### Обзор операционной системы (ОС) Linux

### 1. Интерфейсы системы Linux и структура ядра Linux

В ранние годы разработки системы MINIX и обсуждения этой системы в Интернете многие люди просили (а часто требовали) все больше новых и более сложных функций, и на эти просьбы Эндрю Таненбаум часто отвечал отказом (чтобы сохранить небольшой размер системы, которую студенты могли бы полностью освоить за один семестр). Эти постоянные отказы раздражали многих пользователей. В те времена бесплатной системы FreeBSD еще не было. Наконец, через несколько лет финский студент Линус Торвальдс (Linus Torvalds) решил сам написать еще один клон системы UNIX, который он назвал

**Linux**. Это должна была быть полноценная производственная система, со многими изначально отсутствовавшими в системе MINIX функциями. Первая версия 0.01 операционной системы Linux была выпущена в 1991 году. Она была разработана и собрана на компьютере под управлением MINIX и заимствовала из системы MINIX множество идей, начиная со структуры дерева исходных кодов и заканчивая компоновкой файловой системы. Однако в отличие от микроядерной системы MINIX, Linux была монолитной системой, то есть вся операционная система размещалась в ядре. Размер исходного текста составил 9300 строк на языке C и 950 строк на ассемблере, что приблизительно совпадало с версией MINIX как по размеру, так и по функциональности. Фактически это была переделка системы MINIX — единственной системы, исходный код которой имелся у Торвальдса.

Операционная система Linux быстро росла в размерах и впоследствии развилась в полноценный клон UNIX с виртуальной памятью, более сложной файловой системой и многими другими дополнительными функциями. Хотя изначально система Linux работала только на процессоре Intel 386 (и даже имела встроенный ассемблерный код 386-го процессора в процедурах на языке C), она была быстро перенесена на другие платформы и теперь работает на широком спектре машин — так же, как и UNIX. Следует выделить одно отличие системы Linux от UNIX: она использует многие специальные возможности компилятора gcc, поэтому потребуется приложить немало усилий, чтобы откомпилировать ее стандартным ANSI C-компилятором.

*GNU Compiler Collection (обычно используется сокращение GCC) — набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU.*

*Начало GCC было положено Ричардом Столлманом, который реализовал первый вариант GCC в 1985 году на нестандартном и непереносимом диалекте языка Паскаль; позднее компилятор был переписан на языке Си Леонардом Тауэром (англ. Leonard H. Tower Jr.) и Ричардом Столлманом и выпущен в 1987 году как компилятор для проекта GNU, который сам по себе являлся свободным программным обеспечением. Разработка GCC курируется Free Software Foundation.*

*В настоящее время GCC поддерживается группой программистов со всего мира. GCC является лидером по количеству процессоров и операционных систем, которые он поддерживает.*

*Будучи официальным компилятором системы GNU, GCC также является главным компилятором для сборки ряда других операционных систем; среди них — различные варианты Linux и BSD, а также ReactOS, Mac OS X, OpenSolaris, NeXTSTEP, BeOS и Haiku.*

Следующим основным выпуском системы Linux была версия 1.0, появившаяся в 1994 году. Она состояла примерно из 165 000 строк кода и включала новую файловую систему, отображение файлов на адресное пространство памяти и совместимое с BSD сетевое программное обеспечение с сокетами и TCP/IP. Она также включала многие новые драйверы устройств. В течение следующих двух лет выходили версии с незначительными исправлениями.

К этому времени операционная система Linux стала достаточно совместимой с UNIX, поэтому на нее было перенесено большое количество программного обеспечения для UNIX, что значительно увеличило ее полезность. Кроме того, операционная система Linux привлекла большое количество людей, которые начали работу над ее кодом и расширением (под общим руководством Торвальдса).

Следующий главный выпуск, версия 2.0, вышел в свет в 1996 году. Эта версия состояла примерно из 470 000 строк на языке C и 8000 строк ассемблерного кода. Она включала в себя поддержку 64-разрядной архитектуры, симметричной многозадачности, новых сетевых протоколов и прочих многочисленных функций. Значительную часть общей массы исходного кода составляла обширная коллекция драйверов устройств для постоянно растущего количества поддерживаемых периферийных устройств. Следом за этой версией довольно часто выходили дополнительные выпуски.

Номер версии ядра Linux состоит из четырех чисел: A.B.C.D (например, 2.6.9.11). Первое число обозначает версию ядра. Второе число обозначает основную версию. До ядра 2.6 четные номера версии обозначали стабильную версию ядра, а нечетные — не стабильную (находящуюся в разработке). Начиная с версии ядра 2.6 это не так. Третье число обозначает номер ревизии (например, добавлена поддержка новых драйверов). Четвертое число обозначает исправление ошибок или заплатки системы безопасности. В июле 2011 года Линус Торвальдс анонсировал выпуск Linux 3.0, но не из-за каких-то существенных технических усовершенствований, а просто в честь 20-й годовщины разработки ядра. По состоянию на 2013 год ядро Linux содержит около 16 млн строк кода.

В систему Linux была перенесена внушительная часть стандартного программного обеспечения UNIX, включая популярную оконную систему X Windows и большое количество сетевого программного обеспечения. Кроме того, специально для Linux было написано два различных конкурирующих графических интерфейса пользователя: GNOME и KDE.

Необычной особенностью Linux является ее бизнес-модель: это бесплатное программное обеспечение. Его можно скачать с различных интернет-сайтов, например [www.kernel.org.](http://www.kernel.org/) Система Linux поставляется вместе с лицензией, разработанной Ричардом Столманом, основателем Фонда бесплатных программ (Free Software Foundation). Несмотря на то что система Linux бесплатна, эта лицензия, называющаяся **GPL** (GNU Public License — общедоступная лицензия GNU), по длине превосходит лицензию корпорации Microsoft для операционной системы Windows и указывает, что вы можете и чего не можете делать с кодом. Пользователи могут бесплатно использовать, копировать, модифицировать и распространять исходные коды и двоичные файлы. Основное ограничение касается отдельной продажи или распространения двоичного кода (выполненного на основе ядра Linux) без исходных текстов. Исходные коды (тексты) должны либо поставляться вместе с двоичными файлами, либо предоставляться по требованию.

Хотя Торвальдс до сих пор довольно внимательно контролирует ядро системы, большое количество программ пользовательского уровня было написано другими программистами, многие из которых изначально перешли на Linux из сетевых сообществ MINIX, BSD и GNU. Однако по мере развития системы Linux все меньшая часть сообщества Linux желает ковыряться в исходном коде (свидетельством тому служат сотни книг, описывающих, как установить систему Linux и как ею пользоваться, и только несколько книг, в которых обсуждается сам код или то, как он работает). Кроме того, многие пользователи Linux теперь предпочитают бесплатному скачиванию системы из Интернета покупку одного из CD-ROM-дистрибутивов, распространяемых многочисленными коммерческими компаниями. На веб-сайте [www.linux.org](http://www.linux.org/) перечислено более 100 компаний, продающих различные дистрибутивы Linux. Кроме того, информацию о дистрибутивах Linux и их распространителях можно найти на www.distrowatch.org. По мере того как все больше и больше занимающихся программным обеспечением компаний начинают продавать свои версии Linux и все большее число производителей компьютеров поставляют систему Linux со своими машинами, граница между коммерческим и бесплатным программным обеспечением начинает заметно размываться.

Интересно отметить, что когда мода на Linux начала набирать обороты, она получила поддержку с неожиданной стороны — от корпорации AT&T. В 1992 году университет в Беркли, лишившись финансирования, решил прекратить разработку BSD UNIX на последней версии 4.4BSD (которая впоследствии послужила основой для FreeBSD).

Поскольку эта версия по существу не содержала кода AT&T, университет в Беркли выпустил это программное обеспечение с лицензией открытого исходного кода, которая позволяла всем делать все, что угодно, кроме одной вещи — подавать в суд на университет Калифорнии. Контролировавшее систему UNIX подразделение корпорации AT&T отреагировало немедленно — вы угадали, как, — подав в суд на университет Калифорнии.

Оно также подало иск против компании BSDI, созданной разработчиками BSD UNIX для упаковки системы и продажи поддержки (примерно так сейчас поступают компании типа Red Hat с операционной системой Linux). Поскольку код AT&T практически не использовался, то судебное дело основывалось на нарушении авторского права и торговой марки, включая такие моменты, как телефонный номер 1-800-ITSUNIX компании BSDI. Хотя этот спор в конечном итоге удалось урегулировать в досудебном порядке, он не позволял выпустить на рынок FreeBSD в течение долгого периода — достаточного для того, чтобы система Linux успела упрочить свои позиции. Если бы судебного иска не было, то уже примерно в 1993 году началась бы серьезная борьба между двумя бесплатными версиями системы UNIX, распространяющимися с исходными кодами: царствующим чемпионом — системой BSD (зрелой и устойчивой системой с многочисленными приверженцами в академической среде еще с 1977 года) и энергичным молодым претендентом — системой Linux всего лишь двух лет от роду, но с уже растущим числом последователей среди индивидуальных пользователей. Кто знает, чем обернулась бы эта схватка двух бесплатных версий системы UNIX.

## Обзор системы Linux. Задачи Linux. Операционная система UNIX всегда была интерактивной системой, разработанной для одновременной поддержки множества процессов и множества пользователей. Она была разработана программистами и для программистов — чтобы использовать ее в такой среде, в которой большинство пользователей достаточно опытны и занимаются проектами (часто довольно сложными) разработки программного обеспечения. Во многих случаях большое количество программистов активно работает над созданием общей системы, поэтому в операционной системе UNIX есть большое количество средств, позволяющих людям работать вместе и управлять совместным использованием информации. Очевидно, что модель группы опытных программистов, совместно работающих над созданием сложного программного обеспечения, существенно отличается от модели одного начинающего пользователя, сидящего за персональным компьютером в текстовом процессоре, и это отличие отражается в операционной системе UNIX от начала до конца. Совершенно естественно, что Linux унаследовал многие из этих установок, даже несмотря на то что первая версия предназначалась для персонального компьютера.

## Чего действительно хотят от операционной системы хорошие программисты? Прежде всего, большинство хотело бы, чтобы их система была простой, элегантной и совместимой. Например, на самом нижнем уровне файл должен представлять собой просто набор байтов. Наличие различных классов файлов для последовательного и произвольного доступа, доступа по ключу, удаленного доступа и т. д. (как это реализовано на мейнфреймах) просто является помехой. А если команда

ls A\*

означает вывод списка всех файлов, имя которых начинается с буквы «A», то команда

rm A\*

должна означать удаление всех файлов, имя которых начинается с буквы «A», а не одного файла, имя которого состоит из буквы «A» и звездочки. Эта характеристика иногда называется *принципом наименьшей неожиданности* (principle of least surprise).

Другие свойства, которые, как правило, опытные программисты желают видеть в операционной системе, — это мощь и гибкость. Это означает, что в системе должно быть небольшое количество базовых элементов, которые можно комбинировать, чтобы приспособить их для конкретного приложения. Одно из основных правил системы Linux заключается в том, что каждая программа должна выполнять всего одну функцию — и делать это хорошо. То есть компиляторы не занимаются созданием листингов, так как другие программы могут лучше справиться с этой задачей.

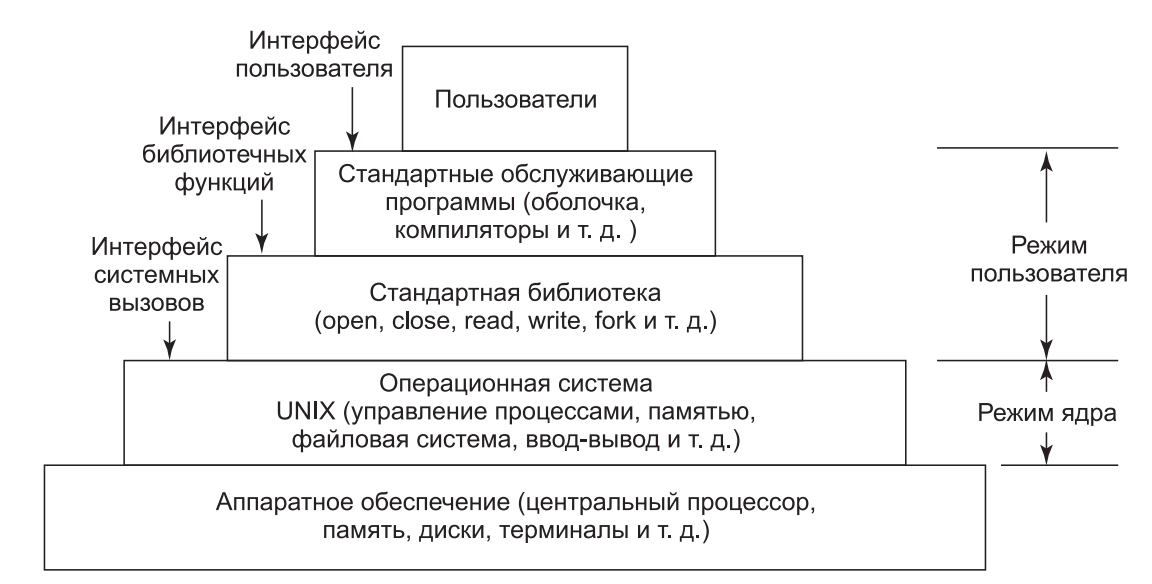
Наконец, у большинства программистов есть сильная неприязнь к бесполезной избыточности. Зачем писать *copy*, когда вполне достаточно *cp*, чтобы однозначно выразить желаемое? Это же пустая трата драгоценного хакерского времени. Чтобы получить список всех строк, содержащих строку «ard», из файла f, программист в операционной системе Linux вводит команду

grep ard f

Противоположный подход состоит в том, что программист сначала запускает программу grep (без аргументов), после чего программа grep приветствует программиста фразой: «Здравствуйте, я grep. Я ищу шаблоны в файлах. Пожалуйста, введите ваш шаблон». Получив шаблон, программа grep запрашивает имя файла. Затем она спрашивает, есть ли еще какие-либо файлы. Наконец, она выводит резюме того, что она собирается делать, и спрашивает, все ли верно. Хотя такой тип пользовательского интерфейса может быть удобен для начинающих пользователей, он бесконечно раздражает опытных программистов. Им требуется слуга, а не нянька.

### Интерфейсы системы Linux. Операционную систему Linux можно рассматривать как пирамиду (рис.1). У основания пирамиды располагается аппаратное обеспечение, состоящее из центрального процессора, памяти, дисков, монитора и клавиатуры, а также других устройств. Операционная система работает на «голом железе». Ее функция заключается в управлении аппаратным обеспечением и предоставлении всем программам интерфейса системных вызовов. Эти системные вызовы позволяют программам пользователя создавать процессы, файлы и прочие ресурсы, а также управлять ими.

### Программы делают системные вызовы, помещая аргументы в регистры (или иногда в стек) и выполняя команду эмулированного прерывания для переключения из пользовательского режима в режим ядра. Поскольку на языке C невозможно написать команду эмулированного прерывания, то этим занимается библиотека, в которой есть по одной процедуре на системный вызов.

****

**Рис.1.** Уровни операционной системы Linux

Эти процедуры написаны на ассемблере, но они могут вызываться из языка C. Каждая такая процедура сначала помещает аргументы в нужное место, а затем выполняет команду эмулированного прерывания. Таким образом, чтобы обратиться к системному вызову *read*, программа на языке C должна вызвать библиотечную процедуру *read*. Кстати, в стандарте POSIX определен именно интерфейс библиотечных функций, а не интерфейс системных вызовов. Иначе говоря, стандарт POSIX определяет, какие библиотечные процедуры должна предоставлять соответствующая его требованиям система, каковы их параметры, что они должны делать и какие результаты возвращать. В стандарте даже не упоминаются реальные системные вызовы.

Помимо операционной системы и библиотеки системных вызовов все версии Linux предоставляют большое количество стандартных программ, некоторые из них указаны в стандарте POSIX 1003.2, тогда как другие могут различаться в разных версиях системы Linux. К этим программам относятся командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами. Именно эти программы и запускает пользователь с клавиатуры. Таким образом, мы можем говорить о трех интерфейсах в операционной системе Linux: интерфейсе системных вызовов, интерфейсе библиотечных функций и интерфейсе, образованном набором стандартных служебных программ.

В большинстве наиболее распространенных дистрибутивов системы Linux для персональных компьютеров этот ориентированный на ввод с клавиатуры интерфейс пользователя был заменен графическим интерфейсом пользователя, ориентированным на использование мыши, для чего не потребовалось никаких изменений в самой системе. Именно эта гибкость сделала систему Linux такой популярной и позволила ей пережить многочисленные изменения лежащей в ее основе технологии.

Графический интерфейс пользователя системы Linux похож на первые графические интерфейсы пользователя, разработанные для UNIX в 70-х годах прошлого века и ставшие популярными благодаря компьютерам Macintosh и впоследствии — системе Windows для персональных компьютеров. Графический интерфейс пользователя создает среду рабочего стола — знакомую нам метафору с окнами, значками, каталогами, панелями инструментов, а также возможностью перетаскивания. Полная среда рабочего стола содержит администратор многооконного режима, который управляет размещением и видом окон, а также различными приложениями и создает согласованный графический интерфейс. Популярными средами рабочего стола для Linux являются GNOME (GNU Network Object Model Environment) и KDE (K Desktop Environment).

Графические интерфейсы пользователя в Linux поддерживает оконная система X Windowing System, которую обычно называют Х11 (или просто Х). Она определяет обмен и протоколы отображения для управления окнами на растровых дисплеях UNIXподобных систем. Х-сервер является главным компонентом, который управляет такими устройствами, как клавиатура, мышь и экран, и отвечает за перенаправление ввода или прием вывода от клиентских программ. Реальная среда графического интерфейса пользователя обычно построена поверх библиотеки низкого уровня (xlib), которая содержит функциональность для взаимодействия с Х-сервером. Графический интерфейс расширяет базовую функциональность Х11, улучшая вид окон, предоставляя кнопки, меню, значки и пр. Х-сервер можно запустить вручную из командной строки, но обычно он запускается во время загрузки диспетчером окон, который отображает графический экран входа в систему.

При работе на Linux-системах с помощью графического интерфейса пользователь может щелчком кнопки мыши запустить приложение или открыть файл, использовать перетаскивание для копирования файлов из одного места в другое и т. д. Кроме того, пользователи могут запускать программу эмуляции терминала xterm, которая предоставляет им базовый интерфейс командной строки операционной системы. Его описание дано в следующем разделе.

### Оболочка. Несмотря на то что Linux имеет графический интерфейс пользователя, большинство программистов и продвинутые пользователи по-прежнему предпочитают интерфейс командной строки, называемый оболочкой (shell). Они часто запускают одно или несколько окон с оболочками из графического интерфейса пользователя и работают в них. Интерфейс командной строки оболочки значительно быстрее в использовании, существенно мощнее, прост в расширении и не грозит пользователю туннельным синдромом запястья из-за необходимости постоянно пользоваться мышью. Далее мы кратко опишем оболочку bash. Она основана на оригинальной оболочке системы UNIX, которая называется оболочкой Бурна (Bourne shell, написана Стивом Бурном, а затем в Bell Labs.), и фактически даже ее название является сокращением от Bourne Again SHell. Используется и множество других оболочек (ksh, csh и т. д.), но bash является оболочкой по умолчанию в большинстве Linux-систем.

Когда оболочка запускается, она инициализируется, а затем выводит на экран символ приглашения к вводу (обычно это знак процента или доллара) и ждет, когда пользователь введет командную строку.

После того как пользователь введет командную строку, оболочка извлекает из нее первое слово, под которым подразумевается череда символов с пробелом или символом табуляции в качестве разделителя. Оболочка предполагает, что это слово является именем запускаемой программы, ищет эту программу и, если находит, запускает ее навыполнение. При этом работа оболочки приостанавливается на время работы запущенной программы. По завершении работы программы оболочка пытается прочитать следующую команду. Здесь важно подчеркнуть, что оболочка представляет собой обычную пользовательскую программу. Все, что ей нужно, — это возможность чтения с клавиатуры и вывода на монитор, а также способность запускать другие программы.

У команд могут быть аргументы, которые передаются запускаемой программе в виде текстовых строк. Например, командная строка

cp src dest

запускает программу cp с двумя аргументами, *src* и *dest*. Эта программа интерпретирует первый аргумент как имя существующего файла. Она копирует этот файл и называет эту копию dest.

Не все аргументы являются именами файлов. В строке

head –20 file

первый аргумент *–20* дает указание программе head напечатать первые 20 строк файла file (вместо принятых по умолчанию 10 строк). Управляющие работой команды или указывающие дополнительные значения аргументы называются **флагами** и по соглашению обозначаются знаком тире. Тире требуется, чтобы избежать двусмысленности, поскольку, например, команда

head 20 file

вполне законна. Она дает указание программе head вывести первые 10 строк файла с именем 20, а затем вывести первые 10 строк второго файла file. Большинство команд Linux-систем могут принимать несколько флагов и аргументов.

Чтобы было легче указывать группы файлов, оболочка принимает так называемые **волшебные символы** (magic charecters), иногда называемые также **групповыми** (wild cards). Например, символ «звездочка» означает все возможные текстовые строки, так что строка

ls \*.c

дает указание программе ls вывести список всех файлов, имена которых оканчиваются на .c. Если существуют файлы x.c, y.c и z.c, то данная команда эквивалентна команде

ls x.c y.c z.c

Другим групповым символом является вопросительный знак, который заменяет один любой символ. Кроме того, в квадратных скобках можно указать множество символов, из которых программа должна будет выбрать один. Например, команда

ls [ape]\*

выводит все файлы, имя которых начинается с символов «a», «p» или «e».

Такая программа, как оболочка, не должна открывать терминал (клавиатуру и монитор), чтобы прочитать с него или сделать на него вывод. Вместо этого запускаемые программы автоматически получают доступ для чтения к файлу, называемому **стандартным устройством ввода** (standard input), а для записи — к файлу, называемому **стандартным устройством вывода** (standard output), и к файлу, называемому **стандартным устройством для вывода сообщений об ошибках** (standard error). По умолчанию всем этим трем устройствам соответствует терминал, то есть чтение со стандартного ввода производится с клавиатуры, а запись в стандартный вывод (или в вывод для ошибок) попадает на экран. Многие Linux-программы читают данные со стандартного устройства ввода и пишут на стандартное устройство вывода. Например, команда sort запускает программу sort, читающую строки с терминала (пока пользователь не нажмет комбинацию клавиш Ctrl+D, чтобы обозначить конец файла), а затем сортирует их в алфавитном порядке и выводит результат на экран.

Стандартные ввод и вывод можно перенаправить, что является очень полезным свойством. Для этого используются символы «<» и «>» соответственно. Разрешается их одновременное использование в одной командной строке. Например, команда sort <in >out заставляет программу sort взять в качестве входного файл in и направить вывод в файл out. Поскольку стандартный вывод сообщений об ошибках не был перенаправлен, то все сообщения об ошибках попадут на экран. Программа, которая считывает данные со стандартного устройства ввода, выполняет определенную обработку этих данных и записывает результат в поток стандартного вывода, называется **фильтром**.

Рассмотрим следующую командную строку, состоящую из трех отдельных команд: sort <in >temp; head –30 <temp; rm temp

Сначала запускается программа sort, которая принимает данные из файла in и записывает результат в файл temp. Когда она завершает свою работу, оболочка запускает программу head, дав ей указание вывести первые 30 строк из файла temp на стандартное устройство вывода, которым по умолчанию является терминал. Наконец, временный файл temp удаляется. При этом он удаляется безвозвратно и уже не может быть восстановлен.

Часто используются командные строки, в которых первая программа в командной строке формирует вывод, используемый второй программой в качестве входа. В приведенном ранее примере для этого использовался временный файл temp. Однако система Linux предоставляет для этого более простой способ. В командной строке sort <in | head –30 используется вертикальная черта, называемая **символом канала** (pipe symbol), она означает, что вывод программы sort должен использоваться в качестве входа для программы head, что позволяет обойтись без создания, использования и удаления временного файла. Набор команд, соединенных символом канала, называется **конвейером** (pipeline) и может содержать произвольное количество команд. Пример четырехкомпонентного конвейера показан в следующей строке:

grep ter \* .t | sort | head –20 | tail –5 >foo

Здесь в стандартное устройство вывода записываются все строки, содержащие строку «ter» во всех файлах, оканчивающихся на .t, после чего они сортируются. Первые 20 строк выбираются программой head, которая передает их программе tail, записывающей последние пять строк (то есть строки с 16-й по 20-ю в отсортированном списке) в файл foo. Вот пример того, как операционная система Linux обеспечивает основные строительные блоки (фильтры), каждый из которых выполняет определенную работу, а также механизм, позволяющий объединять их практически неограниченным количеством способов.

Linux является универсальной многозадачной системой. Один пользователь может одновременно запустить несколько программ, каждую в виде отдельного процесса. Синтаксис оболочки для запуска фонового процесса состоит в использовании амперсанда в конце строки. Таким образом, строка

wc –l <a >b &

запустит программу подсчета количества слов wc, которая сосчитает число строк (флаг *–l*) во входном файле a и запишет результат в файл b, но будет делать это в фоновом режиме. Как только команда будет введена пользователем, оболочка выведет символ приглашения к вводу и будет готова к обработке следующей команды. Конвейеры также могут выполняться в фоновом режиме, например:

sort <x | head &

Можно одновременно запустить в фоновом режиме несколько конвейеров.

Список команд оболочки может быть помещен в файл, а затем можно будет запустить оболочку с этим файлом в качестве стандартного входа. Вторая программа оболочки просто выполнит перечисленные в этом файле команды одну за другой — точно так же, как если бы эти команды вводились с клавиатуры. Файлы, содержащие команды оболочки, называются **сценариями оболочки** (shell scripts). Сценарии оболочки могут присваивать значения переменным оболочки и затем считывать их. Они также могут иметь параметры и использовать конструкции *if*, *for*, *while* и *case*. Таким образом, сценарии оболочки представляют собой настоящие программы, написанные на языке оболочки. Существует альтернативная оболочка Berkley C, разработанная таким образом, чтобы сценарии оболочки (и язык команд вообще) выглядели как можно более похожими на программы на языке C. Поскольку оболочка представляет собой всего лишь пользовательскую программу, было написано много различных оболочек. Пользователи могут подобрать ту из них, которая им больше нравится.

### Утилиты Linux. Пользовательский интерфейс командной строки (оболочки) Linux состоит из большого числа стандартных служебных программ, называемых также утилитами. Грубо говоря, эти программы можно разделить на шесть следующих категорий:

* + - 1. Команды управления файлами и каталогами.
      2. Фильтры.
      3. Средства разработки программ, такие как текстовые редакторы и компиляторы.
      4. Текстовые процессоры.
      5. Системное администрирование.
      6. Разное.

Стандарт POSIX 1003.1-2008 определяет синтаксис и семантику около 150 этих программ, в основном относящихся к первым трем категориям. Идея стандартизации данных программ заключается в том, чтобы можно было писать сценарии оболочки, которые работали бы на всех системах Linux.

Помимо этих стандартных утилит существует еще масса прикладных программ, таких как веб-браузеры, проигрыватели мультимедийных файлов, программы просмотра изображений, офисные пакеты и т. д.

Рассмотрим несколько примеров этих утилит, начиная с программ для управления файлами и каталогами. Команда

cp a b

копирует файл a в b, не изменяя исходный файл. Команда

mv a b

напротив, копирует файл a в b, но удаляет исходный файл. В результате она не копирует файл, а перемещает его. Несколько файлов можно сцепить в один при помощи команды *cat*, считывающей все входные файлы и копирующей их один за другим в стандартный выходной поток. Удалить файлы можно командой *rm*. Команда *chmod* позволяет владельцу изменить права доступа к файлу. Каталоги можно создать командой *mkdir* и удалить командой *rmdir*. Список файлов можно увидеть при помощи команды *ls*. У этой команды множество флагов, управляющих видом формируемого ею листинга. При помощи одних флагов можно задать, насколько подробно будет отображаться каждый файл (размер, владелец, группа, дата создания), другими флагами задается порядок, в котором перечисляются файлы (по алфавиту, по времени последнего изменения, в обратном порядке), третья группа флагов позволяет задать расположение списка файлов на экране и т. д.

Мы уже рассматривали несколько фильтров: команда *grep* извлекает из стандартного входного потока (или из одного или нескольких файлов) строки, содержащие заданную последовательность символов; команда *sort* сортирует входной поток и выводит данные в стандартный выходной поток; команда *head* извлекает первые несколько строк; команда *tail*, напротив, выдает на выход указанное количество последних строк. Кроме того, стандартом 1003.2 определены такие фильтры, как *cut* и *paste*, которые позволяют вырезать из файлов и вставлять в файлы куски текста, команда *od* конвертирует (обычно двоичный) вход в ASCII-строку (в восьмеричный, десятичный или шестнадцатеричный формат), команда *tr* преобразует символы (например, из нижнего регистра в верхний), а команда *pr* форматирует выход для вывода на принтер, позволяя добавлять заголовки, номера страниц и т. д.

Компиляторы и программные средства включают в себя компилятор gcc с языка C и программу ar, собирающую библиотечные процедуры в архивные файлы.

Еще одним важным инструментом является команда *make*, используемая для сборки больших программ, исходный текст которых состоит из множества файлов. Как правило, некоторые из этих файлов представляют собой **заголовочные файлы** (header files), содержащие определения типов, переменных, макросов и прочие декларации. Исходные файлы обычно ссылаются на эти файлы с помощью специальной директивы *include*. Таким образом, два и более исходных файла могут совместно использовать одни и те же декларации. Однако если файл заголовков изменен, то необходимо найти все исходные файлы, зависящие от него, и перекомпилировать их. Задача команды *make* заключается в том, чтобы отслеживать, какой файл от какого заголовочного файла зависит, и автоматически запускать компилятор для тех файлов, которые требуется перекомпилировать. Почти все программы в системе Linux, кроме самых маленьких, компилируются с помощью команды *make*. Часть команд POSIX перечислена в табл.1 вместе с кратким описанием. Во всех версиях операционной системы Linux есть эти программы, а также многие другие.

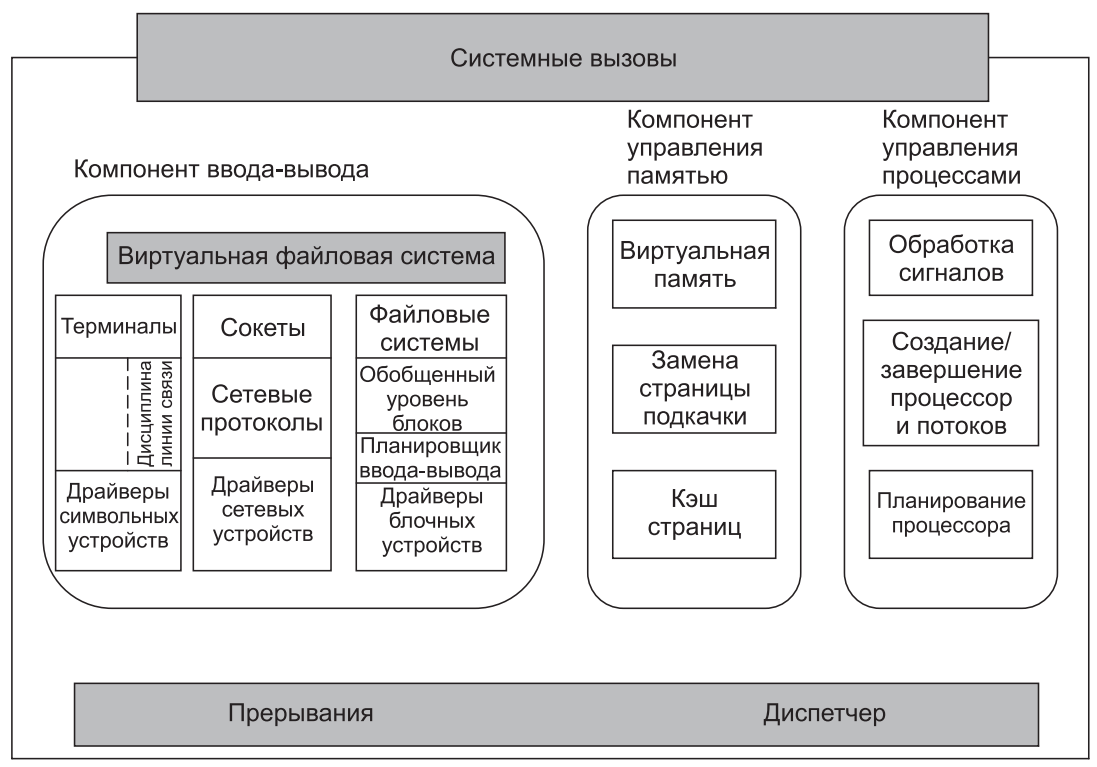
**Таблица 1.** Некоторые утилиты Linux, требуемые стандартом POSIX

|  |  |
| --- | --- |
| **Программа** | **Функция** |
| cat | Конкатенация нескольких файлов в стандартный выходной поток |
| chmod | Изменение режима защиты файла |
| cp | Копирование файлов |
| cut | Вырезание колонок текста из файла |
| grep | Поиск определенного шаблона в файле |
| head | Извлечение из файла первых строк |
| ls | Распечатка каталога |
| make | Компиляция файлов для создания двоичного файла |
| mkdir | Создание каталога |
| od | Восьмеричный дамп файла |
| paste | Вставка колонок текста в файл |
| pr | Форматирование файла для печати |
| rm | Удаление файлов |
| rmdir | Удаление каталога |
| sort | Сортировка строк файла по алфавиту |
| tail | Извлечение из файла последних строк |
| tr | Преобразование символов из одного набора в другой |

### Структура ядра. На рис.1 была показана общая структура системы Linux. Давайте теперь подробнее рассмотрим ядро системы (перед тем как начать изучение планирования процессов и файловой системы).

Ядро работает непосредственно с аппаратным обеспечением и обеспечивает взаимодействие с устройствами ввода-вывода и блоком управления памятью, а также управляет доступом процессора к ним. Нижний уровень ядра (рис.2) состоит из обработчиков прерываний (которые являются основным средством взаимодействия с устройствами) и механизма диспетчеризации на низком уровне. Диспетчеризация производится при возникновении прерывания. При этом код низкого уровня останавливает выполнение работающего процесса, сохраняет его состояние в структурах процессов ядра и запускает соответствующий драйвер. Диспетчеризация процессов производится также тогда, когда ядро завершает некую операцию и пора снова запустить процесс пользователя. Код диспетчеризации написан на ассемблере и представляет собой отдельную от процедуры планирования программу.

Затем мы разделили различные подсистемы ядра на три основных компонента. Компонент ввода-вывода на рис.2 содержит все те части ядра, которые отвечают за взаимодействие с устройствами, а также выполнение сетевых операций и операций ввода-вывода на внешние устройства. На самом высоком уровне все операции вводавывода интегрированы в уровень виртуальной файловой системы (Virtual File System (**VFS**)). То есть на самом верхнем уровне выполнение операции чтения из файла (будь он в памяти или на диске) — это то же самое, что и выполнение операции чтения символа с терминального ввода. На самом низком уровне все операции ввода-вывода проходят через какой-то драйвер устройства. Все драйверы в Linux классифицируются либо как символьные драйверы устройств, либо как блочные драйверы устройств, причем основная разница состоит в том, что поиск и произвольный доступ разрешены только для блочных устройств.



**Рис. 2.** Структура ядра операционной системы Linux

В техническом смысле сетевые устройства — это символьные устройства, но работа с ними ведется несколько иначе, так что лучше их выделить (что и было сделано на рисунке).

Выше уровня драйверов устройств код ядра для каждого типа устройств свой. Символьные устройства могут использоваться двумя разными способами. Некоторым программам, таким как экранные редакторы vi и emacs, требуется каждая нажатая клавиша без какой-либо обработки. Для этого служит необработанный ввод-вывод с терминала (tty). Другое программное обеспечение (такое, как оболочки) принимает на входе уже готовую текстовую строку, позволяя пользователю редактировать ее, пока не будет нажата клавиша Enter для отправки строки в программу. В этом случае поток символов с устройства терминала передается через так называемую дисциплину линии связи (и применяется соответствующее форматирование).

Сетевое программное обеспечение часто бывает модульным, с поддержкой множества различных устройств и протоколов. Уровень выше сетевых драйверов выполняет своего рода функции маршрутизации, обеспечивая отправку правильного пакета к правильному устройству или блоку обработки протокола. Большинство систем Linux содержат в своем ядре полнофункциональный эквивалент аппаратного маршрутизатора (однако его производительность ниже, чем у аппаратного маршрутизатора). Над кодом маршрутизации располагается стек протоколов, который всегда включает протоколы IP и TCP, а также много других дополнительных протоколов. Над сетью располагается интерфейс сокетов, позволяющий программам создавать сокеты (для сетей и протоколов). Для последующего использования сокета возвращается дескриптор файла.

Над дисковыми драйверами располагается планировщик ввода-вывода, который отвечает за упорядочивание и выдачу запросов на дисковые операции таким способом, который экономит излишние перемещения головок (или реализует некую иную системную стратегию).

На самом верху столбца блочных устройств располагаются файловые системы. Linux имеет несколько одновременно сосуществующих файловых систем. Для того чтобы скрыть «страшные» архитектурные отличия аппаратных устройств от реализации файловой системы, уровень обобщенных блочных устройств обеспечивает используемую всеми файловыми системами абстракцию.

Справа на рис.2 находятся два других ключевых компонента ядра Linux. Они отвечают за задачи управления памятью и процессором. В задачи управления памятью входят обслуживание отображения виртуальной памяти на физическую, поддержка кэша страниц, к которым недавно выполнялось обращение (и хорошая стратегия замены страниц), поставка в память (по требованию) новых страниц с кодом и данными.

Основная область ответственности компонента управления процессами — это создание и завершение процессов. В нем имеется также планировщик процессов, который выбирает, какой процесс (вернее, поток) будет работать дальше. Как мы увидим в следующем разделе, ядро Linux работает с процессами и потоками как с исполняемыми модулями и планирует их на основе глобальной стратегии планирования. Код обработки сигналов также принадлежит этому компоненту.

Несмотря на то что эти три компонента представлены на рисунке отдельно, они сильно зависят друг от друга. Файловые системы обычно обращаются к файлам через блочные устройства. Однако для скрытия большой латентности дискового доступа файлы копируются в кэш страниц (находящийся в оперативной памяти). Некоторые файлы (такие, как файлы с информацией об использовании ресурсов времени выполнения) могут даже создаваться динамически и иметь представление только в оперативной памяти. Кроме того, система виртуальной памяти может использовать дисковый раздел или область подкачки в файле (для сохранения части оперативной памяти, когда ей нужно освободить определенные страницы), и поэтому она использует компонент ввода-вывода. Существует и множество других взаимозависимостей.

Помимо статических компонентов ядра Linux поддерживает и динамически загружаемые модули. Эти модули могут использоваться для добавления или замены драйверов устройств по умолчанию, файловых систем, сетевой работы, а также прочих кодов ядра. Эти модули на рис.2 не показаны.

Самый верхний уровень — это интерфейс системных вызовов ядра. Все системные вызовы поступают сюда и вызывают эмулированное прерывание, которое переключает исполнение из пользовательского режима в защищенный режим ядра и передает управление одному из описанных ранее компонентов ядра.

## 2. Процессы в системе Linux

В предыдущих разделах мы начали наш обзор системы Linux с точки зрения пользователя, сидящего за клавиатурой (то есть с того, что видит пользователь в окне xterm). Были приведены примеры часто использующихся команд оболочки и утилит. Этот краткий обзор был завершен рассмотрением структуры системы. Теперь настало время углубиться в ядро и более пристально рассмотреть основные концепции, поддерживаемые системой Linux, а именно: процессы, память, файловую систему и ввод-вывод. Эти понятия важны, так как ими управляют системные вызовы — интерфейс самой операционной системы. Например, существуют системные вызовы для создания процессов и потоков, выделения памяти, открытия файлов и выполнения ввода-вывода.

К сожалению, существует очень много версий системы Linux, и между ними имеются определенные различия. В данной главе основное внимание будет уделено общим свойствам всех версий, а не особенностям какой-либо одной версии. Таким образом, в определенных разделах (особенно в тех, где будет рассматриваться вопрос реализации) может оказаться, что описание не соответствует в равной мере всем версиям.

**Фундаментальные концепции.** Основными активными сущностями в системе Linux являются процессы. Процессы Linux очень похожи на классические последовательные процессы, которые мы изучали в главе 2. Каждый процесс выполняет одну программу и изначально получает один поток управления. Иначе говоря, у процесса есть один счетчик команд, который отслеживает следующую исполняемую команду. Linux позволяет процессу создавать дополнительные потоки (после того, как он начинает выполнение).

Linux представляет собой многозадачную систему и несколько независимых процессов могут работать одновременно. Более того, у каждого пользователя может быть одновременно несколько активных процессов, так что в большой системе могут одновременно работать сотни и даже тысячи процессов. Фактически на большинстве однопользовательских рабочих станций (даже когда пользователь куда-либо отлучился) работают десятки фоновых процессов, называемых **демонами** (daemons). Они запускаются при загрузке системы из сценария оболочки.

Типичным демоном является cron. Он просыпается раз в минуту, проверяя, не нужно ли ему что-то сделать. Если у него есть работа, он ее выполняет, а затем отправляется спать дальше (до следующей проверки).

Этот демон позволяет планировать в системе Linux активность на минуты, часы, дни и даже месяцы вперед. Например, представьте, что пользователю назначено явиться к зубному врачу в 15.00 в следующий вторник. Он может создать запись в базе данных демона cron, чтобы тот просигналил ему, скажем, в 14.30. Когда наступают назначенные день и время, демон cron видит, что у него есть работа, и в нужное время запускает программу звукового сигнала (в виде нового процесса).

Демон cron также используется для периодического запуска задач, например ежедневного резервного копирования диска в 4.00. Другие демоны управляют входящей и исходящей электронной почтой, очередями принтера, проверяют, достаточно ли еще осталось свободных страниц памяти, и т. д. Демоны реализуются в системе Linux довольно просто, так как каждый из них представляет собой отдельный процесс, не зависимый от всех остальных процессов.

Процессы создаются в операционной системе Linux чрезвычайно просто. Системный вызов *fork* создает точную копию исходного процесса, называемого **родительским процессом** (parent process). Новый процесс называется **дочерним процессом** (child process). У родительского и у дочернего процессов есть собственные (приватные) образы памяти. Если родительский процесс впоследствии изменяет какие-либо свои переменные, то эти изменения остаются невидимыми для дочернего процесса (и наоборот).

Открытые файлы используются родительским и дочерним процессами совместно. Это значит, что если какой-либо файл был открыт в родительском процессе до выполнения системного вызова *fork*, он останется открытым в обоих процессах и в дальнейшем. Изменения, произведенные с этим файлом любым из процессов, будут видны другому. Такое поведение является единственно разумным, так как эти изменения будут видны также любому другому процессу, который тоже откроет этот файл.

Тот факт, что образы памяти, переменные, регистры и все остальное у родительского и дочернего процессов идентичны, приводит к небольшому затруднению: как процессам узнать, какой из них должен исполнять родительский код, а какой — дочерний? Секрет в том, что системный вызов *fork* возвращает дочернему процессу число 0, а родительскому — отличный от нуля **PID** (Process IDentifier — идентификатор процесса) дочернего процесса. Оба процесса обычно проверяют возвращаемое значение и действуют так, как показано в листинге 1.

**Листинг 1.** Создание процесса в системе Linux

pid = fork( ); /\* если fork завершился успешно, pid > 0 в

родительском процессе \*/

if (pid < 0) {

handle\_error(); /\* fork потерпел неудачу (например, память

или какая-либо таблица переполнена) \*/

} else if (pid > 0) {

} else {

}

/\* здесь располагается родительский код \*/

/\* здесь располагается дочерний код \*/

Процессы именуются своими PID-идентификаторами. Как уже говорилось, при создании процесса его PID выдается родителю нового процесса. Если дочерний процесс желает узнать свой PID, то он может воспользоваться системным вызовом *getpid*. Идентификаторы процессов используются различным образом. Например, когда дочерний процесс завершается, его родитель получает PID только что завершившегося дочернего процесса. Это может быть важно, так как у родительского процесса может быть много дочерних процессов. Поскольку у дочерних процессов также могут быть дочерние процессы, то исходный процесс может создать целое дерево детей, внуков, правнуков и более дальних потомков.

В системе Linux процессы могут общаться друг с другом с помощью некой формы передачи сообщений. Можно создать канал между двумя процессами, в который один процесс сможет писать поток байтов, а другой процесс сможет его читать. Эти каналы иногда называют **трубами** (pipes). Синхронизация процессов достигается путем блокирования процесса при попытке прочитать данные из пустого канала. Когда данные появляются в канале, процесс разблокируется.

При помощи каналов организуются конвейеры оболочки. Когда оболочка видит строку вроде

sort <f | head

она создает два процесса, *sort* и *head*, а также устанавливает между ними канал таким образом, что стандартный поток вывода программы *sort* соединяется со стандартным потоком ввода программы *head*. При этом все данные, которые пишет *sort*, попадают напрямую к *head*, для чего не требуется временного файла. Если канал переполняется, то система приостанавливает работу *sort* до тех пор, пока *head* не удалит из него хоть сколько-нибудь данных.

**Таблица 2.** Сигналы, требуемые стандартом POSIX

|  |  |
| --- | --- |
| **Сигнал** | **Причина** |
| SIGABRT | Посылается, чтобы прервать процесс и создать дамп памяти |
| SIGALRM | Истекло время будильника |
| SIGFPE | Произошла ошибка при выполнении операции с плавающей точкой (например, деление на 0) |
| SIGHUP | На телефонной линии, использовавшейся процессом, была повешена трубка |
| SIGILL | Пользователь нажал клавишу Del, чтобы прервать процесс |
| SIGQUIT | Пользователь нажал клавишу, требующую выполнения дампа памяти |
| SIGKILL | Посылается, чтобы уничтожить процесс (не может игнорироваться или перехватываться) |
| SIGPIPE | Процесс пишет в канал, из которого никто не читает |
| SIGSEGV | Процесс обратился к неверному адресу памяти |
| SIGTERM | Вежливая просьба к процессу завершить свою работу |
| SIGUSR1 | Может быть определен приложением |
| SIGUSR2 | Может быть определен приложением |

Процессы могут общаться и другим способом — при помощи программных прерываний. Один процесс может послать другому так называемый **сигнал** (signal). Процессы могут сообщить системе, какие действия следует предпринимать, когда придет входящий сигнал. Варианты такие: проигнорировать сигнал, перехватить его, позволить сигналу убить процесс (действие по умолчанию для большинства сигналов). Если процесс выбрал перехват посылаемых ему сигналов, он должен указать процедуру обработки сигналов. Когда сигнал прибывает, управление сразу же передается обработчику. Когда процедура обработки сигнала завершает свою работу, управление снова передается в то место, в котором оно находилось, когда пришел сигнал (это аналогично обработке аппаратных прерываний ввода-вывода). Процесс может посылать сигналы только членам своей **группы процессов** (process group), состоящей из его прямого родителя (и других предков), братьев и сестер, а также детей (и прочих потомков). Процесс может также послать сигнал сразу всей своей группе за один системный вызов.

Сигналы используются и для других целей. Например, если процесс выполняет вычисления с плавающей точкой и непреднамеренно делит на 0 (делает то, что осуждается математиками), то он получает сигнал SIGFPE (Floating-Point Exception SIGnal — сигнал исключения при выполнении операции с плавающей точкой). Сигналы, требуемые стандартом POSIX, перечислены в табл.2. В большинстве систем Linux имеются также дополнительные сигналы, но использующие их программы могут оказаться непереносимыми на другие версии Linux и UNIX.

### Системные вызовы управления процессами в Linux. Рассмотрим теперь системные вызовы Linux, предназначенные для управления процессами. Основные системные вызовы перечислены в табл.3. Обсуждение проще всего начать с системного вызова *fork*. Этот системный вызов (поддерживаемый также в традиционных системах UNIX) представляет собой основной способ создания новых процессов в системах Linux (другой способ мы обсудим в следующем разделе). Он создает точную копию оригинального процесса, включая все описатели файлов, регистры и пр. После выполнения системного вызова *fork* исходный процесс и его копия (родительский и дочерний процессы) идут каждый своим путем. Сразу после выполнения системного вызова *fork* значения всех соответствующих переменных в обоих процессах одинаковы, но после копирования всего адресного пространства родителя (для создания потомка) последующие изменения в одном процессе не влияют на другой процесс. Системный вызов *fork* возвращает значение, равное нулю, для дочернего процесса и значение, равное идентификатору (PID) дочернего процесса, — для родительского. По этому идентификатору оба процесса могут определить, кто из них родитель, а кто — потомок.

### Код возврата s в случае ошибки равен –1; pid — это идентификатор процесса; residual — остаток времени от предыдущего сигнала. Смысл параметров понятен по их названиям.

**Таблица 3.** Некоторые системные вызовы, относящиеся к процессам

|  |  |
| --- | --- |
| **Системный вызов** | **Описание** |
| pid=fork( ) | Создать дочерний процесс, идентичный родительскому |
| pid=waitpid(pid, &statloc, opts) | Ждать завершения дочернего процесса |
| s=execve(name, argv, envp) | Заменить образ памяти процесса |
| exit(status) | Завершить выполнение процесса и вернуть статус |
| s=sigaction(sig, &act, &oldact) | Определить действие, выполняемое при приходе сигнала |
| s=sigreturn(&context) | Вернуть управление после обработки сигнала |
| s=sigprocmask(how, &set, &old) | Исследовать или изменить маску сигнала |
| s=sigpending(set) | Получить набор блокированных сигналов |
| s=sigsuspend(sigmask) | Заменить маску сигнала и приостановить процесс |
| s=kill(pid, sig) | Послать сигнал процессу |
| residual=alarm(seconds) | Установить будильник |
| s=pause( ) | Приостановить выполнение вызывающей стороны до следующего сигнала |

В большинстве случаев после системного вызова *fork* дочернему процессу требуется выполнить отличающийся от родительского процесса код. Рассмотрим работу оболочки. Она считывает команду с терминала, с помощью системного вызова *fork* создает дочерний процесс, ждет выполнения введенной команды дочерним процессом, после чего считывает следующую команду (после завершения дочернего процесса). Для ожидания завершения дочернего процесса родительский процесс делает системный вызов *waitpid*, который ждет завершения потомка (любого потомка, если их несколько). У этого системного вызова три параметра. Первый параметр позволяет вызывающей стороне ждать конкретного потомка. Если этот параметр равен –1, то в этом случае системный вызов ожидает завершения любого дочернего процесса. Второй параметр представляет собой адрес переменной, в которую записывается статус завершения дочернего процесса (нормальное или ненормальное завершение, а также возвращаемое на выходе значение). Это позволяет родителю знать о судьбе своего ребенка. Третий параметр определяет, будет ли вызывающая сторона блокирована или сразу получит управление обратно (если ни один потомок не завершен).

В случае использования оболочки дочерний процесс должен выполнить введенную пользователем команду. Он делает это при помощи системного вызова *exec*, который заменяет весь образ памяти содержимым файла, указанного в первом параметре. Крайне упрощенный вариант оболочки, иллюстрирующей использование системных вызовов *fork*, *waitpid* и *exec*, показан в листинге 2.

В самом общем случае у системного вызова *exec* три параметра: имя исполняемого файла, указатель на массив аргументов и указатель на массив строк окружения. Скоро мы все это опишем. Различные библиотечные процедуры, такие как *execl*, *execv*, *execle* и *execve*, позволяют опускать некоторые параметры или указывать их иными способами. Все эти процедуры обращаются к одному и тому же системному вызову. Хотя сам системный вызов называется *exec*, библиотечной процедуры с таким именем нет (необходимо использовать одну из вышеупомянутых).

**Листинг 2.** Сильно упрощенная оболочка

while (TRUE) { /\* бесконечный цикл \*/

type\_prompt( ); /\* вывести приглашение к вводу \*/ read\_command(command, params); /\* прочитать с клавиатуры строку ввода\*/ pid = fork( ); /\* ответвить дочерний процесс \*/

if (pid < 0) {

printf("Создать процесс невозможно"); /\* ошибка \*/ continue; /\* повторить цикл \*/

}

if (pid != 0) {

waitpid (-1, &status, 0); /\* родительский процесс ждет дочерний

процесс \*/

} else {

execve(command, params, 0); /\* дочерний процесс выполняет работу \*/

}

}

Рассмотрим случай выполнения оболочкой команды

cp file1 file2

используемой для копирования файла file1 в файл file2. После того как оболочка создает дочерний процесс, тот обнаруживает и исполняет файл cp и передает ему информацию о копируемых файлах.

Главная программа файла cp (как и многие другие программы) содержит объявление функции

main(argc, argv, envp)

где *argc* — счетчик количества элементов командной строки, включая имя программы. Для приведенного примера значение *argc* равно 3.

Второй параметр *argv* представляет собой указатель на массив. *i*-й элемент этого массива является указателем на *i*-й элемент командной строки. В нашем примере элемент *argv[0]* указывает на двухсимвольную строку «cp». Соответственно элемент *argv[1]* указывает на пятисимвольную строку «file1», а элемент *argv[2]* — на пятисимвольную строку «file2».

Третий параметр *envp* процедуры *main* представляет собой указатель на среду (массив, содержащий строки вида *имя* = *значение*, используемые для передачи программе такой информации, как тип терминала и имя домашнего каталога). В листинге 2 дочернему процессу переменные среды не передаются, поэтому третий параметр *execve* в данном случае равен нулю.

Если системный вызов *exec* показался вам слишком мудреным, не отчаивайтесь — это самый сложный системный вызов. Все остальные значительно проще. В качестве примера простого системного вызова рассмотрим *exit*, который процессы должны использовать при завершении исполнения. У него есть один параметр — статус выхода (от 0 до 255), возвращаемый родительскому процессу в переменной *status* системного вызова *waitpid*. Младший байт переменной *status* содержит статус завершения (равный нулю при нормальном завершении или коду ошибки — при аварийном). Старший байт содержит статус выхода потомка (от 0 до 255), указанный в вызове завершения потомка. Например, если родительский процесс выполняет оператор

n = waitpid(-1, &status, 0);

то он будет приостановлен до тех пор, пока не завершится какой-либо дочерний процесс. Если, например, дочерний процесс завершится со значением статуса 4 (в параметре библиотечной процедуры *exit*), то родительский процесс будет разбужен со значением *n*, равным PID дочернего процесса, и значением статуса 0x0400 (префикс 0x означает в программах на языке C шестнадцатеричное число). Младший байт переменной *status* относится к сигналам, старший байт представляет собой значение, задаваемое дочерним процессом в виде параметра при обращении к системному вызову *exit*.

Если процесс уже завершился, а родительский процесс не ожидает этого события, то дочерний процесс переводится в так называемое **состояние зомби** (zombie state) — живого мертвеца, то есть приостанавливается. Когда родительский процесс, наконец, обращается к библиотечной процедуре *waitpid*, дочерний процесс завершается.

Некоторые системные вызовы относятся к сигналам, используемым различными способами. Допустим, если пользователь случайно дал текстовому редактору указание отобразить содержимое очень длинного файла, а затем осознал свою ошибку, то ему потребуется некий способ прервать работу редактора. Обычно для этого пользователь нажимает специальную клавишу (например, Del или Ctrl+C), в результате чего редактору посылается сигнал. Редактор перехватывает сигнал и останавливает вывод.

Чтобы заявить о своем желании перехватить тот или иной сигнал, процесс может воспользоваться системным вызовом *sigaction*. Первый параметр этого системного вызова — сигнал, который требуется перехватить (см. табл.2). Второй параметр представляет собой указатель на структуру, в которой хранится указатель на процедуру обработки сигнала (вместе с различными прочими битами и флагами). Третий параметр указывает на структуру, в которую система возвращает информацию о текущей обработке сигналов на случай, если позднее его нужно будет восстановить.

Обработчик сигнала может выполняться сколь угодно долго. Однако на практике обработка сигналов занимает очень мало времени. Когда процедура обработки сигнала завершает свою работу, она возвращается к той точке, из которой ее прервали.

Системный вызов *sigaction* может использоваться также для игнорирования сигнала или чтобы восстановить действие по умолчанию, заключающееся в уничтожении процесса.

Нажатие на клавишу Del не является единственным способом послать сигнал. Системный вызов *kill* позволяет процессу послать сигнал другому родственному процессу. Выбор названия для данного системного вызова (kill — убить, уничтожить) не особенно удачен, так как чаще всего сигналы посылаются процессами для того, чтобы быть перехваченными. А вот неперехваченный сигнал, конечно же, убьет получателя.

Во многих приложениях реального времени бывает необходимо прервать процесс через определенный интервал времени, чтобы что-то сделать, например передать повторно потерянный пакет по ненадежной линии связи. Для обработки данной ситуации имеется системный вызов *alarm* (будильник). Параметр этого системного вызова задает временной интервал в секундах, по истечении которого процессу посылается сигнал SIGALRM. У процесса в каждый момент времени может быть только один будильник. Например, если делается системный вызов *alarm* с параметром 10 с, а через 3 с снова делается вызов *alarm* с параметром 20 с, то будет сгенерирован только один сигнал — через 20 с после второго вызова. Первый сигнал будет отменен вторым обращением к вызову *alarm*. Если параметр системного вызова *alarm* равен нулю, то такое обращение отменяет любой невыполненный сигнал *alarm*. Если сигнал *alarm* не перехватывается, то выполняется действие по умолчанию — и процесс уничтожается. Технически возможно игнорирование данного сигнала, но смысла это не имеет. Зачем программе просить отправить сигнал чуть позже, а затем его игнорировать?

Иногда случается так, что процессу нечем заняться, пока не придет сигнал. Например, рассмотрим обучающую программу, проверяющую скорость чтения и понимание текста. Она отображает на экране некий текст, а затем делает системный вызов *alarm*, чтобы система послала ей сигнал через 30 с. Пока студент читает текст, программе делать нечего. Она может находиться (ничего не делая) в коротком цикле, но будет напрасно расходовать время центрального процессора, которое может понадобиться фоновому процессу или другому пользователю. Лучшее решение заключается в использовании системного вызова *pause*, который дает указание операционной системе Linux приостановить работу процесса до появления следующего сигнала. И горе той программе, которая останавливается без выставления сигнала будильника.

### Реализация процессов и потоков в Linux. Процесс в Linux подобен айсбергу: то, что вы видите, представляет собой всего лишь выступающую над водой его часть, но не менее важная часть скрыта под водой. У каждого процесса есть пользовательская часть, в которой работает программа пользователя. Однако когда один из потоков делает системный вызов, то происходит эмулированное прерывание с переключением в режим ядра. После этого поток начинает работу в контексте ядра с другой картой памяти и полным доступом ко всем ресурсам машины. Это все тот же самый поток, но теперь обладающий большей властью, а также со своим стеком ядра и счетчиком команд в режиме ядра. Это важно, так как системный вызов может блокироваться на полпути: например, в ожидании завершения дисковой операции. При этом счетчик команд и регистры будут сохранены таким образом, чтобы позднее поток можно было перезапустить в режиме ядра.

Ядро Linux внутренним образом представляет процессы как **задачи** (tasks) при помощи структуры задач task\_struct. В отличие от подходов других операционных систем (которые делают различия между процессом, легковесным процессом и потоком), Linux использует структуру задач для представления любого контекста исполнения. Поэтому процесс с одним потоком представляется одной структурой задач, а многопоточный процесс будет иметь по одной структуре задач для каждого из потоков пользовательского уровня. Наконец, само ядро является многопоточным и имеет потоки уровня ядра, которые не связаны ни с какими пользовательскими процессами и выполняют код ядра. Мы вернемся к обработке многопоточных процессов (и потоков вообще) далее в этом же разделе.

Для каждого процесса в памяти всегда находится его дескриптор типа task\_struct. Он содержит важную информацию, необходимую ядру для управления всеми процессами (в том числе параметры планирования, списки дескрипторов открытых файлов и т. д.). Дескриптор процесса (вместе с памятью стека режима ядра для процесса) создается при создании процесса.

Для совместимости с другими системами UNIX процессы в Linux идентифицируются при помощи идентификатора процесса (Process Identifier (PID)). Ядро организует все процессы в двунаправленный список структур задач. В дополнение к доступу к дескрипторам процессов при помощи перемещения по связанным спискам PID можно отобразить на адрес структуры задач и немедленно получить доступ к информации процесса.

Структура задачи содержит множество полей. Некоторые из этих полей содержат указатели на другие структуры данных или сегменты (например, содержащие информацию об открытых файлах). Некоторые из этих сегментов относятся к структуре процесса для пользовательского уровня (которая не представляет никакого интереса, если пользовательский процесс не выполняется). Поэтому они могут быть вытеснены в файл подкачки (чтобы не расходовать память на ненужную информацию). Например, несмотря на то что процессу может быть послан сигнал в то время, когда он вытеснен, он не может читать файл. По этой причине информация о сигналах всегда должна находиться в памяти — даже когда процесса в памяти нет. В то же время информация о дескрипторах файлов может храниться в пользовательской структуре и доставляться только тогда, когда процесс находится в памяти и может выполняться.

Информация в дескрипторе процесса подразделяется на следующие категории:

* + - 1. **Параметры планирования.** Приоритет процесса, израсходованное за последний учитываемый период процессорное время, количество проведенного в режиме ожидания времени. Вся эта информация используется для выбора процесса, который будет выполняться следующим.
      2. **Образ памяти.** Указатели на сегменты: текста, данных и стека или на таблицы страниц. Если сегмент текста используется совместно, то указатель текста указывает на общую таблицу текста. Когда процесса нет в памяти, то здесь также содержится информация о том, как найти части процесса на диске.
      3. **Сигналы.** Маски, указывающие, какие сигналы игнорируются, какие перехватываются, какие временно заблокированы, а какие находятся в процессе доставки.
      4. **Машинные регистры.** Когда происходит эмулированное прерывание в ядро, то машинные регистры (включая регистры с плавающей точкой) сохраняются здесь.
      5. **Состояние системного вызова.** Информация о текущем системном вызове (включая параметры и результаты).
      6. **Таблица дескрипторов файлов.** Когда делается системный вызов, использующий дескриптор файла, то файловый дескриптор используется как индекс в этой таблице для обнаружения соответствующей этому файлу структуры данных (i-node).
      7. **Учетные данные.** Указатель на таблицу, в которой отслеживается использованное процессом пользовательское и системное время процессора. Некоторые системы также хранят здесь предельные значения времени процессора, которое может использовать процесс, максимальный размер его стека, количество блоков страниц, которое он может использовать, и пр.
      8. **Стек ядра.** Фиксированный стек для использования той частью процесса, которая работает в режиме ядра.
      9. **Разное.** Текущее состояние процесса, ожидаемые процессом события (если таковые есть), время до истечения интервала будильника, PID процесса, PID родительского процесса, идентификаторы пользователя и группы.

Зная все это, легко объяснить, как в системе Linux создаются процессы. Механизм создания нового процесса довольно прост. Для дочернего процесса создаются новый дескриптор процесса и пользовательская область, которая заполняется в большей степени из родительского процесса. Дочерний процесс получает PID, затем настраивается его карта памяти. Кроме того, дочернему процессу предоставляется совместный доступ к файлам родительского процесса. Затем настраиваются регистры дочернего процесса, после чего он готов к запуску.

Когда выполняется системный вызов *fork*, вызывающий процесс выполняет эмулированное прерывание в ядро и создает структуру задач и несколько других сопутствующих структур данных (таких, как стек режима ядра и структура thread\_info). Эта структура выделяется на фиксированном смещении от конца стека процесса и содержит несколько параметров процесса (вместе с адресом дескриптора процесса). Поскольку дескриптор процесса хранится в определенном месте, системе Linux нужно всего несколько эффективных операций, чтобы найти структуру задачи для выполняющегося процесса.

Большая часть содержимого дескриптора процесса заполняется значениями из дескриптора родителя. Затем Linux ищет доступный PID, который в этот момент не используется любыми другими процессами, и обновляет элемент хэш-таблицы PID, чтобы там был указатель на новую структуру задачи. В случае конфликтов в хэштаблице дескрипторы процессов могут быть сцеплены. Она также настраивает поля в task\_struct, чтобы они указывали на соответствующий предыдущий/следующий процесс в массиве задач.

В принципе, теперь следует выделить память для данных потомка и сегментов стека и сделать точные копии сегментов родителя, поскольку семантика системного вызова *fork* говорит о том, что никакая область памяти не используется совместно родительским и дочерним процессами. Текстовый сегмент может либо копироваться, либо использоваться совместно (поскольку он доступен только для чтения). В этот момент дочерний процесс готов работать.

Однако копирование памяти является дорогим удовольствием, поэтому все современные Linux-системы слегка жульничают. Они выделяют дочернему процессу его собственные таблицы страниц, но эти таблицы указывают на страницы родительского процесса, помеченные как доступные только для чтения. Когда какой-либо процесс (дочерний или родительский) пытается писать в такую страницу, происходит нарушение защиты. Ядро видит это и выделяет процессу, нарушившему защиту, новую копию этой страницы, которую помечает как доступную для чтения и записи. Таким образом, копируются только те страницы, в которые дочерний процесс пишет. Такой механизм называется **копированием при записи** (copy on write). При этом дополнительно экономится память, так как страницы с программой не копируются.

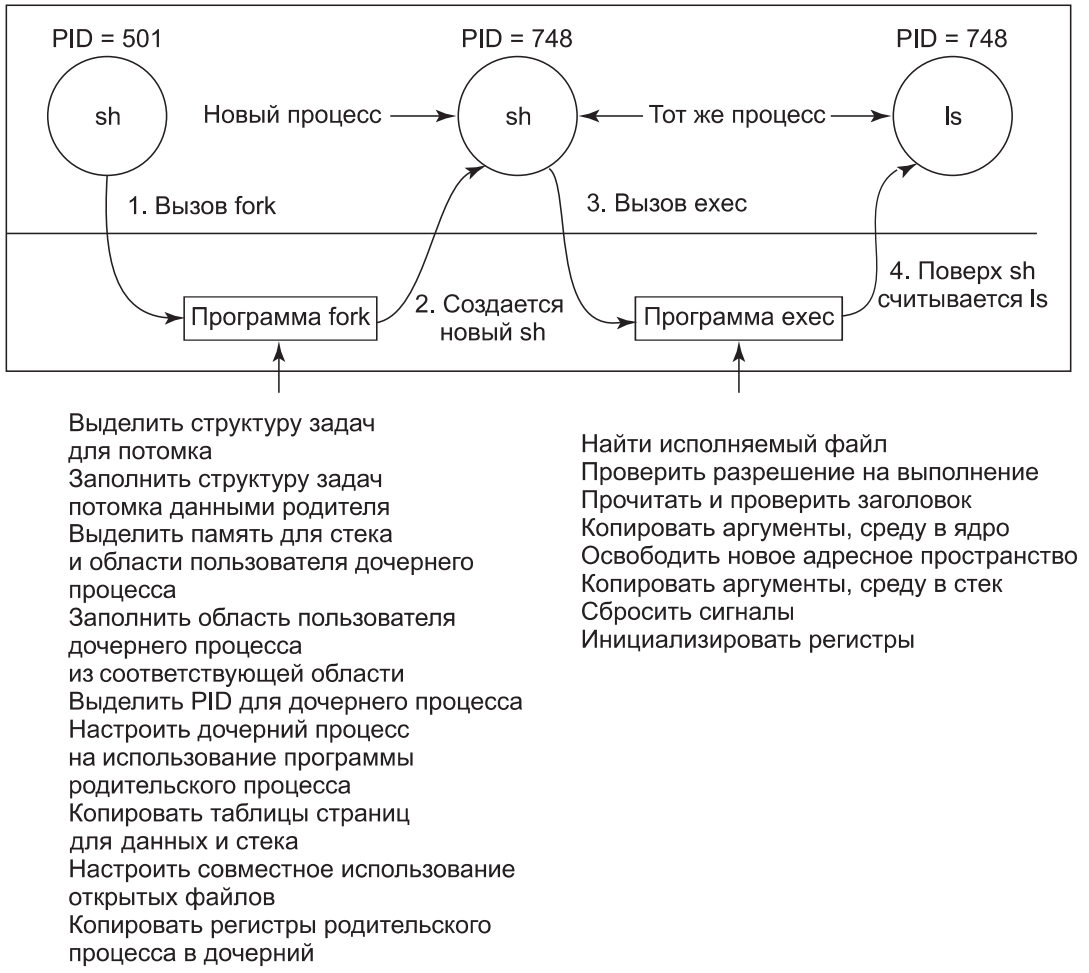
После того как дочерний процесс начинает работу, его код (в нашем примере это копия оболочки) делает системный вызов *exec*, задавая имя команды в качестве параметра. При этом ядро находит и проверяет исполняемый файл, копирует в ядро аргументы и строки окружения, а также освобождает старое адресное пространство и его таблицы страниц.

Теперь надо создать и заполнить новое адресное пространство. Если системой поддерживается отображение файлов на адресное пространство памяти (как это делается, например, в Linux и практически во всех остальных системах на основе UNIX), то новые таблицы страниц настраиваются следующим образом: в них указывается, что страниц в памяти нет (кроме, возможно, одной страницы со стеком), а содержимое адресного пространства зарезервировано исполняемым файлом на диске. Когда новый процесс начинает работу, он немедленно вызывает страничную ошибку, в результате которой первая страница кода подгружается из исполняемого файла. Таким образом, ничего не нужно загружать заранее, что позволяет быстро запускать программы, а в память загружать только те страницы, которые действительно нужны программам. (Эта стратегия фактически является подкачкой по требованию в ее самом чистом виде — см. главу 3.) Наконец, в новый стек копируются аргументы и строки окружения, сигналы сбрасываются, а все регистры устанавливаются в нуль. С этого момента новая команда может начинать исполнение.

Описанные шаги показаны на рис.3 при помощи следующего примера: пользователь вводит с терминала команду *ls*, оболочка создает новый процесс, клонируя себя с помощью системного вызова *fork*. Новая оболочка затем делает системный вызов *exec*, чтобы считать в свою область памяти содержимое исполняемого файла ls. После этого можно приступать к выполнению ls.

#### Потоки в Linux. Потоки в общих чертах обсуждались в главе 2. Здесь мы сосредоточимся на потоках ядра в Linux, и в особенности на различиях модели потоков в Linux и других UNIXсистемах. Чтобы лучше понять предоставляемые моделью Linux уникальные возможности, начнем с обсуждения некоторых трудных решений, присутствующих в многопоточных системах.

При знакомстве с потоками основная проблема заключается в выдерживании корректной традиционной семантики UNIX. Рассмотрим сначала системный вызов *fork*. Предположим, что процесс с несколькими (реализуемыми в ядре) потоками делает системный вызов *fork*. Следует ли в новом процессе создавать все остальные потоки? Предположим, что мы ответили на этот вопрос утвердительно. Допустим также, что один из остальных потоков был блокирован (в ожидании ввода с клавиатуры). Должен ли поток в новом процессе также быть блокирован ожиданием ввода с клавиатуры? Если да, то какому потоку достанется следующая набранная на клавиатуре строка? Если нет, то что должен делать этот поток в новом процессе?



**Рис.3.** Шаги выполнения команды ls, введенной в оболочке

Эта проблема касается и многих других аспектов. В однопоточном процессе такой проблемы не возникает, так как единственный поток не может быть блокирован при вызове *fork*. Теперь рассмотрим случай, при котором в дочернем процессе остальные потоки не создаются. Предположим, что один из несозданных потоков удерживает мьютекс, который пытается получить единственный созданный поток нового процесса (после выполнения вызова *fork)*. В этом случае мьютекс никогда не будет освобожден и новый поток повиснет навсегда. Существует также множество других проблем. И простого решения нет.

Файловый ввод-вывод представляет собой еще одну проблемную область. Предположим, что один поток блокирован при чтении из файла, а другой поток закрывает файл или делает системный вызов *lseek*, чтобы изменить текущий указатель файла. Что произойдет дальше? Кто знает?

Обработка сигналов тоже представляет собой сложный вопрос. Должны ли сигналы направляться определенному потоку или всему процессу в целом? Вероятно, сигнал SIGFPE (Floating-Point Exception SIGnal — сигнал исключения при выполнении операции с плавающей точкой) должен перехватываться тем потоком, который его вызвал. А что, если он его не перехватывает? Следует ли убить этот поток? Или следует убить все потоки? Рассмотрим теперь сигнал SIGINT, генерируемый сидящим за клавиатурой пользователем. Какой поток должен перехватывать этот сигнал? Должен ли у всех потоков быть общий набор масок сигналов? При решении подобных проблем любые попытки вытянуть нос в одном месте приводят к тому, что в каком-либо другом месте увязает хвост. Корректная реализация семантики потоков (не говоря уже о коде) представляет собой нетривиальную задачу.

Операционная система Linux поддерживает потоки в ядре довольно интересным способом, с которым стоит познакомиться. Эта реализация основана на идеях из системы 4.4BSD, но в дистрибутиве 4.4BSD потоки на уровне ядра реализованы не были, так как у Калифорнийского университета в Беркли деньги кончились раньше, чем библиотеки языка C могли быть переписаны так, чтобы решить описанные ранее проблемы.

Исторически процессы были контейнерами ресурсов, а потоки — единицами исполнения. Процесс содержал один или несколько потоков, которые совместно использовали адресное пространство, открытые файлы, обработчики сигналов и все остальное. Все было понятно и просто.

В 2000 году в Linux был введен новый мощный системный вызов *clone*, который размыл различия между процессами и потоками и, возможно, даже инвертировал первенство этих двух концепций. Вызова *clone* нет ни в одной из версий UNIX. Классически при создании нового потока исходный поток (потоки) и новый поток совместно использовали все, кроме регистров, — в частности, дескрипторы для открытых файлов, обработчики сигналов, прочие глобальные свойства — все это было у каждого процесса, а не у потока. Системный вызов *clone* дал возможность все эти аспекты сделать характерными как для процесса, так и для потока. Формат вызова выглядит следующим образом:

pid = clone(function, stack\_ptr, sharing\_flags, arg);

Вызов *clone* создает новый поток либо в текущем, либо в новом процессе (в зависимости от флага *sharing\_flags*). Если новый поток находится в текущем процессе, то он совместно с существующими потоками использует адресное пространство и каждая последующая запись в любой байт адресного пространства (любым потоком) тут же становится видима всем остальным потокам процесса. Если же адресное пространство совместно не используется, то новый поток получает точную копию адресного пространства, но последующие записи из нового потока уже не видны старым потокам. Здесь используется та же семантика, что и у системного вызова *fork* по стандарту POSIX.

В обоих случаях новый поток начинает выполнение функции *function* с аргументом *arg* в качестве единственного параметра. Также в обоих случаях новый поток получает собственный приватный стек, при этом указатель стека инициализируется параметром *stack\_ptr*.

Параметр *sharing\_flags* представляет собой битовый массив, обеспечивающий существенно более тонкую настройку совместного использования, чем традиционные системы UNIX. Каждый бит может быть установлен независимо от остальных, и каждый из них определяет, копирует новый поток эту структуру данных или использует ее совместно с вызывающим потоком. В табл.4 показаны некоторые элементы, которые можно использовать совместно или копировать — в соответствии со значением битов в *sharing\_flags*.

Бит *CLONE\_VM* определяет, будет виртуальная память (то есть адресное пространство) использоваться совместно со старыми потоками или будет копироваться. Если этот бит установлен, то новый поток просто добавляется к старым потокам, так что в результате системный вызов *clone* создает новый поток в существующем процессе. Если этот бит сброшен, то новый поток получает собственное приватное адресное пространство. Это означает, что результат выданных из него команд процессора *STORE* не виден существующим потокам. Такое поведение подобно поведению системного вызова *fork*. Создание нового адресного пространства равнозначно определению нового процесса.

**Таблица 4.** Биты массива sharing\_flags

|  |  |  |
| --- | --- | --- |
| **Флаг** | **Значение при установке в 1** | **Значение при установке в 0** |
| CLONE\_VM | Создать новый поток | Создать новый процесс |
| CLONE\_FS | Общие рабочий каталог, каталог root и umask | Не использовать их совместно |
| CLONE\_FILES | Общие дескрипторы файлов | Копировать дескрипторы файлов |
| CLONE\_SIGHAND | Общая таблица обработчика сигналов | Копировать таблицу |
| CLONE\_PARENT | Новый поток имеет того же родителя, что и вызывающий | Родителем нового потока является вызывающий |

Бит *CLONE\_FS* управляет совместным использованием рабочего каталога и каталога root, а также флага *umask*. Даже если у нового потока есть собственное адресное пространство, при установленном бите *CLONE\_FS* старый и новый потоки будут совместно использовать рабочие каталоги. Это означает, что вызов *chdir* одним из потоков изменит рабочий каталог другого потока, несмотря на то что у другого потока есть собственное адресное пространство. В системе UNIX вызов *chdir* потоком всегда изменяет рабочий каталог всех остальных потоков этого процесса, но никогда не меняет рабочих каталогов других процессов. Таким образом, этот бит обеспечивает такую разновидность совместного использования, которая недоступна в традиционных версиях UNIX.

Бит *CLONE\_FILES* аналогичен биту *CLONE\_FS*. Если он установлен, то новый поток предоставляет свои дескрипторы файлов старым потокам, так что вызовы *lseek* одним потоком становятся видимыми для других потоков, что также обычно справедливо для потоков одного процесса, но не потоков различных процессов. Аналогично бит *CLONE\_SIGHAND* разрешает или запрещает совместное использование таблицы обработчиков сигналов старым и новым потоками. Если таблица общая (даже для потоков в различных адресных пространствах), то изменение обработчика в одном потоке повлияет на обработчики в других потоках.

И наконец, каждый процесс имеет родителя. Бит *CLONE\_PARENT* управляет тем, кто является родителем нового потока. Это может быть родитель вызывающего потока (в таком случае новый поток является братом вызывающего потока) либо сам вызывающий поток (в таком случае новый поток является потомком вызывающего). Есть еще несколько битов, которые управляют другими вещами, но они не так важны.

Такая детализация вопросов совместного использования стала возможна благодаря тому, что в системе Linux для различных элементов, перечисленных в начале раздела

«Реализация процессов и потоков в Linux» (параметры планирования, образ памяти и т. д.), используются отдельные структуры данных. Структура задач просто содержит указатели на эти структуры данных, поэтому легко создать новую структуру задач для каждого клонированного потока и сделать так, чтобы она указывала либо на структуры (планирования потоков, памяти и пр.) старого потока, либо на копии этих структур. Сам факт возможности такой высокой степени детализации совместного использования еще не означает, что она полезна, особенно потому что в традиционных версиях UNIX это не поддерживается. Если какая-либо программа в системе Linux пользуется этой возможностью, то это означает, что она больше не является переносимой на UNIX.

Модель потоков Linux порождает еще одну трудность. UNIX-системы связывают с процессом один PID (независимо от того, однопоточный он или многопоточный). Чтобы сохранять совместимость с другими UNIX-системами, Linux различает идентификаторы процесса PID и идентификаторы задачи TID. Оба этих поля хранятся в структуре задач. Когда вызов *clone* используется для создания нового процесса (который ничего не использует совместно со своим создателем), PID устанавливается в новое значение, в противном случае задача получает новый TID, но наследует PID. Таким образом, все потоки процесса получат тот же самый PID, что и первый поток процесса.

### Планирование в Linux. Теперь мы рассмотрим алгоритм планирования системы Linux. Начнем с того, что потоки в системе Linux реализованы в ядре, поэтому планирование основано на потоках, а не на процессах.

В операционной системе Linux алгоритмом планирования различаются три класса потоков:

* + - 1. Потоки реального времени, обслуживаемые по алгоритму FIFO (First in First Out — первым прибыл, первым обслужен).
      2. Потоки реального времени, обслуживаемые в порядке циклической очереди.
      3. Потоки разделения времени.

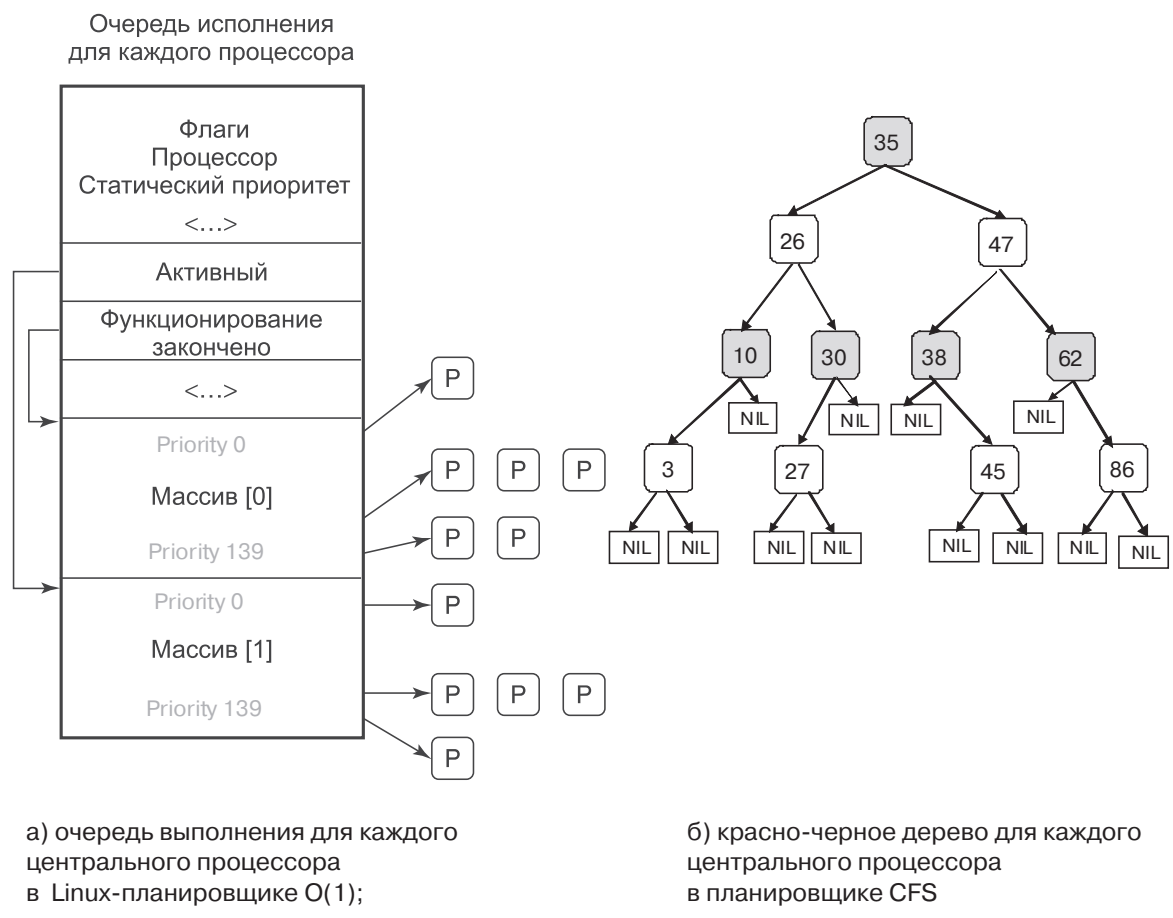
Обслуживаемые по алгоритму FIFO потоки реального времени имеют наивысшие приоритеты и не могут вытесняться другими потоками, за исключением такого же потока реального времени FIFO с более высоким приоритетом, перешедшего в состояние готовности. Обслуживаемые в порядке циклической очереди потоки реального времени представляют собой то же самое, что и потоки реального времени FIFO, но с тем отличием, что они имеют квант времени и могут вытесняться по таймеру. Такой находящийся в состоянии готовности поток выполняется в течение кванта времени, после чего помещается в конец своей очереди. Ни один из этих классов не является на самом деле классом реального времени. Здесь нельзя задать предельный срок выполнения задания и гарантировать его выполнение. Эти классы просто имеют более высокий приоритет, чем потоки стандартного класса разделения времени. Причина, по которой в системе Linux эти классы называются классами реального времени, состоит в том, что операционная система Linux соответствует стандарту POSIX 1003.4 (расширения реального времени для UNIX), в котором используются такие имена. Потоки реального времени внутри представлены приоритетами от 0 до 99, причем 0 — самый высокий, а 99 — самый низкий приоритет реального времени.

Обычные потоки составляют отдельный класс и планируются по особому алгоритму, поэтому с потоками реального времени они не конкурируют. Внутри системы этим потокам ставится в соответствие уровень приоритета от 100 до 139, то есть внутри себя Linux различает 140 уровней приоритета (для потоков реального времени и обычных). Так же как и обслуживаемым в порядке циклической очереди потокам реального времени, Linux выделяет обычным потокам время центрального процессора в соответствии с их требованиями и уровнями приоритета. В Linux время измеряется количеством тиков. В старых версиях Linux таймер работал на частоте 1000 Гц и каждый тик составлял 1 мс — этот интервал называют «**джиффи**» (jiffy — мгновение, миг, момент). В самых последних версиях частота тиков может настраиваться на 500, 250 или даже 1 Гц. Чтобы избежать нерационального расхода циклов центрального процессора на обслуживание прерываний от таймера, ядро может быть даже сконфигурировано в режиме «без тиков», что может пригодиться при запуске в системе только одного процесса или простое центрального процессора и необходимости перехода в энергосберегающий режим. И наконец, на самых новых системах таймеры с высоким разрешением позволяют ядру отслеживать время с джиффи-градацией.

Как и большинство UNIX-систем, Linux связывает с каждым потоком значение *nice*. По умолчанию оно равно 0, но его можно изменить при помощи системного вызова *nice(value)*, где *value* меняется от –20 до +19. Это значение определяет статический приоритет каждого потока. Вычисляющий в фоновой программе значение числа π (с точностью до миллиарда знаков) пользователь может использовать этот вызов в своей программе, чтобы быть вежливым по отношению к другим пользователям. Только системный администратор может запросить лучшее по сравнению с другими обслуживание (со значением от –20 до –1). В качестве упражнения вы можете попробовать определить причину этого ограничения.

Далее более подробно будут рассмотрены два применяемых в Linux алгоритма планирования. Их свойства тесно связаны с конструкцией очереди выполнения (runqueue), основной структуры данных, используемой планировщиком для отслеживания всех имеющихся в системе задач, готовых к выполнению, и выбора следующей запускаемой задачи. Очередь выполнения связана с каждым имеющимся в системе центральным процессором. Исторически сложилось так, что самым популярным Linux-планировщиком был Linux O(1). Свое название он получил из-за способности выполнять операции управления задачами, такие как выбор задачи или постановка задачи в очередь выполнения за одно и то же время независимо от количества задач в системе. В планировщике O(1) очередь выполнения организована в двух массивах: активных задач и задач, чье время истекло. Как показано на рис.4,*а*, каждый из них представляет собой массив из 140 заголовков списков, соответствующих разным приоритетам. Каждый заголовок списка указывает на дважды связанный список процессов заданного приоритета Основы работы планировщика можно описать следующим образом.

Планировщик выбирает в массиве активных задач задачу из списка задач с самым высоким приоритетом. Если квант времени этой задачи истек, то она переносится в список задач, чье время истекло (возможно, с другим уровнем приоритета). Если задача блокируется (например, в ожидании события ввода-вывода) до истечения ее кванта времени, то после события она помещается обратно в исходный массив активных задач, а ее квант уменьшается на количество уже использованного времени процессора. После полного истечения кванта времени она также будет помещена в массив задач, чье время истекло. Когда в массиве активных задач ничего не остается, планировщик просто меняет указатели, чтобы массивы задач, чье время истекло, стали массивами активных задач, и наоборот. Этот метод гарантирует, что задачи с низким приоритетом не «умирают от голода» (за исключением случая, когда потоки реального времени со схемой FIFO полностью захватывают ресурсы процессора, что маловероятно).

 **Рис.4.** Иллюстрация очереди исполнения: *а* — для Linux-планировщика O(1);

*б* — для панировщика Completely Fair Scheduler

При этом разным уровням приоритета присваиваются разные размеры квантов времени и больший квант времени присваивается процессам с более высоким приоритетом. Например, задачи с уровнем приоритета 100 получат квант времени 800 мс, а задачи с уровнем приоритета 139 — 5 мс.

Идея данной схемы в том, чтобы быстро убрать процессы из ядра. Если процесс пытается читать дисковый файл, то секунда ожидания между вызовами *read* невероятно затормозит его. Гораздо лучше позволить ему выполняться сразу же после завершения каждого запроса, чтобы он мог быстро выполнить следующий запрос. Аналогично, если процесс был блокирован в ожидании клавиатурного ввода, то это точно интерактивный процесс, и ему нужно дать высокий приоритет сразу же, как он станет готов к выполнению, чтобы обеспечить хорошее обслуживание интерактивных процессов. Поэтому интенсивно использующие процессор процессы получают все, что остается (когда блокированы все завязанные на ввод-вывод и интерактивные процессы).

Поскольку Linux (как и любая другая операционная система) заранее не знает, будет задача интенсивно использовать процессор или ввод-вывод, она применяет эвристику интерактивности. Для этого Linux различает статический и динамический приоритет. Динамический приоритет потока постоянно пересчитывается для того, чтобы, во-первых, поощрять интерактивные потоки, а во-вторых, наказывать пожирателей процессорных ресурсов. В планировщике O(1) максимальное увеличение приоритета равно –5, поскольку более низкое значение приоритета соответствует более высокому приоритету, полученному планировщиком. Максимальное снижение приоритета равно +5. Планировщик поддерживает связанную с каждой задачей переменную *sleep\_avg*. Когда задача просыпается, эта переменная получает приращение, когда задача вытесняется или истекает ее квант, эта переменная уменьшается на соответствующее значение. Это значение используется для динамического изменения приоритета на величину от –5 до +5. Планировщик Linux пересчитывает новый уровень приоритета при перемещении потока из списка активных в список закончивших функционирование.

Алгоритм планирования O(1) относится к планировщику, ставшему популярным в ранних версиях ядра 2.6, а впервые он был введен в нестабильной версии ядра 2.5. Предыдущие алгоритмы показывали низкую производительность на многопроцессорных системах и не могли хорошо масштабироваться с увеличением количества задач. Поскольку представленное в предыдущем параграфе описание указывает, что решение планирования может быть принято путем доступа к соответствующему списку активных процессов, его можно выполнить за постоянное время О(1), не зависящее от количества процессов в системе. Но несмотря на положительное свойство, заключавшееся в постоянном времени проведения операций, у планировщика O(1) имелись существенные недостатки. Наиболее значимый состоял в том, что эвристическое правило, используемое для определения степени интерактивности задачи, а следовательно уровня ее приоритета, было слишком сложным и несовершенным, что выражалось в низкой производительности интерактивных задач.

Для решения этой проблемы Инго Молнар (Ingo Molnar), который также был создателем планировщика O(1), предложил новый планировщик, названный совершенно справедливым планировщиком (Completely Fair Scheduler (CFS)). Этот планировщик был основан на идее, изначально принадлежавшей Кону Коливасу (Con Kolivas) для более раннего планировщика, и впервые был интегрирован в ядро версии 2.6.23. Он и сейчас является планировщиком по умолчанию для задач, не являющихся задачами реального времени.

Главная идея, положенная в основу CFS, заключается в использовании в качестве структуры очереди выполнения *красно-черного дерева*. Задачи в дереве выстраиваются на основе времени, которое затрачивается на их выполнение на ценральном процессоре и называется виртуальным временем выполнения — *vruntime*. CFS подсчитывает время выполнения задачи с точностью до наносекунды. Как показано на рис.4, *б*, каждый внутренний узел дерева соотносится с задачей. Дочерние узлы слева соотносятся с задачами, которые тратят меньше времени центрального процессора, и поэтому их выполнение будет спланировано раньше, а дочерние записи справа от узла относятся к задачам, которые до этих пор потребляли больше времени центрального процессора. Листья дерева не играют в планировщике никакой роли.

Алгоритм планирования может быть кратко изложен следующим образом. CFS всегда планирует задачу, которой требуется наименьшее количество времени центрального процессора, обычно это самый левый узел дерева. Периодически CFS увеличивает значение *vruntime* задачи на основе того времени, которое уже было затрачено на ее выполение, и сравнивает его со значением текущего самого левого узла дерева. Если выполняемая задача все еще имеет наименьшее значение *vruntime*, ее выполнение продолжается. В противном случае она будет вставлена в соответствующее место в красно-черном дереве, и центральный процессор получит задачу, соответсвующую новому самому левому узлу дерева.

Чтобы учесть различия между приоритетами задач и значениями переменной *nice*, CFS изменяет эффективную ставку, при которой проходит виртуальное время выполнения задания, когда оно запущено на центральном процессоре. Для задач с более низким уровнем приоритета время течет быстрее, их значение *vruntime* будет расти быстрее, и в зависимости от других задач в системе они будут отлучаться от центрального процессора и заново вставляться в дерево раньше, чем если бы у них был более высокий приоритет. Благодаря этому CFS избегает использования отдельных структур очередей выполнения для разных уровней приоритетов.

Таким образом, выбор узла для выполнения может быть сделан за постоянное время, а вот вставка задачи в очередь выполнения осуществляется за время *O*(log(*N*)), где *N* — количество задач в системе. С учетом уровней загруженности текущих систем это время продолжает оставаться вполне приемлемым, но с увеличением вычисляемого объема узлов и количества задач, которые могут выполняться в системах, в частности в серверном пространстве, вполне возможно, что в дальнейшем будут предложены новые алгоритмы планирования.

Кроме основных алгоритмов планирования Linux-планировщик имеет специальные функциональные возможности, которые особенно полезны для многопроцессорных или многоядерных платформ. Во-первых, с каждым процессором многопроцессорной системы связана структура очереди выполнения. Планировщик старается использовать преимущества родственного планирования и планировать задачи на тот процессор, на котором они выполнялись ранее. Во-вторых, имеется набор системных вызовов для указания требований к родственной принадлежности потока. И наконец, планировщик выполняет периодическую балансировку нагрузки между очередями выполнения разных процессоров, чтобы обеспечить балансировку загрузки системы (выдерживая в то же время определенные требования к производительности и соблюдению родственной принадлежности).

Планировщик рассматривает только готовые к выполнению задачи, которые помещаются в соответствующие очереди выполнения. Те задачи, которые не готовы к выполнению и ждут выполнения различных операций ввода-вывода (или других событий ядра), помещаются в другую структуру данных (**очередь ожидания** —waitqueue). Такая очередь связана с каждым событием, которого могут дожидаться задачи. Заголовок очереди ожидания содержит указатель на связанный список задач и спин-блокировку. Спин-блокировка нужна для обеспечения одновременного манипулирования очередью ожидания из кода ядра и обработчиков прерываний (или других асинхронных вызовов).

**Синхронизация в Linux**

В предыдущих разделах уже упоминалось, что для предотвращения параллельного изменения структур данных, подобных очередям ожидания, в Linux используются спин-блокировки. Фактически в коде ядра переменные синхронизации встречаются во многих местах. Далее будет предоставлен краткий обзор конструкций синхронизации, доступных в Linux.

Более ранние ядра системы Linux имели просто одну большую блокировку ядра (big kernel lock (**BLK**)). Это решение оказалось крайне неэффективным, особенно для многопроцессорных платформ (поскольку мешало процессам на разных процессорах одновременно выполнять код ядра). Поэтому было введено множество новых точек синхронизации (с гораздо большей избирательностью).

Linux предоставляет несколько типов переменных синхронизации, которые используются внутри ядра и доступны приложениям и библиотекам на пользовательском уровне. На самом нижнем уровне Linux предоставляет оболочки вокруг аппаратно поддерживаемых атомарных инструкций с помощью операций *atomic\_set* и *atomic\_read*. Кроме этого, поскольку современное оборудование изменяет порядок операций с памятью, Linux предоставляет барьеры памяти. Использование таких операций, как *rmb* и *wmb*, гарантирует, что все относящиеся к памяти операции чтения-записи, предшествующие вызову барьера, завершаются до любого последующего обращения к памяти.

Чаще используемые конструкции синхронизации относятся к более высокому уровню. Потоки, не желающие осуществлять блокировку (из соображений производительности или точности), используют обычные спин-блокировки, а также спин-блокировки по чтению-записи. В текущей версии Linux реализуется так называемая билетная (ticketbased) спин-блокировка, имеющая выдающуюся производительность на SMP и мультиядерных системах. Потоки, которым разрешено или которые нуждаются в блокировке, используют такие конструкции, как мьютексы и семафоры. Для выявления состояния переменной синхронизации без блокировки Linux поддерживает неблокируемые вызовы, подобные *mutex\_trylock* и *sem\_trywait*. Поддерживаются и другие типы переменных синхронизации вроде фьютексов (futexes), завершений (completions), блокировок «чтение — копирование — обновление» (read — copy — update (RCU)) и т. д. И наконец, синхронизация между ядром и кодом, выполняемым подпрограммами обработки прерываний, может также достигаться путем динамического отключения и включения соответствующих прерываний.

**Загрузка Linux.** Точные детали процесса загрузки варьируются от системы к системе, но в основном загрузка состоит из следующих шагов. Когда компьютер включается, система BIOS выполняет тестирование при включении (Power-On-Self-Test, POST), а также начальное обнаружение устройств и их инициализацию (поскольку процесс загрузки операционной системы зависит от доступа к дискам, экрану, клавиатуре и т. д.). Затем в память считывается и исполняется первый сектор загрузочного диска (**главная загрузочная запись** — Master Boot Record (**MBR**)). Этот сектор содержит небольшую (512-байтовую) программу, считывающую автономную программу под названием *boot* с загрузочного устройства, например с SATAили SCSI-диска. Программа boot сначала копирует саму себя в фиксированный адрес памяти в старших адресах, чтобы освободить нижнюю память для операционной системы.

После этого перемещения программа boot считывает корневой каталог с загрузочного устройства. Чтобы сделать это, она должна понимать формат файловой системы и каталога (например, в случае загрузчика **GRUB** (GRand Unified Bootloader)). Другие популярные загрузчики (такие, как LILO компании Intel) от файловой системы не зависят. Им нужны карта блоков и адреса низкого уровня, которые описывают физические сектора, головки и цилиндры (для поиска необходимых для загрузки секторов).

Затем boot считывает ядро операционной системы и передает ему управление. На этом программа boot завершает свою работу, после чего уже работает ядро системы.

Начальный код ядра написан на ассемблере и является в значительной мере машинно зависимым. Как правило, этот код настраивает стек ядра, определяет тип центрального процессора, вычисляет количество имеющейся в наличии оперативной памяти, отключает прерывания, разрешает работу блока управления памятью и, наконец, вызывает процедуру main (написанную на языке C), чтобы запустить основную часть операционной системы.

Код на языке C также должен проделать значительную работу по инициализации, но эта инициализация скорее логическая, нежели физическая. Она начинается с того, что выделяется память под буфер сообщений, что должно помочь отладке проблем с загрузкой системы. По мере выполнения инициализации в этот буфер записываются сообщения, информирующие о том, что происходит в системе. В случае неудачной загрузки их можно выудить оттуда с помощью специальной программы диагностики. Этот буфер подобен черному ящику, который обычно ищут на месте крушения самолета.

Затем выделяется память для структур данных ядра. Большинство этих структур имеют фиксированный размер, но, например, размер кэша страниц и некоторых структур таблиц страниц зависит от доступного объема оперативной памяти.

Затем операционная система начинает определение конфигурации компьютера. Операционная система считывает файлы конфигурации, в которых сообщается, какие типы устройств ввода-вывода могут присутствовать, и проверяет, какие из устройств действительно присутствуют. Если проверяемое устройство отвечает, то оно добавляется к таблице подключенных устройств. Если устройство не отвечает, то оно считается отсутствующим и в дальнейшем игнорируется. В отличие от традиционных версий UNIX, драйверы устройств системы Linux не обязаны быть статически связанными и могут загружаться динамически (как это, кстати, может быть сделано во всех версиях MS-DOS и Windows).

Аргументы в пользу динамической загрузки драйверов и против нее весьма интересны, и их стоит кратко упомянуть. Главный аргумент в пользу динамической загрузки заключается в том, что клиентам с различными конфигурациями может быть поставлен один и тот же двоичный файл, который автоматически загрузит необходимые ему драйверы, возможно, даже по сети. Главный аргумент против динамической загрузки состоит в том, что этот метод противоречит принципам безопасности системы. Если вы обеспечиваете работу защищенного сайта (например, базы данных банка или корпоративного веб-сервера), то, вероятно, захотите сделать невозможной вставку случайного кода в ядро операционной системы. Системный администратор может хранить исходные тексты операционной системы и объектные файлы на защищенной машине и выполнять на ней все работы по сборке системы, после чего переносить двоичный код ядра на другие машины по локальной сети. Если драйверы не могут загружаться динамически, то такой сценарий предотвращает установку в ядро неотлаженного или злонамеренного кода (системными операторами или еще кем-либо, кому известен пароль суперпользователя). Более того, в больших системах конфигурация аппаратуры точно известна уже во время компиляции и компоновки операционной системы. Изменения производятся довольно редко, поэтому перекомпоновка системы при добавлении нового устройства не представляет собой проблемы.

После завершения конфигурации всего аппаратного обеспечения нужно аккуратно загрузить процесс 0, настроить его стек и запустить этот процесс. Процесс 0 продолжает инициализацию, выполняя такие задачи, как программирование таймера реального времени, монтирование корневой файловой системы и создание процесса 1 (init) и страничного демона (процесс 2).

Процесс init проверяет свои флаги, в зависимости от которых он запускает операционную систему либо в однопользовательском, либо в многопользовательском режиме. В первом случае он создает процесс, выполняющий оболочку, и ждет, когда тот завершит свою работу. Во втором случае процесс init создает процесс, исполняющий инициализационный сценарий оболочки системы /etc/rc, который может выполнить проверку непротиворечивости файловой системы, смонтировать дополнительные файловые системы, запустить демонов и т. д. Затем он считывает файл /etc/ttys, в котором перечисляются терминалы и некоторые их свойства. Для каждого разрешенного терминала он создает копию самого себя, которая затем выполняет программу getty.



**Рис. 5.** Последовательность исполняемых процессов при загрузке некоторых версий системы Linux

Программа getty устанавливает для каждой линии (некоторые из них могут быть, например, модемами) скорость и прочие свойства, после чего выводит на терминале приглашение к входу в систему:

login:

После этого программа getty пытается прочитать с клавиатуры имя пользователя. Когда пользователь садится за терминал и вводит свое регистрационное имя, программа getty завершает свою работу выполнением программы регистрации /bin/login. После этого программа login запрашивает у пользователя его пароль, шифрует его и сравнивает с зашифрованным паролем, хранящимся в файле паролей /etc/passwd. Если пароль введен верно, то программа login вместо себя запускает оболочку пользователя, которая ожидает первой команды. Если пароль введен неверно, то программа login еще раз спрашивает имя пользователя. Этот механизм проиллюстрирован на рис.5 для системы с тремя терминалами.

На рисунке процесс getty, работающий на терминале 0, все еще ждет ввода. На терминале 1 пользователь ввел имя регистрации, поэтому программа getty запустила поверх себя процесс login, запрашивающий пароль. На терминале 2 уже произошла успешная регистрация, в результате чего оболочка вывела приглашение к вводу (*%*).

Пользователь ввел команду cp f1 f2 в результате которой оболочка создала дочерний процесс, исполняющий программу *cp*. Процесс оболочки заблокирован в ожидании завершения дочернего процесса, после чего оболочка снова выведет приглашение к вводу и будет ждать ввода с клавиатуры. Если бы пользователь на терминале 2 вместо *cp* ввел *cc*, то запустилась бы главная программа компилятора языка C, который, в свою очередь, запустил бы несколько дочерних процессов для выполнения различных проходов компилятора.