX System V для ОС Solaris и Linux.

Уровень в Solaris/ Linux	Англ. название	Описание
0,5/0	System halt	Останов системы (используется для завершения работы, для отключения питания компьютера после останова в Solaris используется уровень 5)
1(s)/1(s)	Single user mode	Однопользовательский режим (используется для администрирования)
2/2	Local multiuser without remote network	Многопользовательский режим без сетевого доступа
3/3	Full multiuser with network	Многопользовательский режим с сетевым доступом (основной режим для сервера)
4/4	—	Обычно не используется, но в некоторых дистрибутивах Linux соответствует уровню *Full multiuser with network and xdm* (графический вход)
-/5	Full multiuser with network and xdm	Многопользовательский режим с сетевым доступом и графическим дисплейным менеджером (основной режим для рабочей станции)
6/6	System reboot	Перезагрузка системы

[1] Альтернативный вариант — BSD-стиль, применяемый в ОС UNIX ветви BSD, а также в ряде дистрибутивов Linux.

[2] Применяется также термин *уровень запуска* или просто *режим*.

При старте ОС процесс `init` переходит (поэтапно) на один из возможных уровней выполнения из диапазона 1–5, при завершении работы — на уровни 6 или 0. Основным уровнем выполнения для серверного компьютера под управлением ОС Solaris или Linux является 3-й уровень, для рабочей станции Linux — 5-й уровень. Системный администратор root имеет возможность переключать ОС на различные уровни выполнения с помощью команды `init`, указав в качестве аргумента уровень. Например, команда `init 6` инициирует перезагрузку системы, а команда `init 0` — завершение работы и выключение. В современных дистрибутивах ОС UNIX существуют, как правило, специальные команды оболочки для выполнения указанных действий:

- `halt` или `poweroff` — завершение работы и выключение компьютера,
- `reboot` — завершение работы и перезагрузка компьютера,
- `shutdown` — завершение работы системы с возможностью указания различных опций, например, отложить выполнение команды на некоторое время, назначить проверку используемых ФС и т.п.

Команды, приводящие к изменению уровня выполнения системы, может вызывать только суперпользователь root. Вызов указанных команд можно осуществлять и удаленно через Telnet или SSH-соединение. Узнать текущий уровень выполнения можно с помощью команды `runlevel`.

Согласно принципу инициализации и управления UNIX System V, все сценарии управления службами располагаются в каталоге `/etc/rc.d` или `/etc/init.d`[1]. Механизм сценариев управления службами предполагает наличие отдельного каталога `/etc/rcN.d` (в ОС Solaris) или `/etc/rc.d/rcN.d` (в большинстве ОС Linux) для каждого `N`-го уровня выполнения, в который и помещаются сценарии данного уровня. В действительности (например, в ОС Linux), сами сценарии располагаются в каталоге `/etc/init.d`, а в каталоги `/etc/rc.d/rcN.d` помещаются символьные ссылки на них. Сценарии пишутся на языке shell, обычно в расчете на классическую оболочку Boure Shell (`sh`).

Поведение процесса `init` во многом определяется настройками в файле `/etc/inittab`, представляющем собой текстовый файл конфигурации

[1] Часто один из этих каталогов в действительности является символьной ссылкой на другой каталог.

процесса `init`, где каждая строка состоит из четырех полей, разделенных двоеточиями (строки комментариев начинаются со знака `#`):

```
id:runlevels:action:process
```

Описание полей приведено ниже.

`id` — уникальный 1–4-символьный идентификатор строки. Для строк настройки `getty` или `mingetty` указывает терминал, на котором будет запущена данная программа (символ после `/dev/tty` в имени файла устройства);

`runlevels` — перечень уровней выполнения, на которых данная строка рассматривается. Каждый уровень выполнения задается одной цифрой без разделителей;

`action` — действие, которое должно быть выполнено, например `respawn` для того, чтобы выполнить команду в следующем поле многократно или `once`, чтобы выполнить команду только один раз;

`process` — выполняемая команда.

По умолчанию, система загружается на уровень выполнения, указанный в следующей строке файла `inittab` (уровень 3 в данном случае):

```
id:3:initdefault:
```

В следующей строке указывается действие в случае нажатия комбинации клавиш `Ctrl-Alt-Del` (в данном случае — перезагрузка системы):

```
ca:12345:ctrlaltdel:/sbin/shutdown -r now
```

Программы поддержки терминалов определяются следующим образом (в данном случае — первая виртуальная консоль в ОС Linux):

```
1:2345:respawn:/sbin/mingetty tty1
```

При инициализации системы процесс `init` может выводить информацию о ходе своей работы на терминал (в ОС Linux). При этом возможны ситуации, когда процессу потребуется вмешательство системного администратора для решения тех или иных возникших проблем, например, при проверке ФС дисков. В ОС Linux можно перевести загрузку системы в интерактивный режим сразу при старте процесса `init` для пошагового выполнения каждого этапа загрузки и инициализации.

10.3. Группы и сеансы процессов

В ОС UNIX процессы образуют *группы процессов*. После создания процесса ему присваивается уникальный идентификатор (PID), а также ядро назначает процессу *идентификатор группы процессов PGID* (от англ. Process Group ID). Группа процессов включает один или более процессов и существует, пока в системе присутствует хотя бы один процесс этой группы. Идентификатор группы процессов PGID устанавливается равным идентификатору PID процесса, образовавшего данную группу посредством системного вызова `setpgid()`. Такой процесс становится *лидером группы*, и идентификатор PGID, таким образом, является уникальным. Временной интервал, начинающийся с создания группы и заканчивающийся, когда последний процесс ее покинет, называется *временем жизни группы*. Последний процесс может либо завершить свое выполнение, либо перейти в другую группу (если он не был лидером группы).

Многие системные вызовы могут быть применены как к единичному процессу, так и ко всем процессам группы. Например, системный вызов `kill()` может отправить сигнал как одному процессу, так и всем процессам указанной группы. Точно так же функция `waitpid()` позволяет родительскому процессу ожидать завершения конкретного процесса или любого процесса группы.

Каждый процесс, помимо этого, является членом *сеанса* (англ. session), являющегося набором одной или нескольких групп процессов. Понятие сеанса было введено в ОС UNIX для логического объединения процессов, а точнее групп процессов, созданных в результате регистрации и последующей работы пользователя в системе, а также для создания изолированного окружения для процесса-демона и его потомков. Таким образом, термин «сеанс работы» в системе тесно связан с понятием сеанса, описывающего набор процессов, которые порождены пользователем за время пребывания в системе. *Идентификатор сеанса процесса SID* (от англ. Session ID) — это идентификатор группы PGID процесса, который является *лидером сеанса*.

Системные вызовы `getpgrp()` и `getpgid()` позволяют узнать идентификатор группы PGID процесса. Системные вызовы `setpgrp()` и `setpgid()` позволяют изменить значение PGID процесса, т. е. сделать процесс членом

существующей группы или создать новую группу процессов, при условии, что обе группы процессов являются членами одного сеанса. Процесс имеет возможность установить идентификатор группы для себя и для своих потомков (дочерних процессов). Однако процесс не может изменить идентификатор группы для дочернего процесса, который выполнил системный вызов `exec()`, запускающий на выполнение другую программу [3].

Системный вызов `getsid()` возвращает идентификатор сеанса процесса. Вызов функции `setsid()` приводит к созданию нового сеанса. Новый сеанс создается лишь при условии, что процесс не является лидером какого-либо сеанса. В случае успеха процесс становится лидером сеанса и лидером новой группы.

Понятия группы и сеанса тесно связаны с терминалом или, точнее, с драйвером терминала. Каждый сеанс может иметь один ассоциированный терминал, который называется *управляющим терминалом*, а группы, созданные в данном сеансе, наследуют этот управляющий терминал. Наличие управляющего терминала позволяет ядру контролировать стандартный ввод/вывод процессов, а также дает возможность отправить сигнал всем процессам ассоциированной с терминалом группы, например, при его отключении. Типичным примером является регистрация и работа пользователя в системе. При входе в систему терминал пользователя становится управляющим для лидера сеанса (в данном случае для командного интерпретатора shell) и всех процессов, порожденных лидером (в данном случае для всех процессов, которые запускает пользователь из командной строки интерпретатора). При выходе пользователя из системы shell завершает свою работу и таким образом отключается от управляющего терминала, что вызывает отправление сигнала `SIGHUP` всем незавершенным процессам текущей группы. Это гарантирует, что после завершения работы пользователя в системе не останется запущенных им активных процессов [3].

Список литературы

[1] Иртегов Д.В. Введение в операционные системы. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2012. — 1040 с.

[2] Глушков В.М. Энциклопедия кибернетики. — в 2-х т. — Киев, 1974.

[3] Робачевский А.М. Операционная система UNIX. — СПб.: БХВ-Санкт-Петербург, 1999. — 528 с.

[4] Дунаев С.Б. UNIX-сервер. Настройка, конфигурирование, работа в операционной среде, Internet-возможности. В 2-х т. — М.: «ДИАЛОГ-МИФИ», 1999. — 304 с.

[5] Федосеев А. UNIX: учебный курс. — 2006.
http://www.openspin.org/materials/courses/admin/index.html

[6] Курячий Г.В. Операционная система UNIX. — Интернет университет информационных технологий, 2004.
http://www.intuit.ru/department/os/osunix/

[7] Кравчук В. Основы операционной системы UNIX. — 2004.
http://www.opennet.ru/docs/RUS/unix_basic/index.html

[8] Соловьев А. Программирование на Shell (UNIX). — Учебное пособие.
http://www.citforum.ru/programming/shell/index.shtml

[9] Торчинский Ф.И. Организация UNIX-систем и ОС Solaris 9. — Интернет университет информационных технологий, 2005.
http://www.intuit.ru/department/os/ossolaris/

[10] Торчинский Ф.И. Администрирование ОС Solaris 9. — Интернет университет информационных технологий, 2005.
http://www.intuit.ru/department/os/adminsolaris/

[11] Торчинский Ф.И., Ильин Е.С. Введение в администрирование ОС Solaris 10. — Интернет университет информационных технологий, 2008.
http://www.intuit.ru/department/os/intadmsolaris10/

[12] Торчинский Ф.И., Ильин Е.С. Системное администрирование ОС Solaris 10. — Интернет университет информационных технологий, 2008.
http://www.intuit.ru/department/os/sysadmsolaris10/

[13] Хабибуллин И.Ш. Указания по работе в UNIX. Ч.1. — Казань: Казан. гос. ун-т, 1996. — 56 с.

[14] Хабибуллин И.Ш. Указания по работе в UNIX. Ч.2. — Казань: Казан. гос. ун-т, 1996. — 37 с.

[15] Виртуальный компьютерный музей. Интернет ресурс.
http://www.computer-museum.ru

[16] Ланина Э.П. История развития вычислительной техники. — ИрГТУ, Иркутск, 2001 г. — 166 с.

[17] Джермейн К. Программирование на IBM/360. — М.: Мир, 1978. — 870 с.

[18] Галатенко В.А. Программирование в стандарте POSIX. — Интернет университет информационных технологий, 2004.
http://www.intuit.ru/department/se/pposix/

[19] IEEE Std 1003.1, 2004 Edition.
http://www.unix.org/version3/ieee_std.html

[20] Таненбаум Э. Современные операционные системы. — СПб.: Издательский дом Питер, 2002. — 1037 с.

[21] Шагурин И.И., Бродин В.Б., Мозговой Г.П. 80386: описание и система команд. — М.: Малип, 1992. — 160 с.

[22] Рыжов Е., Карпов А. Архитектура AMD64 (EM64T). — ООО «СиПроВер», 2008. https://www.realcoding.net/articles/arkhitektura-amd64-em64t.html

[23] AMD64 Architecture Programmer's Manual. Volume 1: Application Programming. — AMD, 2009. — 304 с.
http://support.amd.com/TechDocs/24592.pdf

[24] AMD64 Architecture Programmer's Manual. Volume 2: System Programming. — AMD, 2009. — 494 с.
http://support.amd.com/TechDocs/24593.pdf

[25] Карпов В.Е., Коньков К.А. Основы операционных систем. Курс лекций. — Интернет университет информационных технологий, 2004.
http://www.intuit.ru/studies/courses/2192/31/info

[26] Filesystem Hierarchy Standard (Version 2.3). — 2004.
http://www.pathname.com/fhs/pub/fhs-2.3.pdf

[27] Немет Э. и др. UNIX: руководство системного администратора. / Э. Немет, Г. Снайдер, С. Сибас, Т.Р. Хейн. — К.: BHV, 2000. — 832 с.

[28] Немет Э. и др. UNIX: руководство системного администратора. Для профессионалов. 3-е изд. / Э. Немет, Г. Снайдер, С. Сибас, Т.Р. Хейн. — СПб.: Питер; К.: BHV, 2003. — 925 с.

[29] Bach M.J. The design of the UNIX operating system. Prentice-Hall Inc. — Englewood Cliffs, New Jersey, 1986. — 471 с.

[30] Бах Дж.М. Архитектура операционной системы UNIX. — 1986.
http://www.opennet.ru/docs/RUS/unix/

[31] Кузнецов С.Д. Операционная система UNIX. — CITForum.
http://citforum.ru/operating_systems/unix/contents.shtml

[32] Стивенс У. UNIX: взаимодействие процессов. — СПб.: Питер, 2003. — 573 с.

[33] Карпов В.Е., Коньков К.А. Основы операционных систем. Практикум. — Интернет университет информационных технологий, 2004.
http://www.intuit.ru/studies/courses/2249/52/info

[34] Торчинский Ф.И., Ильин Е.С. Операционная система Solaris. — Бином, 2009. — 600 с.

Алфавитный указатель