Глава 10. Краткий обзор современных операционных систем

Теперь, после знакомства с основными понятиями, относящимися к операционным системам, и изучения конкретных механизмов, реализующих известные методы организации вычислительных процессов, вкратце рассмотрим архитектурные особенности современных операционных систем для персональных компьютеров типа IBM PC.

Прежде всего, отметим тот общеизвестный факт, что наиболее популярными являются операционные системы семейства Windows компании Microsoft. Это и Windows 95/98/ME, и Windows NT/2000, и новое поколение Windows XP/2003 — этим операционным системам посвящена отдельная глава (см. главу 11). Здесь же мы рассмотрим операционные системы, не относящиеся к продуктам Microsoft, — это UNIX-подобные операционные системы Linux и FreeBSD, а также системы QNX и OS/2. При изучении известных всему миру систем с общим названием Linux и системы FreeBSD, по которым сейчас появляется немало монографий и учебников, упор будет сделан именно на основных архитектурных особенностях семейства UNIX, в абсолютном своем большинстве относящихся ко всем UNIX-системам. Система QNX была выбрана потому, что является наиболее известной и удачной операционной системой реального времени. Операционную систему OS/2 мы рассмотрим последней. Хотя сейчас эта система уже практически всеми забыта[1], она была одной из первых полноценных и надежных мультипрограммных и мультизадачных операционных систем для персональных компьютеров, в которой поддерживалось несколько операционных сред.

[1] В настоящее время ее используют те организации, которые в свое время создали под нее свои приложения, вложив немалые средства. И поскольку система по-прежнему в основном неплохо выполняет свои функции, эти организации не спешат вкладывать деньги для переноса своих задач на новые платформы.

Семейство операционных систем UNIX

UNIX является исключительно удачным примером реализации простой мультипрограммной и многопользовательской операционной системы. В свое время она проектировалась как инструментальная система для разработки программного обеспечения. Своей уникальностью система UNIX обязана во многом тому обстоятельству, что была, по сути, создана всего двумя разработчиками[1], которые делали ее исключительно для себя и первое время использовали на мини-ЭВМ с очень скромными вычислительными ресурсами. Первая версия этой системы занимала всего около 12 Кбайт и могла работать на компьютерах с очень небольшим объемом оперативной памяти. Поскольку при создании второй версии UNIX разработчики отказались от языка ассемблера и специально придумали язык высокого уровня, на котором можно было бы писать не только системные, но и прикладные программы (речь идет о языке C), то и сама система UNIX, и приложения, выполняющиеся в ней, стали легко переносимыми (мобильными). Компилятор с языка C для всех оттранслированных программ дает реентерабельный и разделяемый код, что позволяет эффективно использовать имеющиеся в системе ресурсы.

Общая характеристика и особенности архитектуры

Первой целью при разработке этой системы было стремление сохранить простоту и обойтись минимальным количеством функций. Все реальные сложности оставлялись пользовательским программам.

Второй целью была общность. Одни и те же методы и механизмы должны были использоваться во многих случаях:

* обращение к файлам, устройствам ввода-вывода и буферам межпроцессных сообщений выполняются с помощью одних и тех же примитивов;
* одни и те же механизмы именования, присвоения альтернативных имен и защиты от несанкционированного доступа применяются и к файлам с данными, и к каталогам, и к устройствам;
* одни и те же механизмы работают в отношении программно и аппаратно инициируемых прерываний.

Третья цель заключалась в том, чтобы сложные задачи можно было решать, комбинируя существующие небольшие программы, а не разрабатывая их заново.

Наконец, четвертая цель состояла в создании мультитерминальной операционной системы с эффективными механизмами разделения не только процессорного времени, но и всех остальных ресурсов. В мультитерминальной операционной системе на одно из первых мест по значимости выходят вопросы защиты одних вычис-

[1] Создателями системы UNIX считаются Кен Томпсон и Деннис Ритчи. В своей операционной системе Томпсон и Ритчи учли опыт работы над проектом сложной мультизадачной операционной системы с разделением времени, которая имела название MULTICS (MULTiplexed Information and Computing System) Название новой системы UNIX произошло от аббревиатуры UNICS (Uniplexed Information and Computing System)

лительных процессов от вмешательства других вычислительных процессов. Причем для реализации третьей цели необходимо было создать механизмы полноценного обмена данными между программными модулями, из которых предполагалось составлять конечные программы.

Операционная система UNIX обладает простым, но очень мощным командным языком и независимой от устройств файловой системой. Важным, хотя и простым с позиций реализации такой возможности, является тот факт, что система UNIX предоставляет пользователям средства направления выхода одной программы непосредственно на вход другой. В результате достигается четвертая цель — большие программные системы можно создавать путем композиции имеющихся небольших программ, а не путем написания новых, что в большинстве случаев упрощает задачу. UNIX-системы существуют уже 30 лет, и к настоящему времени имеется чрезвычайно большой набор легко переносимых из системы в систему отлично отлаженных и проверенных временем приложений.

В число системных и прикладных программ, поставляемых с UNIX-системами, входят редакторы текстов, программируемые интерпретаторы командного языка, компиляторы с нескольких популярных языков программирования, включая C, C++, ассемблер, PERL, FORTRAN и многие другие, компоновщики (редакторы межпрограммных связей), отладчики, многочисленные библиотеки системных и пользовательских программ, средства сортировки и ведения баз данных, многочисленные административные и обслуживающие программы. Для абсолютного большинства всех этих программ имеется документация, в том числе исходные тексты программ (как правило, хорошо комментированные). Кроме того, описания и документация по большей части доступны пользователям в интерактивном режиме. Используется иерархическая файловая система с полной защитой, работа со съемными томами, обеспечивается независимость от устройств.

Центральной частью UNIX-систем является ядро (kernel). Оно состоит из большого количества модулей и с точки зрения архитектуры считается монолитным. Однако в ядре всегда можно выделить три основные подсистемы: управления процессами, управления файлами, управления операциями ввода-вывода между центральной частью и периферийными устройствами. Подсистема управления процессами организует выполнение и диспетчеризацию процессов, их синхронизацию и разнообразное межпроцессное взаимодействие. Важнейшая функция подсистемы управления процессами — это распределение оперативной памяти и (для современных систем) организация виртуальной памяти. Подсистема управления файлами тесно связана и с подсистемой управления процессами, и с драйверами. Ядро может быть перекомпилировано с учетом конкретного состава устройств компьютера и решаемых задач. Не все драйверы могут быть включены в состав ядра, часть из них может вызываться из ядра. Более того, очень большое количество системных функций выполняется системными программными модулями, не входящими непосредственно в ядро, но вызываемых из ядра. Основные системные функции, которые должно выполнять ядро совместно с остальными системными модулями, строго стандартизированы. За счет этого во многом достигается переносимость кода между разными версиями UNIX и абсолютно различным аппаратным обеспечением.

Основные понятия

Одним из достоинств ОС UNIX является то, что система базируется на небольшом числе понятий; рассмотрим их вкратце. Здесь необходимо отметить, что настоящая книга не претендует на полноценное изложение основ работы и детальное описание архитектуры системы UNIX (или Linux). На эту тему имеется достаточное количество специальной литературы, например отличная монография [39] или такие замечательные книги, как [23, 43]. Тем не менее, исходя из имеющегося опыта преподавания предметов, относящихся к операционным системам и системному программному обеспечению, считаю полезным изложить здесь минимальный набор основных понятий, который часто помогает студентам «погрузиться в мир UNIX», отличающийся от привычного всем окружения Windows.

Виртуальная машина

Система UNIX многопользовательская. Каждому пользователю после регистрации (входа в систему) предоставляется виртуальный компьютер, в котором есть все необходимые ресурсы: процессор (процессорное время выделяется на основе круговой, или карусельной, диспетчеризации и с использованием динамических приоритетов, что позволяет обеспечить равенство в обслуживании), оперативная память, устройства, файлы. Текущее состояние такого виртуального компьютера, предоставляемого пользователю, называется *образом*. Можно сказать, что процесс — это выполнение образа. Образ процесса состоит:

* из образа памяти;
* значений общих регистров процессора;
* состояния открытых файлов;
* текущего каталога файлов;
* другой информации.

Образ процесса во время выполнения процесса размещается в основной памяти. В старых версиях UNIX образ можно было «сбросить» на диск, если какому-либо более приоритетному процессу требовалось место в основной памяти. Напомним, что такое замещение процессов называется свопингом (swapping). В современных реализациях, поддерживающих, как правило, страничный механизм виртуальной памяти, прежде всего выгружаются неиспользуемые страницы, а не целиком образ. В частности, в системах Linux свопинг образов не применяется, но создается специальный[1] раздел на магнитном диске для файла подкачки (swap-file), где размещаются виртуальные страницы выполняющихся процессов, для которых не хватает места в оперативной памяти. Таким образом, замещаются не процессы, а их отдельные страницы.

Образ памяти делится на три логических сегмента:

* сегмент реентерабельных процедур (начинается с нулевого адреса в виртуальном адресном пространстве процесса);

[1] Сигнатура этого раздела обозначается как 082h

* сегмент данных (располагается следом за сегментом процедур и может расти в сторону больших адресов);
* сегмент стека (начинается со старшего адреса и растет в сторону младших адресов по мере занесения в него информации при вызовах подпрограмм и при прерываниях).

В современных версиях UNIX-систем все виртуальное адресное пространство каждого образа отображается на реальную физическую память компьютера. Используется страничный механизм организации виртуальной памяти. И следует различать замещение процессов и подкачку страниц, хотя в обоих случаях используется термин swapping.

Пользователь

Мы уже отмечали, что с самого начала операционная система UNIX замышлялась как интерактивная многопользовательская система. Другими словами, UNIX предназначена для мультитерминальной работы. Чтобы начать работать, пользователь должен «войти» в систему, введя со свободного терминала свое учетное, или входное, имя (account name, или login) и пароль (password). Человек, зарегистрированный в учетных файлах системы и, следовательно, имеющий учетное имя, называется зарегистрированным пользователем системы. Регистрацию новых пользователей обычно выполняет администратор системы. Пользователь не может изменить свое учетное имя, но может установить и/или изменить свой пароль. Пароли хранятся в отдельном файле в закодированном виде.

Ядро операционной системы UNIX идентифицирует каждого пользователя по его идентификатору (User Identifier, UID), уникальному целому значению, присваиваемому пользователю при регистрации в системе. Кроме того, каждый пользователь относится к некоторой группе пользователей, которая также идентифицируется некоторым целым значением (Group Identifier, GID). Значения UID и GID для каждого зарегистрированного пользователя сохраняются в учетных файлах системы и приписываются процессу, в котором выполняется командный интерпретатор, запущенный при входе пользователя в систему. Эти значения наследуются каждым новым процессом, запущенным от имени данного пользователя, и используются ядром системы для контроля правомочности доступа к файлам, выполнения программ и т. д.

Все пользователи операционной системы UNIX явно или неявно работают с файлами. Файловая система операционной системы UNIX имеет древовидную структуру [39]. Промежуточными узлами дерева являются каталоги со ссылками на другие каталоги или файлы, а листья дерева соответствуют файлам или пустым каталогам. Каждому зарегистрированному пользователю соответствует некоторый каталог файловой системы, который называется *домашним* (home) каталогом пользователя. При входе в систему пользователь получает неограниченный доступ к своему домашнему каталогу и всем каталогам и файлам, содержащимся в нем. Пользователь может создавать, удалять и модифицировать каталоги и файлы, содержащиеся в домашнем каталоге. Потенциально возможен доступ и ко всем другим файлам, однако он может быть ограничен, если пользователь не имеет достаточных привилегий.

Суперпользователь

Очевидно, что администратор системы, который тоже является зарегистрированным пользователем, чтобы управлять всей системой, должен обладать существенно большими, чем обычные пользователи, привилегиями. В операционных системах UNIX эта задача решается путем выделения единственного нулевого значения UID. Пользователь с таким значением UID называется *суперпользователем* (superuser) и обозначается словом *root* (корень). Он имеет неограниченные права на доступ к любому файлу и на выполнение любой программы. Кроме того, такой пользователь имеет возможность полного контроля над системой. Он может остановить ее и даже разрушить. По этой причине не рекомендуется работать под этой учетной записью. Администратор должен создать себе обычную учетную запись простого пользователя, а для выполнения действий, связанных с административными полномочиями, рекомендуется использовать команду su. Команда su запрашивает у пользователя пароль суперпользователя, и, если он указан правильно, операционная система переводит сеанс пользователя в режим работы суперпользователя. После выполнения необходимых действий, требующих привилегий суперпользователя, следует выполнить команду exit, которая и вернет администратору статус простого пользователя.

Еще одним важным отличием суперпользователя от обычного пользователя операционной системы UNIX является то, что на суперпользователя не распространяются ограничения на используемые ресурсы. Для обычных пользователей устанавливаются такие ограничения, как максимальный размер файла, максимальное число сегментов разделяемой памяти, максимально допустимое пространство на диске и т. д. Суперпользователь может изменять эти ограничения для других пользователей, но на него они не действуют.

Интерфейс пользователя

Традиционный способ взаимодействия пользователя с системой UNIX основывается на командных языках. После входа пользователя в систему для него запускается один из командных интерпретаторов (в зависимости от параметров, сохраняемых в файле /etc/passwd). Обычно в системе поддерживается несколько командных интерпретаторов с похожими, но различающимися своими возможностями командными языками. Общее название для любого командного интерпретатора ОС UNIX — *оболочка* (shell), поскольку любой интерпретатор представляет внешнее окружение ядра системы. По умолчанию в системах Linux командным интерпретатором является bash. В принципе он может быть заменен другим, но практически никто этого не делает.

Вызванный командный интерпретатор выдает приглашение на ввод пользователем командной строки, которая может содержать простую команду, конвейер команд или последовательность команд. После выполнения очередной командной строки и выдачи на экран терминала или в файл соответствующих результатов интерпретатор команд снова выдает приглашение на ввод командной строки, и так до тех пор, пока пользователь не завершит свой сеанс работы и не выйдет из системы.

Командные языки, используемые в UNIX, достаточно просты, чтобы новые пользователи могли быстро начать работать, и достаточно мощны, чтобы можно было использовать их для написания сложных программ. Последняя возможность опирается на механизм *командных файлов* (shell scripts), которые могут содержать произвольные последовательности командных строк. При указании имени командного файла вместо очередной команды интерпретатор читает файл строка за строкой и последовательно интерпретирует команды.

Поскольку в настоящее время все большее распространение получают графические интерфейсы, в операционных системах семейства UNIX стали все чаще работать в X-Window. X-Window — это графический интерфейс, позволяющий пользователям взаимодействовать со своими вычислениями и с системой в графическом режиме. В отличие от систем Windows компании Microsoft, графический интерфейс для UNIX-систем не является основным, в системе можно работать и без него. Прежде всего, графический режим разрабатывался для приложений, предназначенных для работы с графикой. Однако в последние годы его стали применять гораздо чаще, особенно в системах Linux, которые начинают использовать не только как серверные операционные системы, но и как системы для персональных компьютеров.

Графический интерфейс в UNIX-системах основан на модели клиент-сервер. Серверная часть X-Window — это аппаратно-зависимая система ввода-вывода, которая непосредственно взаимодействует с приложением и видеоподсистемой, клавиатурой и мышью. При этом серверная часть должна работать на компьютере, производящем вычисления. Взаимодействие с пользователем осуществляется через клиентскую часть, которая обеспечивает вывод данных на дисплей и прием их с устройств ввода. Клиентская часть должна быть на том компьютере, за которым работает пользователь. Таким образом, можно работать в графическом режиме, сидя за одним компьютером, в то время как собственно вычисления могут происходить и на другом компьютере.

Один из клиентов X-Window — это оконный менеджер (также называемый диспетчером окон). Он управляет размещением окон на экране, определяет их вид и характер управляющих элементов. То есть именно он и предоставляет пользователю графический интерфейс (GUI), тогда как X-Window — это его основа.

В системах Linux наиболее популярными менеджерами графического интерфейса являются KDE и GNOME. Для запуска X-Window в системах семейства UNIX (и Linux) используется команда startx.

Команды и командный интерпретатор

Как уже упоминалось, оболочкой (shell) в UNIX-системе называют механизм взаимодействия между пользователями и системой. По сути дела, это интерпретатор команд, который считывает набираемые пользователем строки и запускает указанные в командах программы, которые и выполняют запрошенные системные функции и операции. Полный командный язык, интерпретируемый оболочкой, богат возможностями и достаточно сложен, однако большинство команд просты в использовании, и запомнить их не составляет труда.

Командная строка состоит из имени команды (а именно имени выполняемого файла), за которым следует список аргументов, разделенных пробелами. Оболочка разбивает командную строку на компоненты. Указанный в команде файл загружается, и ему обеспечивается доступ к заданным в команде аргументам.

Любой командный язык оболочки фактически состоит из трех частей:

* служебных конструкций, позволяющих манипулировать текстовыми строками и строить сложные команды на основе простых команд;
* встроенных команд, выполняемых непосредственно интерпретатором командного языка;
* команд, представляемых отдельными выполняемыми файлами.

В свою очередь, набор команд последнего вида включает стандартные команды (системные утилиты, такие как vi, cc и т. д.) и команды, созданные пользователями системы. Для того чтобы выполняемый файл, разработанный пользователем ОС UNIX, можно было запускать как команду оболочки, достаточно определить в одном из исходных файлов функцию с именем main (имя main должно быть глобальным, то есть перед ним не должно указываться ключевое слово static). Если употребить в качестве имени команды имя такого выполняемого файла, командный интерпретатор создаст новый процесс и запустит в нем указанную выполняемую программу, начиная с вызова функции main.

Тело функции main, вообще говоря, может быть произвольным (для интерпретатора существенно только наличие входной точки в программу с именем main), но для того чтобы создать команду, которой можно задавать параметры, придерживаются некоторых стандартных правил. В этом случае каждая функция main должна определяться с двумя параметрами — argc и argv. После вызова команды параметру argc будет соответствовать число символьных строк, указанных в качестве аргументов вызова команды, а argv — массив указателей на переменные, содержащие эти строки. При этом имя самой команды составляет первую строку аргументов (то есть после вызова значение argc всегда больше или равно 1). Код функции main должен проанализировать допустимость заданного значения argc и соответствующим образом обработать заданные текстовые строки.

Например, следующий текст на языке С может быть использован для создания команды, которая выводит на экран текстовую строку, заданную в качестве ее аргумента:

```
#include <stdio.h>
main (argc, argv)
int  argc;
char  *argv[];
{
  if (argc != 2)
{ printf("usage: %s your-text\n", argv[0]);
  exit;
}
  printf("%s\n",  argv[1]);
}
```

Процессы

Процесс в системах UNIX — это процесс в классическом понимании этого термина, то есть это программа, выполняемая в собственном виртуальном адресном пространстве. Когда пользователь входит в систему, автоматически создается процесс, в котором выполняется программа командного интерпретатора. Если командному интерпретатору встречается команда, соответствующая выполняемому файлу, то он создает новый процесс и запускает в нем соответствующую программу, начиная с функции main. Эта запущенная программа, в свою очередь, может создать процесс и запустить в нем другую программу (та тоже должна содержать функцию main) и т. д.

Для образования нового процесса и запуска в нем программы используются два системных вызова API — fork() и exec(имя_выполняемого_файла). Системный вызов fork() приводит к созданию нового адресного пространства, состояние которого абсолютно идентично состоянию адресного пространства основного процесса (то есть в нем содержатся те же программы и данные). Для дочернего процесса заводятся копии всех сегментов данных.

Другими словами, сразу после выполнения системного вызова fork() основной (родительский) и порожденный процессы являются абсолютными близнецами; управление в том и другом находится в точке, непосредственно следующей за вызовом fork(). Чтобы программа могла разобраться, в каком процессе (основном или порожденном) она теперь работает, функция fork() возвращает разные значения: О в порожденном процессе и целое положительное число в основном процессе. Этим целым положительным числом является уже упоминавшийся идентификатор процесса (PID). Таким образом, родительский процесс будет знать идентификатор своего дочернего процесса и может при необходимости управлять им.

Теперь, если мы хотим запустить новую программу в порожденном процессе, нужно обратиться к системному вызову exec, указав в качестве аргументов вызова имя файла, содержащего новую выполняемую программу, и, возможно, одну или несколько текстовых строк, которые будут переданы в качестве аргументов функции main новой программы. Выполнение системного вызова exec приводит к тому, что в адресное пространство порожденного процесса загружается новая выполняемая программа и запускается *с* адреса, соответствующего входу в функцию main. Другими словами, это приводит к замене текущего программного сегмента и текущего сегмента данных, которые были унаследованы при выполнении вызова fork, соответствующими сегментами, заданными в файле. Прежние сегменты теряются. Это эффективный метод смены выполняемой процессом программы, но не самого процесса. Файлы, уже открытые до вызова примитива exec, остаются открытыми после его выполнения.

В следующем примере пользовательская программа, вызываемая как команда оболочки, выполняет в отдельном процессе стандартную команду ls оболочки, которая выдает на экран содержимое текущего каталога.

```
main()
{if(fork()== 0)    wait(0);    /* родительский процесс */
  else execl("ls", "ls", 0); /* порожденный процесс */
}
```

Таким образом, с практической точки зрения процесс в UNIX является объектом, создаваемым в результате выполнения функции fork(). Каждый процесс за исключением начального (нулевого) порождается в результате вызова другим процессом функции fork(). Каждый процесс имеет одного родителя, но может породить много процессов. Начальный (нулевой) процесс является особенным процессом, который создается в результате загрузки системы. После порождения нового процесса с идентификатором 1 нулевой процесс становится процессом подкачки и реализует механизм виртуальной памяти. Процесс с идентификатором 1, известный под именем init, является предком любого другого процесса в системе и связан с каждым процессом особым образом.

Функционирование

Теперь, когда мы познакомились с основными понятиями, рассмотрим наиболее характерные моменты функционирования UNIX-системы.

Выполнение процессов

Процесс может выполняться в одном из двух состояний, а именно *пользовательском* и *системном*. В пользовательском состоянии процесс выполняет пользовательскую программу и имеет доступ к пользовательскому сегменту данных. В системном состоянии процесс выполняет программы ядра и имеет доступ к системному сегменту данных.

Когда пользовательскому процессу требуется выполнить системную функцию, он делает *системный вызов*. Фактически происходит вызов ядра системы как подпрограммы. С момента системного вызова процесс считается системным. Таким образом, пользовательский и системный процессы являются двумя фазами одного и того же процесса, но они никогда не пересекаются между собой. Каждая фаза пользуется своим собственным стеком. Стек задачи содержит аргументы, локальные переменные и другую информацию относительно функций, выполняемых в режиме задачи. Диспетчерский процесс не имеет пользовательской фазы.

В UNIX-системах организуется *разделение времени* (time sharing), то есть каждому процессу выделяется квант времени. Либо процесс завершается сам до истечения отведенного ему кванта времени, либо он приостанавливается по истечении кванта и продолжает свое исполнение при очередном получении нового кванта времени. Механизм диспетчеризации характеризуется достаточно справедливым распределением процессорного времени между всеми процессами. Пользовательским процессам приписываются приоритеты в зависимости от получаемого ими процессорного времени. Процессам, которые получили много процессорного времени, назначают более низкие приоритеты, в то время как процессам, которые получили лишь немного процессорного времени, наоборот, повышают приоритет. Вспомните рассмотренные ранее механизмы динамических приоритетов. Такой метод диспетчеризации обеспечивает хорошее время реакции для всех пользователей системы. Все системные процессы имеют более высокие приоритеты по сравнению с пользовательскими и поэтому всегда обслуживаются в первую очередь.

Подсистема ввода-вывода

Функции ввода-вывода в UNIX задаются в основном с помощью пяти системных вызовов: open, close, read, write и seek.

Открыть файл можно следующей командой:

```
file_descriptor = open (file_name, mode)
```

Здесь mode — режим открытия файла (чтение, запись или то и другое); file_descriptor — дескриптор файла, служит для последующих ссылок на данный файл; file_name — имя открываемого файла.

Чтение и запись осуществляются командами следующего вида:

```
after_reading_bytes = read (file_descriptor, buffer, bytes)
after_writing_bytes= write (file_descriptor, buffer, bytes)
```

Здесь bytes — количество байтов, которые должны быть прочитаны или записаны; after_reading_bytes и after_writing_bytes — реально прочитанное и записанное количество байтов соответственно.

При чтении возможны три ситуации, в каждой из которых чтение происходит последовательно:

* если это первое чтение из файла, то оно осуществляется последовательно с самого начала файла;
* если операции чтения предшествовала другая операция чтения из этого файла, то текущая операция предоставит данные, непосредственно следующие за предыдущими;
* если предшествовала операция поиска seek (см. далее), то чтение осуществляется последовательно от точки смещения, указанной в операции seek.

Это же справедливо и по отношению к операции записи в файл. Обратите внимание, что все эти вызовы относятся к последовательному доступу и эффект прямой адресации достигается с помощью команды seek, смещающей текущую позицию файла:

```
Seek (file_descriptor, displacement, displacement_type)
```

Здесь параметр displacement_type (тип смещения) определяет, является смещение абсолютным или относительным, а также задано оно числом байтов или числом блоков по 512 байт.

Важно заметить, что команда seek исполняется для магнитных дисков так же, как и для магнитных лент, которые нынче уже практически не используются, но во времена появления и становления UNIX-систем были часто используемым устройством.

Чтобы закрыть файл, достаточно выполнить следующую команду:

```
close (file_descriptor)
```

Еще три примитива — gtty, stty, stat — позволяют получать и задавать информацию о файлах и терминалах.

Те же самые команды ввода-вывода применяются и к физическим устройствам. В UNIX-системах физические устройства представлены специальными файлами в единой структуре файловой системы. Это означает, что пользователь не может

написать зависящую от устройств программу, если только эта зависимость не отражена в самом потоке передаваемых данных. Стандартные файлы ввода и вывода, приписываемые пользовательскому терминалу, открывать обычным путем не требуется. Терминал открывается автоматически по команде входа в систему login.

Система ввода-вывода UNIX в отличие от большинства других систем ориентирована на работу скорее с потоком данных, а не с записями. Здесь *поток данных* (stream)[1] — это последовательность байтов, заканчивающаяся разделителем (то есть символом конца потока). Понятие потока данных позволяет проще добиться независимости от устройств и унификации файлов с физическими устройствами и конвейерами. Тем самым пользователь получает гибкость в работе с группами данных, но на него ложатся и дополнительные заботы, поскольку ему приходится писать программы управления данными. Пользователь может при необходимости относительно легко самостоятельно реализовать работу с записями. Чтобы работать с записями фиксированной длины, достаточно просто задавать постоянную длину во всех командах чтения и записи. Для нахождения позиции нужной записи при фиксированной длине записей нужно умножить длину записи на номер записи и выполнить команду seek. Работу с записями переменной длины можно организовать, если разместить в начале каждой записи поле фиксированного размера, содержащее значение длины записи.

Перенаправление ввода-вывода

Механизм перенаправления ввода-вывода является одним из наиболее элегантных, мощных и одновременно простых механизмов UNIX. Цель, которая ставилась при разработке этого механизма, состоит в следующем. Поскольку UNIX — это интерактивная система, которая создавалась в конце 60-х — начале 70-х годов, то обычно программы считывали текстовые строки с алфавитно-цифрового терминала и выводили результирующие текстовые строки на экран терминала. Для того чтобы обеспечить большую гибкость при использовании таких программ, желательно было иметь возможность вводить в них данные непосредственно из файлов или с выхода других программ и выводить их данные в файл или на вход других программ.

Реализация этого механизма основывается на следующих свойствах операционных систем семейства UNIX. Во-первых, любой ввод-вывод трактуется как ввод из некоторого файла и вывод в некоторый файл. Клавиатура и экран терминала тоже интерпретируются как файлы (первый можно только читать, а во второй можно только писать). Во-вторых, доступ к любому файлу производится через его дескриптор (положительное целое число). Фиксируются три значения дескрипторов файлов. Файл с дескриптором 1 называется файлом стандартного ввода (stdin), файл с дескриптором 2 — файлом стандартного вывода (stdout), и файл с дескриптором 3 — файлом стандартного вывода диагностических сообщений (stderr). В-третьих, программа, запущенная в некотором процессе, «наследует» от породившего процесса все дескрипторы открытых файлов.

[1] Не путать с потоком выполнения, или тредом (thread).

В головном процессе интерпретатора командного языка файлом стандартного ввода является клавиатура терминала пользователя, а файлами стандартного вывода и вывода диагностических сообщений — экран терминала. Однако при запуске любой команды можно сообщить интерпретатору (средствами соответствующего командного языка), какой файл или выход какой программы должен служить файлом стандартного ввода для запускаемой программы, а также какой файл или вход какой программы должен служить для запускаемой программы файлом стандартного вывода или файлом вывода диагностических сообщений. Тогда интерпретатор перед выполнением системного вызова exec открывает указанные файлы, подменяя смысл дескрипторов 1, 2 и 3.

То же самое может проделать и любая другая программа, запускающая третью программу в специально созданном процессе. Следовательно, все, что требуется для нормального функционирования механизма перенаправления ввода-вывода, — это придерживаться при программировании соглашения об использовании дескрипторов stdin, stdout и stderr. Это не очень трудно, поскольку в наиболее распространенных функциях библиотеки ввода-вывода printf, scanf и error вообще не требуется указывать дескриптор файла. Функция printf неявно использует дескриптор stdout, функция scanf — дескриптор stdin, функция error — дескриптор stderr.

Файловая система

Файл в системе UNIX представляет собой множество символов с произвольным доступом. В файле могут содержаться любые данные, помещенные туда пользователем, и файл не имеет никакой иной структуры, кроме той, какую создаст в нем пользователь.

Структура файловой системы

Здесь мы вкратце рассмотрим одну из первых реализаций файловой системы, поскольку основные ее идеи сохраняются до сих пор.

Информация на дисках размещается блоками. В первой версии файловой системы размер блока был равен 512 байт. Во многих современных файловых системах, разработанных для конкретной версии UNIX-клона, размер блока больше. Это позволяет повысить быстродействие файловых операций. Например, в системе FFS (Fast File System — быстродействующая файловая система) размер блока равен 8192 байт.

В рассматриваемой версии файловой системы раздел диска разбивается на следующие области (рис. 10. 1):

* неиспользуемый блок;
* управляющий блок, или суперблок, в котором хранится размер логического диска и границы других областей;
* *i*-список, состоящий из описаний файлов, называемых *l*-узлами;
* область для хранения содержимого файлов.

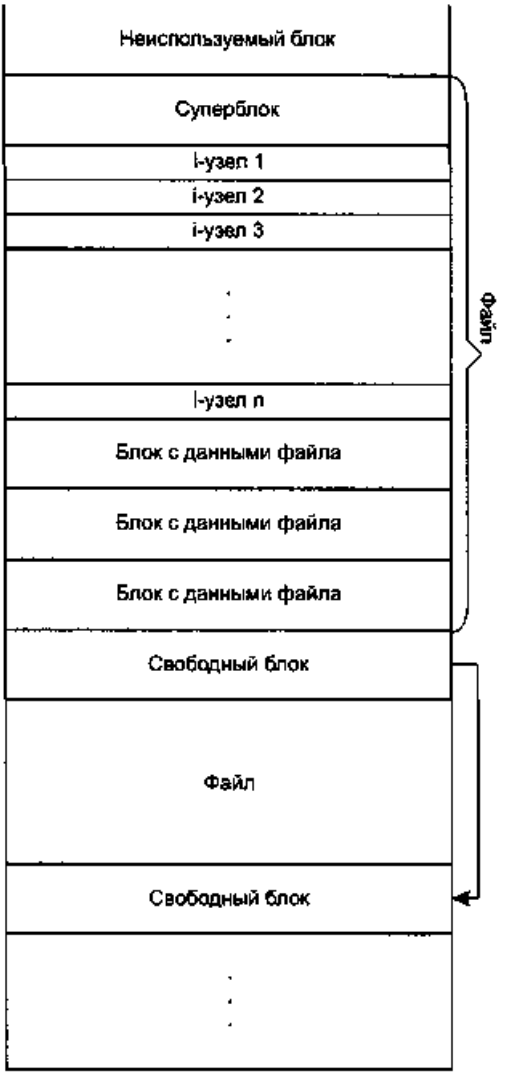

Рис. 10. 1. Организация файловой системы в ОС UNIX

Каждый i-узел содержит:

* идентификатор владельца;
* идентификатор группы владельца;
* биты защиты;
* физические адреса на диске или ленте, где находится содержимое файла;
* размер файла;
* время создания файла;
* время последнего изменения (modification time) файла;

* время последнего изменения атрибутов (change time) файла;
* число связей-ссылок, указывающих на файл;
* индикатор типа файла (каталог, обычный файл или специальный файл).

Следом за *i*-списком идут блоки, предназначенные для хранения содержимого файлов. Пространство на диске, оставшееся свободным от файлов, образует связанный список свободных блоков.

Таким образом, файловая система UNIX представляет собой структуру данных, размещенную на диске и содержащую управляющий суперблок с описанием файловой системы в целом, массив *i*-узлов, в котором определены все файлы в файловой системе, сами файлы и, наконец, совокупность свободных блоков. Выделение пространства под данные осуществляется блоками фиксированного размера.

Каждый файл однозначно идентифицируется *старшим номером устройства, младшим номером устройства* и *i-номером* (индексом *i*-узла данного файла в массиве *i*-узлов). Когда вызывается драйвер устройства, по старшему номеру индексируется массив входных точек в драйверы. По младшему номеру драйвер выбирает одно устройство из группы идентичных физических устройств.

Файл-каталог, в котором перечислены имена файлов, позволяет установить соответствие между именами и самими файлами. Каталоги образуют древовидную структуру. На каждый обычный файл или файл устройства могут иметься ссылки в различных узлах этой структуры. В непривилегированных программах запись в каталог не разрешена, но при наличии соответствующих разрешений они могут читать их. Дополнительных связей между каталогами нет.

Большое число системных каталогов UNIX использует для собственных нужд. Один из них, корневой каталог, является базой для всей структуры каталогов, и, «отталкиваясь» от него, можно найти все файлы. В других системных каталогах содержатся программы и команды, предоставляемые пользователям, а также файлы устройств.

Имена файлов задаются последовательностью имен каталогов, разделенных косой чертой (/) и приводящих к концевому узлу (листу) некоторого дерева. Если имя файла начинается с косой черты, то поиск по дереву начинается в корневом каталоге. Если же имя файла не имеет в начале косой черты, то поиск начинается с текущего каталога. Имена файлов, начинающиеся с символов../ (две точки и косая черта), подразумевают начало поиска в каталоге, родительском по отношению к текущему. Имя файла stuff (персонал) указывает на элемент stuff в текущем каталоге. Имя файла /work/alex/stuff приводит к поиску каталога work в корневом каталоге, затем к поиску каталога alex в каталоге work и, наконец, к поиску элемента stuff в каталоге alex. Сама по себе косая черта (/) обозначает корневой каталог. В приведенном примере нашла отражение типичная иерархическая структура файловой системы, например work может обозначать диск (устанавливаемый при работе пользователя), alex может быть каталогом пользователя, а stuff может принадлежать alex.

Файл, не являющийся каталогом, может встречаться в различных каталогах, возможно, под разными именами. Это называется *связыванием*. Элемент в каталоге,

относящийся к одному файлу, называется связью. В UNIX-системах все такие связи имеют равный статус. Файлы не принадлежат каталогам. Скорее, файлы существуют независимо от элементов каталогов, а связи в каталогах указывают на реальные (физические) файлы. Файл «исчезает», когда удаляется последняя связь, указывающая на него. Биты защиты, заданные в связях, могут отличаться от битов в исходном файле. Таким образом решается проблема избирательного ограничения на доступ к файлам.

С каждым поддерживаемым системой устройством ассоциируется один или большее число специальных файлов. Операции ввода-вывода для специальных файлов осуществляются так же, как и для обычных дисковых файлов с той лишь разницей, что эти операции активизируют соответствующие устройства. Специальные файлы обычно находятся в каталоге /dev. На специальные файлы могут указывать связи точно так же, как на обычные файлы.

От файловой системы не требуется, чтобы она целиком размещалась на том устройстве, где находится корень. Запрос от системы mount (на установку носителей и т. п.) позволяет встраивать в иерархию файлов файлы на сменных томах. Команда mount имеет несколько аргументов, но обязательных аргументов у стандартного варианта ее использования два: имя файла блочного устройства и имя каталога. В результате выполнения этой команды файловая подсистема, расположенная на указанном устройстве, подключается к системе таким образом, что ее содержимое заменяет собой содержимое заданного в команде каталога. Поэтому для *монтирования* соответствующего тома обычно используют пустой каталог. Команда umount выполняет обратную операцию — «отсоединяет» файловую систему, после чего диск с данными можно физически извлечь из системы. Например, для записи данных на дискету необходимо ее «подмонтировать», а после работы — «размонтировать».

Монтирование файловых систем позволяет получить единое логическое файловое пространство, в то время как реально отдельные каталоги с файлами могут находиться в разных разделах одного жесткого диска и даже на разных жестких дисках. Причем, что очень важно, сами файловые системы для монтируемых разделов могут быть разными. Например, при работе в системе Linux мы можем иметь часть разделов с файловой системой EXT2FS, а часть разделов — с файловой системой EXT3FS. Важно, чтобы ядро знало эти файловые системы.

Защита файлов

Защита файлов осуществляется при помощи номера, идентифицирующего пользователя, и десяти битов защиты — атрибутов доступа. Права доступа подразделяются на три типа: чтение (read), запись (write) и выполнение (execute). Эти типы прав доступа могут быть предоставлены трем классам пользователей: владельцу файла, группе, в которую входит владелец, и всем прочим пользователям. Девять из этих битов управляют защитой по чтению, записи и исполнению для владельца файла, других членов группы, в которую входит владелец, и всех других пользователей. Файл всегда связан с определенным пользователем — своим владельцем и с определенной группой, то есть у него есть уже известные нам идентификаторы

пользователя (UID) и группы (GID). Изменять права доступа к файлу разрешено только его владельцу. Изменить владельца файла может только суперпользователь, изменить группу — суперпользователь или владелец файла.

Программа, выполняющаяся в системе, всегда запускается от имени определенных пользователя и группы (обычно основной группы этого пользователя), но связь процессов с пользователями и группами организована сложнее. Различают идентификаторы доступа к файловой системе для пользователя (File System access User ID, FSUID) и для группы (File System access Group ID, FSGID), а также эффективные идентификаторы пользователя (Effective User ID, EUID) и группы (Effective Group ID, EGID). Кроме того, при доступе к файлам учитываются полномочия (capabilities), присвоенные самому процессу.

При создании файл получает идентификатор UID, совпадающий с FSUID процесса, который его создает, а также идентификатор GID, совпадающий с FSGID этого процесса.

Атрибуты доступа определяют, что разрешено делать с данным файлом данной категории пользователей. Имеется всего три операции: чтение, запись и выполнение.

При создании файла (или при создании еще одного имени для уже существующего файла) модифицируется не сам файл, а каталог, в котором появляются новые ссылки на узлы. Удаление файла заключается в удалении ссылки. Таким образом, право на создание или удаление файла — это право на запись в каталог.

Право на выполнение каталога интерпретируется как право на поиск в нем (прохождение через него). Оно позволяет обратиться к файлу с помощью пути, содержащему данный каталог, даже тогда, когда каталог не разрешено читать, и поэтому список всех его файлов недоступен.

Помимо трех названных основных атрибутов доступа существуют дополнительные, используемые в следующих случаях. Атрибуты SUID и SGID важны при запуске программы: они требуют, чтобы программа выполнялась не от имени запустившего ее пользователя (группы), а от имени владельца (группы) того файла, в котором она находится. Если файл программы имеет атрибут SUID (SGID), то идентификаторы FSUID и EUID (FSGID и EGID) соответствующего процесса не наследуются от процесса, запустившего его, а совпадают с UID (GID) файла. Благодаря этому пользователи получают возможность запустить системную программу, которая создает свои рабочие файлы в закрытых для них каталогах.

Кроме того, если процесс создает файл в каталоге, имеющем атрибут SGID, то файл получает GID не по идентификатору FSGID процесса, а по идентификатору GID каталога. Это удобно для коллективной работы: все файлы и вложенные каталоги[1] в каталоге автоматически оказываются принадлежащими одной и той же группе, хотя создавать их могут разные пользователи. Есть еще один атрибут SVTX, который нынче относится к каталогам. Он показывает, что из каталога, имеющего этот атрибут, ссылку на файл может удалить только владелец файла. Существуют две стандартные формы записи прав доступа — символьная и восьмеричная. Символь-

[1] Вложенные каталоги часто называют подкаталогами (subdirectory).

ная запись представляет собой цепочку из десяти знаков, первый из которых не относится собственно к правам, а обозначает тип файла. Используются следующие обозначения:

* - (дефис) — обычный файл;
* d — каталог;
* c — символьное устройство;
* b — блочное устройство;
* p — именованный канал (named pipe);
* s — сокет (socket)[1];
* I — символическая ссылка.

Далее следуют три последовательности, каждая из трех символов, соответствующие правам пользователя, группы и всех остальных. Наличие права на чтение обозначается символом r, на запись — символом w, на выполнение — символом x, отсутствие какого-либо права — символом - (дефис) в соответствующей позиции.

Наличие атрибута SUID (SGID) обозначается прописной буквой S в позиции права на выполнение для владельца (группы), если выполнение не разрешено, и строчной буквой s, если разрешено.

Восьмеричная запись — это шестизначное число, первые два знака которого обозначают тип файла и довольно часто опускаются, третья цифра — атрибуты GUID (4), SGID (2) и SVTX (1), оставшиеся три — права владельца, группы и всех остальных соответственно. Очевидно, что право на чтение можно представить числом 4, право на запись — числом 2, а право на выполнение — числом 1.

Например, стандартный набор прав доступа для каталога /tmp в символьной форме выглядит как drwxrwxrwx, а в восьмеричной — как 041777 (каталог; чтение, запись и поиск разрешены всем; установлен атрибут SVTX). А набор прав -r-S-xw-, или в числовом виде — 102412, означает, что это обычный файл, владельцу разрешается читать его, но не выполнять и не изменять, пользователям из группы (за исключением владельца) — выполнять (причем во время работы программа получит права владельца файла), но не читать и не изменять, а всем остальным — изменять, но не читать и не выполнять.

Большинство программ создают файлы с разрешением на чтение и запись для всех пользователей, а каталоги — с разрешением на чтение, запись и поиск для всех пользователей. Этот исходный набор атрибутов логически складывается с *пользовательской маской создания файла* (user file-creation mask, umask), которая обычно ограничивает доступ. Например, значения u=rwx, g=rwx, o=r-x для пользовательской маски следует понимать так: у владельца и группы остается полный набор прав, а всем остальным запрещается запись. В восьмеричном виде оно запишется как 002 (первая цифра — ограничения для владельца, вторая — для группы, третья — для

[1] Сокет — это понятие, связанное со стеком протоколов TCP/IP, который является «родным» для UNIX. Его следует понимать как некий адрес или порт, через который связываются удаленные программы.

остальных; запрещение чтения — 4, записи — 2, выполнения — 1). Владелец файла может изменить права доступа к нему командой chmod.

Взаимодействие между процессами

Операционная система UNIX в полной мере отвечает требованиям технологии клиент-сервер. Эта универсальная модель служит основой построения любых сколь угодно сложных систем, в том числе и сетевых. Разработчики СУБД, коммуникационных систем, систем электронной почты, банковских систем и т. д. во всем мире широко используют технологию клиент-сервер. Для построения программных систем, работающих по принципам модели «клиент-сервер», в UNIX существуют следующие механизмы:

* сигналы;
* семафоры;
* программные каналы;
* очереди сообщений;
* сегменты разделяемой памяти;
* вызовы удаленных процедур.

Многие из этих механизмов нам уже знакомы, поэтому рассмотрим их вкратце. Для более глубокого изучения этих вопросов можно рекомендовать известную работу [43].

Сигналы

Если рассматривать выполнение процесса на виртуальном компьютере, который предоставляется каждому пользователю, то в такой системе должна существовать система прерываний, отвечающая стандартным требованиям:

* обработка исключительных ситуаций;
* средства обработки внешних и внутренних прерываний;
* средства управления системой прерываний (маскирование и демаскирование).

Всем этим требованиям в UNIX отвечает механизм сигналов, который позволяет не только воспринимать и обрабатывать сигналы, но и порождать их и посылать на другие машины (процессы). Сигналы могут быть синхронными, когда инициатор сигнала — сам процесс, и асинхронными, когда инициатор сигнала — интерактивный пользователь, сидящий за терминалом. Источником асинхронных сигналов может быть также ядро, когда оно контролирует определенные состояния аппаратуры, рассматриваемые как ошибочные.

Сигналы можно рассматривать как простейшую форму взаимодействия между процессами. Они используются для передачи от одного процесса другому или от ядра ОС какому-либо процессу уведомления о возникновении определенного события.

Семафоры

Механизм семафоров, реализованный в UNIX-системах, является обобщением классического механизма семафоров, предложенного известным голландским спе-

циалистом профессором Дейкстрой. Семафор в операционной системе семейства UNIX состоит из следующих элементов:

* значения семафора;
* идентификатора процесса, который хронологически последним работал с семафором;
* числа процессов, ожидающих увеличения значения семафора;
* числа процессов, ожидающих нулевого значения семафора.

Для работы с семафорами имеются следующие три системных вызова:

* semget — создание и получение доступа к набору семафоров;
* semop — манипулирование значениями семафоров (именно этот системный вызов позволяет с помощью семафоров организовать синхронизацию процессов);
* semctl — выполнение разнообразных управляющих операций над набором семафоров.

Системный вызов semget имеет следующий синтаксис:

```
id = semget(key, count, flag).
```

Здесь параметры key и flag определяют ключ объекта и дополнительные флаги. Параметр count задает число семафоров в наборе семафоров, обладающих одним и тем же ключом. После этого индивидуальный семафор идентифицируется дескриптором набора семафоров и номером семафора в этом наборе. Если к моменту выполнения системного вызова semget набор семафоров с указанным ключом уже существует, то обращающийся процесс получит соответствующий дескриптор, но так и не узнает о реальном числе семафоров в группе (хотя позже это все-таки можно узнать с помощью системного вызова semctl).

Основным системным вызовом для манипулирования семафором является semop:

```
oldval = semop( id, oplist, count),
```

Здесь id — это ранее полученный дескриптор группы семафоров, oplist — массив описателей операций над семафорами группы, а count — размер этого массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора. Каждый элемент массива oplist имеет следующую структуру:

* номер семафора в указанном наборе семафоров;
* операция;
* флаги.

Если проверка прав доступа проходит нормально и указанные в массиве oplist номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов выполняется следующим образом. Для каждого элемента массива oplist значение соответствующего семафора изменяется в соответствии со значением поля операции, как показано ниже.

* Если значение поля операции положительно, то значение семафора увеличивается на единицу, а все процессы, ожидающие увеличения значения семафора, активизируются *(пробуждаются* — в терминологии UNIX).

* Если значение поля операции равно нулю и значение семафора также равно нулю, выбирается следующий элемент массива oplist. Если же значение поля операции равно нулю, а значение семафора отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, причем обратившийся процесс переводится в состояние ожидания *(засыпает* — в терминологии UNIX).
* Если значение поля операции отрицательно и его абсолютное значение меньше или равно значению семафора, то ядро прибавляет это отрицательное значение к значению семафора. Если в результате значение семафора стало нулевым, то ядро активизирует (пробуждает) все процессы, ожидающие нулевого значения этого семафора. Если же значение семафора оказывается меньше абсолютной величины поля операции, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора, и откладывает *(усыпляет)* текущий процесс до наступления этого события.

Интересно заметить, что основным поводом для введения массовых операций над семафорами было стремление дать программистам возможность избегать тупиковых ситуаций, возникающих в связи с семафорной синхронизацией. Это обеспечивается тем, что системный вызов semop, каким бы длинным он ни был (по причине потенциально неограниченной длины массива oplist), выполняется как атомарная операция, то есть во время выполнения semop ни один другой процесс не может изменить значение какого-либо семафора.

Наконец, среди флагов-параметров системного вызова semop может содержаться флаг с символическим именем IPC_NOWAIT, наличие которого заставляет ядро UNIX не блокировать текущий процесс, а лишь сообщать в ответных параметрах о возникновении ситуации, приведшей к блокированию процесса в случае отсутствия флага IPC_NOWAIT. Мы не будем обсуждать здесь возможности корректного завершения работы с семафорами при незапланированном завершении процесса; заметим только, что такие возможности обеспечиваются.

Системный вызов semctl имеет следующий формат:

```
semctl( id, number, cmd, arg);
```

Здесь id — это дескриптор группы семафоров, number — номер семафора в группе, cmd — код операции, arg — указатель на структуру, содержимое которой интерпретируется по-разному в зависимости от операции. В частности, с помощью вызова semctl можно уничтожить индивидуальный семафор в указанной группе. Однако детали этого системного вызова настолько громоздки, что лучше рекомендовать в случае необходимости обращаться к технической документации используемого варианта операционной системы.

Программные каналы

Мы уже познакомились с программными каналами в главе 7. Рассмотрим этот механизм еще раз, так сказать, в его исходном, изначальном толковании.

Программные *каналы* (pipes) в системе UNIX являются очень важным средством взаимодействия и синхронизации процессов. Теоретически программный канал

позволяет взаимодействовать любому числу процессов, обеспечивая дисциплину *FIFO* (First In First Out — первый пришедший первым и выбывает). Другими словами, процесс, читающий из программного канала, прочитает те данные, которые были записаны в программный канал раньше других. В традиционной реализации программных каналов для хранения данных использовались файлы. В современных версиях операционных систем семейства UNIX для реализации программных каналов применяются другие средства взаимодействия между процессами (в частности, очереди сообщений).

В UNIX различаются два вида программных каналов — именованные и неименованные. Именованный программный канал может служить для общения и синхронизации произвольных процессов, знающих имя данного программного канала и имеющих соответствующие права доступа. Неименованным программным каналом могут пользоваться только породивший его процесс и его потомки (необязательно прямые).

Для создания именованного программного канала (или получения к нему доступа) используется обычный файловый системный вызов open. Для создания же неименованного программного канала существует специальный системный вызов pipe (исторически более ранний). Однако после получения соответствующих дескрипторов оба вида программных каналов используются единообразно с помощью стандартных файловых системных вызовов read, write и close.

Системный вызов pipe имеет следующий синтаксис:

```
pipe(fdptr);
```

Здесь fdptr — это указатель на массив из двух целых чисел, в который после создания неименованного программного канала будут помещены дескрипторы, предназначенные для чтения из программного канала (с помощью системного вызова read) и записи в программный канал (с помощью системного вызова write). Дескрипторы неименованного программного канала — это обычные дескрипторы файлов, то есть такому программному каналу соответствуют два элемента таблицы открытых файлов процесса. Поэтому при последующих системных вызовах read и write процесс совершенно не обязан отличать случай использования программных каналов от случая использования обычных файлов (собственно, на этом и основана идея перенаправления ввода-вывода и организации конвейеров).

Для создания именованных программных каналов (или получения доступа к уже существующим каналам) используется обычный системный вызов open. Основным отличием от случая открытия обычного файла является то, что если именованный программный канал открывается для записи и ни один процесс не открыл тот же программный канал для чтения, то обращающийся процесс блокируется до тех пор, пока некоторый процесс не откроет данный программный канал для чтения. Аналогично обрабатывается открытие для чтения.

Запись данных в программный канал и чтение данных из программного канала (независимо от того, именованный он или не именованный) выполняются с помощью системных вызовов read и write. Отличие от случая использования обычных файлов состоит лишь в том, что при записи данные помещаются в начало канала, а при чтении выбираются (освобождая соответствующую область памяти) из конца канала.

Окончание работы процесса с программным каналом (независимо от того, именованный он или не именованный) производится с помощью системного вызова close.

Очереди сообщений

Для обмена данными между процессами используется механизм очередей сообщений, который поддерживается следующими системными вызовами:

* msgget — образование новой очереди сообщений или получение дескриптора существующей очереди;
* msgsnd — отправка сообщения (точнее, его постановка в указанную очередь сообщений);
* msgrcv — прием сообщения (точнее, выборка сообщения из очереди сообщений);
* msgctl — выполнение ряда управляющих действий.

Ядро хранит сообщения в виде связного списка (очереди), а дескриптор очереди сообщений является индексом в массиве заголовков очередей сообщений.

Системный вызов msgget имеет следующий синтаксис:

```
msgqid = msgget(key, flag);
```

Здесь параметры key и flag имеют то же значение, что и в вызове semget при запросе семафора.

При выполнении системного вызова msgget ядро UNIX-системы либо создает новую очередь сообщений, помещая ее заголовок в таблицу очередей сообщений и возвращая пользователю дескриптор вновь созданной очереди, либо находит элемент таблицы очередей сообщений, содержащий указанный ключ, и возвращает соответствующий дескриптор очереди.

Для отправки сообщения используется системный вызов msgsnd:

```
msgsnd(msgqid, msg, count, flag);
```

Здесь msg — указатель на структуру, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив (собственно сообщение); count — размер сообщения в байтах; flag — значение, которое определяет действия ядра при выходе за пределы допустимых размеров внутренней буферной памяти.

Для приема сообщения используется системный вызов msgrcv:

```
count = msgrcv(id, msg, maxcount, type, flag);
```

Здесь msg — указатель на структуру данных в адресном пространстве пользователя, предназначенную для размещения принятого сообщения; maxcount — размер области данных (массива байтов) в структуре msg; type — тип сообщения, которое требуется принять; flag — значение, которое указывает ядру, что следует предпринять, если в указанной очереди сообщений отсутствует сообщение с указанным типом. Возвращаемое значение системного вызова задает реальное число байтов, переданных пользователю.

Следующий системный вызов служит для опроса состояния описателя очереди сообщений, изменения его состояния (например, изменения прав доступа к очереди) и для уничтожения указанной очереди сообщений:

```
msgctl(id, cmd, mstatbuf);
```

Разделяемая память

Для работы с разделяемой памятью используются четыре системных вызова:

* shmget — создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;
* shmat — подключает сегмент с указанным дескриптором к виртуальной памяти обращающегося процесса;
* shmdt — отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;
* shmctl — служит для управления разнообразными параметрами, связанными с существующим сегментом.

После того как сегмент разделяемой памяти подключен к виртуальной памяти процесса, процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи, не прибегая к дополнительным системным вызовам.

Синтаксис системного вызова shmget выглядит следующим образом:

```
shmid = shmget( key, size, flag);
```

Параметр size определяет желаемый размер сегмента в байтах. Далее работа происходит по общим правилам. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является дескриптор существующего сегмента (и обратившийся процесс так и не узнает реального размера сегмента, хотя впоследствии его можно узнать с помощью системного вызова shmctl). В противном случае создается новый сегмент, размер которого не меньше, чем установленный в системе минимальный размер сегмента разделяемой памяти, и не больше, чем установленный максимальный размер. Создание сегмента не означает немедленного выделения для него основной памяти. Это действие откладывается до первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Аналогично, при выполнении последнего системного вызова отключения сегмента от виртуальной памяти соответствующая основная память освобождается.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову shmat:

```
virtaddr = shmat( id, addr, flags),
```

Здесь id — ранее полученный дескриптор сегмента; addr — требуемый процессу виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является реальный виртуальный адрес начала сегмента (его значение не обязательно совпадает со значением параметра addr). Если значением addr является нуль, ядро выбирает подходящий виртуальный адрес начала сегмента.

Для отключения сегмента от виртуальной памяти используется системный вызов shmdt:

```
shmdt(addr),
```

Здесь addr — виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный с помощью системного вызова shmat. При этом система гарантирует (опираясь на данные таблицы сегментов процесса), что указанный виртуальный адрес действительно является адресом начала разделяемого сегмента в виртуальной памяти данного процесса.

Для управления памятью служит системный вызов shmctl:

```
shmctl( id, and, shsstatbuf);
```

Параметр cmd идентифицирует требуемое конкретное действие, то есть ту или иную функцию. Наиболее важной является функция уничтожения сегмента разделяемой памяти, которое производится следующим образом. Если к моменту выполнения системного вызова ни один процесс не подключил сегмент к своей виртуальной памяти, то основная память, занимаемая сегментом, освобождается, а соответствующий элемент таблицы разделяемых сегментов объявляется свободным. В противном случае в элементе таблицы сегментов выставляется флаг, запрещающий выполнение системного вызова shmget по отношению к этому сегменту, но процессам, успевшим получить дескриптор сегмента, по-прежнему разрешается подключать сегмент к своей виртуальной памяти. При выполнении последнего системного вызова отключения сегмента от виртуальной памяти операция уничтожения сегмента завершается.

Вызовы удаленных процедур

Во многих случаях взаимодействие процессов соответствует отношениям клиент-сервер. Один из процессов (клиент) запрашивает у другого процесса (сервера) некоторую услугу (сервис) и не продолжает свое выполнение до тех пор, пока эта услуга не будет выполнена (то есть пока процесс-клиент не получит соответствующие результаты). Видно, что семантически такой режим взаимодействия эквивалентен вызову процедуры. Отсюда и соответствующее название — вызов удаленной процедуры (Remote Procedure Call, RPC). Другими словами, процесс обращается к процедуре, которая не принадлежит данному процессу. Она может находиться даже на другом компьютере. Операционная система UNIX по своей «идеологии» идеально подходит для того, чтобы быть сетевой операционной системой, на основе которой можно создавать распределенные системы и организовывать распределенные вычисления. Свойства переносимости позволяют создавать «операционно-однородные» сети, включающие разнородные компьютеры. Однако остается проблема разного представления данных в компьютерах разной архитектуры. Поэтому одной из основных идей RPC является автоматическое обеспечение преобразования форматов данных при взаимодействии процессов, выполняющихся на разнородных компьютерах.

Реализация механизма вызовов удаленных процедур (RPC) достаточно сложна, поскольку этот механизм должен обеспечить работу взаимодействующих процессов, находящихся на разных компьютерах. Если в случае обращения к процедуре, расположенной на том же компьютере, процесс общается с ней через стек или общие области памяти, то в случае удаленного вызова передача параметров процедуре превращается в передачу запроса по сети. Соответственно, и получение результата также осуществляется с помощью сетевых механизмов.

Вызов удаленных процедур включает следующие шаги [39].

1. Процесс-клиент осуществляет вызов локальной процедуры, которую называют *заглушкой* (stub). Задача этого модуля-заглушки — принять аргументы, преобразовать их в стандартную форму и сформировать сетевой запрос. Упаковка аргументов и создание сетевого запроса называется *сборкой* (marshalling).
2. Сетевой запрос пересылается на удаленную систему, где соответствующий модуль ожидает такой запрос и при его получении извлекает параметры вызова процедуры, то есть выполняет *разборку* (unmarshalling), а затем передает их серверу удаленной процедуры. После выполнения осуществляется обратная передача.

Операционная система Linux

Linux — это современная UNIX-подобная операционная система для персональных компьютеров и рабочих станций, удовлетворяющая стандарту POSIX.

Как известно, Linux — это свободно распространяемая версия UNIX-систем, которая первоначально разрабатывалась Линусом Торвальдсом (torvalds@kruuna.helsinki.fi) в университете Хельсинки (Финляндия). Он предложил разрабатывать ее совместно и выдвинул условие, согласно которому исходные коды являются открытыми, любой может их использовать и изменять, но при этом обязан оставить открытым и свой код, внесенный в тот или иной модуль системы. Все компоненты системы, включая исходные тексты, распространяются с лицензией на свободное копирование и установку для неограниченного числа пользователей.

Таким образом, система Linux была создана с помощью многих программистов и энтузиастов UNIX-систем, общающихся между собой через Интернет. К данному проекту добровольно подключились те, кто имеет достаточно навыков и способностей развивать систему. Большинство программ Linux разработаны в рамках проекта GNU из Free Software Foundation (Кембридж, штат Массачусетс). Но в него внесли свою лепту и многие программисты со всего мира.

Изначально система Linux создавалась как «самодельная» UNIX-подобная реализация для машин типа IBM PC с процессором i80386. Однако вскоре Linux стала настолько популярна и ее поддержало такое большое число компаний, что в настоящее время имеются реализации этой операционной системы практически для всех типов процессоров и компьютеров на их основе. На базе Linux создаются и встроенные системы, и суперкомпьютеры. Система поддерживает кластеризацию и большинство современных интерфейсов и технологий.

Большинство свойств Linux присущи другим реализациям UNIX, кроме того, имеются некоторые уникальные свойства. Этот раздел представляет собой лишь краткий обзор этих свойств.

Linux — это полноценная многозадачная многопользовательская операционная система (точно так же, как и все другие версии UNIX). Это означает, что одновременно много пользователей могут работать на одной машине, параллельно выполняя множество программ. Поскольку при работе за персональным компьютером практически никто не подключает к нему дополнительные терминалы (хотя это в

принципе возможно), пользователь просто имитирует работу за несколькими терминалами. В этом смысле можно говорить о *виртуальных терминалах.* По умолчанию пользователь регистрируется на первом терминале. При этом он получает примерно следующее сообщение:

```
Mandrake Linux release 9.0 (dolphin) for i586
Kernel 2.4.16-16mdk on an 1686 /tty1
Vienna login:
```

Здесь во второй строке слово tty1 означает, что пользователь сейчас взаимодействует с системой через первый виртуальный терминал. Собственно работа на нем возможна только после аутентификации — ввода своих учетного имени и пароля.

При желании открыть второй или последующий сеанс работы на соответствующем терминале, пользователь должен нажать комбинацию клавиш Alt+Fi, где i обозначает номер функциональной клавиши и одновременно номер соответствующего виртуального терминала. Всего Linux поддерживает до семи терминалов, причем седьмой терминал связан с графическим режимом работы и использованием одного из оконных менеджеров. Однако если пользователь работает в графическом режиме, то для перехода в один из алфавитно-цифровых терминалов следует воспользоваться комбинацией клавиш Ctrl+Alt+Fi. В каждом сеансе пользователь может запускать свои задачи.

Система Linux достаточно хорошо совместима с рядом стандартов для UNIX (насколько можно говорить о стандартизации UNIX) на уровне исходных текстов, включая IEEE POSIX.1, System V и BSD. Она и создавалась с расчетом на такую совместимость. Большинство свободно распространяемых через Интернет программ для UNIX может быть откомпилировано для Linux практически без особых изменений[1]. Кроме того, все исходные тексты для Linux, включая ядро, драйверы устройств, библиотеки, пользовательские программы и инструментальные средства распространяются свободно. Другие специфические внутренние черты Linux включают контроль работ по стандарту POSIX (используемый оболочками, такими как csh и bash), псевдотерминалы (pty), поддержку национальных и стандартных раскладок клавиатур динамически загружаемыми драйверами клавиатур.

Linux поддерживает различные типы файловых систем для хранения данных. Некоторые файловые системы, такие как EXT2FS, были созданы специально для Linux. Поддерживаются также другие типы файловых систем, например Minix-1 и Xenix. Кроме того, реализована система управления файлами на основе FAT, позволяющая непосредственно обращаться к файлам, находящимся в разделах с этой файловой системой. Поддерживается также файловая система ISO 9660 CD-ROM для работы с дисками CD-ROM. Имеются системы управления файлами и на томах с HPFS и NTFS, правда, они работают только на чтение файлов. Созданы варианты системы управления файлами и для доступа к FAT32; эта файловая система в операционной системе Linux называется VFAT.

[1] Справедливости ради следует заметить, что в последнее время в Linux наметились тенденции все большего отхода от принятых в семействе UNIX стандартов и увеличения количества различий в разных дистрибутивах Linux. Эти различия распространяются и на структуру каталогов файловой системы, что приводит к определенным проблемам при переносе прикладных программ из одной системы Linux в другую.

Linux, как и все UNIX-системы, поддерживает полный набор протоколов стека TCP/IP для сетевой работы. Программное обеспечение для работы в Интернет/интранет включает драйверы устройств для многих популярных сетевых адаптеров технологии Ethernet, протоколы SLIP (Serial Line Internet Protocol), PLIP (Parallel Line Internet Protocol), PPP (Point-to-Point Protocol), NFS (Network File System) и пр. Поддерживается весь спектр клиентов и услуг TCP/IP, таких как FTP, telnet, NNTP и SMTP.

Ядро Linux сразу было создано с учетом возможностей защищенного режима 32-разрядных процессоров 80386 и 80486 фирмы Intel. В частности, в Linux используется парадигма описания памяти в защищенном режиме и другие новые свойства процессоров с архитектурой ia32. Для защиты пользовательских программ друг от друга и операционной системы от них Linux работает исключительно в защищенном режиме[1], реализованном в процессорах фирмы Intel. В защищенном режиме только программный код, исполняющийся в нулевом кольце защиты, имеет непосредственный доступ к аппаратным ресурсам компьютера — памяти и устройствам ввода-вывода. Пользовательские и системные обрабатывающие программы работают в третьем кольце защиты. Они обращаются к аппаратным ресурсам компьютера исключительно через системные подпрограммы, функционирующие в нулевом кольце защиты. Таким образом, пользовательским программам предоставляются только те услуги, которые реализованы разработчиками операционной системы. При этом системные подпрограммы обеспечивают выполнение только тех функций, которые безопасны с точки зрения операционной системы.

Как и в классических UNIX-системах, Linux имеет макроядро, которое содержит уже известные нам три подсистемы. Ядро обеспечивает выделение каждому процессу отдельного адресного пространства, так что процесс не имеет возможности непосредственного доступа к данным других процессов и ядра операционной системы. Тем более что сегмент кода, сегмент данных и стек ядра располагаются в нулевом кольце защиты. Для обращения к физическим устройствам компьютера ядро вызывает соответствующие драйверы, управляющие аппаратурой компьютера. Поскольку драйверы функционируют в составе ядра, их код будет выполняться в нулевом (привилегированном) кольце защиты, и они могут получить прямой доступ к аппаратным ресурсам компьютера.

В отличие от старых версий UNIX, в которых задачи выгружались во внешнюю память на магнитных дисках целиком, ядро Linux использует аппаратную поддержку процессорами страничного механизма организации виртуальной памяти. Поэтому в Linux замещаются отдельные страницы. То есть с диска в память загружаются те виртуальные страницы образа, которые сейчас реально требуются, а неиспользуемые страницы выгружаются на диск в файл подкачки. Возможно разделение страниц кода, то есть использование одной страницы, физически уже один раз загруженной в память, несколькими процессами. Другими словами, реентерабельность кода, присущая всем UNIX-системам, осталась. В настоящее время имеются ядра для этой системы, оптимизированные для работы с процессорами Intel и AMD

[1] Напомним, что только в этом режиме процессоры с архитектурой ia32 используют 32-разрядную адресацию и имеют доступ ко всей оперативной памяти.

последнего поколения, хотя основные архитектурные особенности защищенного режима работы изменились мало. Уже разработаны ядра для работы с 64-разрядными процессорами от Intel и AMD.

Ядро также поддерживает универсальный пул памяти для пользовательских программ и дискового кэша. При этом для кэширования может использоваться вся свободная память, и наоборот, требуемый объем памяти, отводимой для кэширования файлов, уменьшается при работе больших программ. Этот механизм, называемый агрессивным кэшированием, позволяет более эффективно расходовать имеющуюся память и увеличить производительность системы.

Исполняемые программы задействуют динамически связываемые библиотеки (Dynamic Link Library, DLL), то есть эти программы могут совместно использовать библиотеку, представленную одним физическим файлом на диске. Это позволяет занимать меньше места на диске исполняемым файлам, особенно тем, которые многократно вызывают библиотечные функции. Есть также статические связываемые библиотеки для тех, кто желает пользоваться отладкой на уровне объектных кодов или иметь «полные» исполняемые программы, не нуждающиеся в разделяемых библиотеках. В Linux разделяемые библиотеки динамически связываются во время выполнения, позволяя программисту заменять библиотечные модули своими собственными.

Операционная система FreeBSD

Помимо Linux к свободно распространяемым операционным системам семейства UNIX следует отнести FreeBSD. Принципиальное и самое важное различие между этими операционными системами заключается в том, что согласно принятому соглашению в системы Linux каждый может внести свои изменения, но при этом обязан также сделать свой код открытым. Не все компании на это согласны. Многие предпочитают воспользоваться исходными текстами и готовыми решениями, но не открывать секретов своего программного обеспечения, сделанного с помощью использованного открытого кода. Поэтому в настоящее время сложилась такая ситуация, что имеется уже несколько десятков компаний, занимающихся созданием дистрибутивов для этой операционной системы. Каждая компания, подготавливающая дистрибутив, помимо собственно операционной системы добавляет к нему свой инсталлятор[1], утилиты, в том числе менеджер пакетов программ, конфигураторы и, наконец, большой набор прикладного программного обеспечения. При этом она привносит в систему свои изменения, не согласуя их с другими (за исключением самого ядра, работу над которым по-прежнему курирует Торвальдс). Таким образом, можно констатировать, что у системы Linux как совокупности собственно операционной системы и программного обеспечения, поставляемого с ней, нет единого координатора. С одной стороны, это приводит к заметному прогрессу системы, она быстро реагирует на новые устройства и технологии. Сейчас уже на компьютере с Linux можно играть в современные трехмерные игры, просматри-

[1] Программа установки программного обеспечения на компьютер, в том числе программа установки операционной системы (от англ. «install» — установить).

вать видеофильмы, кодированные в соответствии с самыми современными форматами, слушать и писать музыку и т. д. С другой стороны, пользователи сталкиваются с проблемами переносимости приложений, созданных для этих (и других UNIX-подобных) систем, поскольку нет единого координатора.

В противоположность Linux операционная система FreeBSD имеет такого координатора — это университет в Беркли, Калифорния. Любой может изучить тексты кодов этой операционной системы и предложить внести в нее свои изменения, но это не означает, что так и будет сделано, даже если изменения разумны. Только координирующая группа BSD имеет на это право.

Итак, FreeBSD — это тоже UNIX-подобная операционная система с открытым исходным кодом. Однако несмотря на то, что она родилась раньше и в той же мере бесплатна, что и Linux, многие о ней даже не слышали. Дело в том, что эта операционная система не имеет такой раскрученной рекламы, как проект Linux, хотя история BSD уходит корнями в более далекие годы. При этом необходимо заметить, что в плане производительности, стабильности, качества кода специалисты практически единодушно отдают предпочтение операционной системе FreeBSD. В частности, еще одним важным отличием FreeBSD от Linux является то, что ядро FreeBSD построено по принципам микроядерных операционных систем, тогда как Linux — это макроядерная операционная система.

Сетевая операционная система реального времени QNX

Вспомним основные принципы, обязательная реализация которых позволяет создавать операционные системы реального времени (ОСРВ). Первым обязательным требованием к архитектуре операционной системы реального времени является *многозадачность* в истинном смысле этого слова. Очевидно, что варианты с псевдомногозадачностью (а точнее, с невытесняющей многозадачностью) в системах Windows 3.X или Novell NetWare неприемлемы, поскольку они допускают возможность блокировки или даже полного развала системы одним неправильно работающим процессом. Для предотвращения блокировок вычислений ОСРВ должна использовать квантование времени (то есть использовать вытесняющую, а не кооперативную многозадачность), что сделать достаточно просто. Вторая проблема — организация *надежных вычислений* — может быть эффективно решена за счет специальных аппаратных возможностей процессора. При построении системы для работы на персональных компьютерах типа IBM PC для этого необходимы процессоры типа Intel 80386 и выше, чтобы иметь возможность организовать функционирование операционной системы в защищенном (32-разрядном) режиме работы процессора. Для эффективного обслуживания прерываний операционная система должна использовать алгоритм диспетчеризации, обеспечивающий *вытесняющее планирование, основанное на приоритетах.* Наконец, крайне желательна эффективная поддержка *сетевых коммуникаций* и наличие развитых механизмов *взаимодействия между процессами,* поскольку реальные технологические системы обычно управляются целым комплексом компьютеров и/или контроллеров. Весь-

ма желательно также, чтобы операционная система поддерживала многопоточность (не только мультипрограммный, но и мультизадачный режимы) и симметричную мультипроцессорность. И наконец, при соблюдении всех перечисленных условий операционная система должна быть *способна работать на ограниченных аппаратных ресурсах,* поскольку одна из ее основных областей применения — встроенные системы. К сожалению, данное условие обычно реализуется путем простого урезания стандартных сервисных средств.

Операционная система QNX является мощной операционной системой, разработанной для процессоров с архитектурой ia32. Она позволяет проектировать сложные программные комплексы, работающие в реальном времени как на отдельном компьютере, так и в локальной вычислительной сети. Встроенные средства QNX обеспечивают поддержку многозадачного режима на одном компьютере и взаимодействие параллельно выполняемых задач на разных компьютерах, работающих в среде локальной вычислительной сети. Таким образом, эта операционная система хорошо подходит для построения распределенных систем.

Основным языком программирования в системе является C. Основная операционная среда соответствует стандарту POSIX. Это позволяет с небольшими доработками переносить ранее разработанное программное обеспечение в QNX для организации их работы в среде распределенной обработки.

Операционная система QNX, будучи сетевой и мультизадачной, в то же время является многопользовательской (многотерминальной). Кроме того, она масштабируема. С точки зрения пользовательского интерфейса и интерфейса прикладного программирования она очень похожа на UNIX, поскольку выполняет требования стандарта POSIX. Однако QNX — это не версия UNIX, хотя почему-то многие так считают. Система QNX была разработана, что называется, «с нуля» канадской фирмой QNX Software Systems Limited в 1989 году по заказу Министерства обороны США, причем на совершенно иных архитектурных принципах, нежели использовались при создании операционной системы UNIX.

QNX была первой коммерческой операционной системой, построенной на принципах микроядра и обмена сообщениями. Система реализована в виде совокупности независимых (но взаимодействующих путем обмена сообщениями) процессов различного уровня (менеджеры и драйверы), каждый из которых реализует определенный вид услуг. Эти идеи позволили добиться нескольких важнейших преимуществ. Вот как об этом написано на сайте, посвященном операционной системе QNX [14].

* *Предсказуемость* означает применимость системы к задачам жесткого реального времени. QNX является операционной системой, которая дает полную гарантию того, что процесс с наивысшим приоритетом начнет выполняться практически немедленно, и критически важное событие (например, сигнал тревоги) никогда не будет потеряно. Ни одна версия UNIX не может достичь подобного качества, поскольку нереентерабельный код ядра слишком велик. Любой системный вызов из обработчика прерывания в UNIX может привести к непредсказуемой задержке (то же самое можно сказать про Windows NT).

* *Масштабируемость* и *эффективность* достигаются оптимальным использованием ресурсов и означают применимость QNX для встроенных (embedded) систем. В данном случае мы не увидим в каталоге /dev множества файлов, соответствующих ненужным драйверам, что характерно для UNIX-систем. Драйверы и менеджеры можно запускать и удалять (кроме файловой системы, что очевидно) динамически, просто из командной строки. Мы можем иметь только те услуги, которые нам реально нужны, причем это не требует серьезных усилий и не порождает проблем.
* *Расширяемость* и *надежность* обеспечиваются одновременно, поскольку написанный драйвер не нужно компилировать в ядро, рискуя вызвать нестабильность системы. Менеджеры ресурсов (служба логического уровня) работают в третьем кольце защиты, и вы можете добавлять свои менеджеры, не опасаясь за систему. Драйверы работают в первом кольце и могут вызвать проблемы, но не фатального характера. Кроме того, их достаточно просто писать и отлаживать.
* *Быстрый сетевой протокол FLEET*[1] прозрачен для обмена сообщениями, автоматически обеспечивает отказоустойчивость, балансирование нагрузки и маршрутизацию между альтернативными путями доступа.
* *Компактная графическая подсистема Photon,* построенная на тех же принципах модульности, что и сама операционная система, позволяет получить полнофункциональный интерфейс GUI (расширенный интерфейс Motif), работающий вместе с POSIX-совместимой операционной системой всего в 4 Мбайт памяти, начиная с i80386 процессора.

Архитектура системы QNX

Итак, QNX — это операционная система реального времени для персональных компьютеров, позволяющая эффективно организовать распределенные вычисления. В системе реализована концепция связи между задачами на основе сообщений, посылаемых от одной задачи к другой, причем задачи эти могут решаться как на одном и том же компьютере, так и на разных, но связанных между собой локальной вычислительной сетью. Реальное время и концепция связи между процессами посредством сообщений оказывают решающее влияние и на разрабатываемое для операционной системы QNX программное обеспечение, и на программиста, стремящегося с максимальной выгодой использовать преимущества системы.

Микроядро операционной системы QNX имеет объем всего в несколько десятков килобайтов (в одной из версий — 10 Кбайт, в другой — менее 32 Кбайт, хотя есть вариант и на 46 Кбайт), то есть это одно из самых маленьких ядер среди всех существующих операционных систем. В этом объеме помещаются [26]:

* механизм передачи сообщений между процессами IPC (Inter Process Communication — взаимодействие между процессами);
* редиректор (redirector) прерываний;

[1] Это фирменная технология, о которой несколько более подробно рассказано далее.

* блок планирования выполнения задач (иначе говоря, диспетчер задач);
* сетевой интерфейс для перенаправления сообщений (менеджер Net).

Механизм IPC обеспечивает пересылку сообщений между процессами и является одной из важнейших частей операционной системы, так как все взаимодействие между процессами, в том числе и системными, происходит через сообщения. Сообщение в операционной системе QNX — это последовательность байтов произвольной длины (0-65 535 байт) произвольного формата. Протокол обмена сообщениями может выглядеть, например, таким образом. Задача блокируется для ожидания сообщения. Другая задача посылает первой сообщение и при этом блокируется сама, ожидая ответа. Первая задача деблокируется, обрабатывает сообщение и отвечает, деблокируя вторую задачу.

Сообщения и ответы, пересылаемые между процессами при их взаимодействии, находятся в теле отправляющего их процесса до того момента, когда они могут быть приняты. Это означает, что, с одной стороны, снижается вероятность повреждения сообщения в процессе передачи, а с другой — уменьшается объем оперативной памяти, необходимый для работы ядра. Кроме того, становится меньше пересылок из памяти в память, что разгружает процессор. Особенностью процесса передачи сообщений является то, что в сети, состоящей из нескольких компьютеров, работающих под управлением QNX, сообщения могут прозрачно передаваться процессам, выполняющимся на любом из узлов. Определены в QNX еще и два дополнительных метода передачи сообщений — *метод представителей* (proxy) и *метод сигналов* (signal).

Представители используются в случаях, когда процесс должен передать сообщение, но не должен при этом блокироваться на передачу. Тогда вызывается функция qnx_proxy_attach() и создается представитель. Он накапливает в себе сообщения, которые должны быть доставлены другим процессам. Любой процесс, знающий идентификатор представителя, может вызвать функцию Trigger(), после чего будет доставлено первое в очереди сообщение. Функция Trigger() может вызываться несколько раз, и каждый раз представитель будет доставлять следующее сообщение. При этом представитель содержит буфер, в котором может храниться до 65 535 сообщений.

Как известно, механизм сигналов уже давно используется в операционных системах, в том числе и в UNIX. Операционная система QNX также поддерживает множество сигналов, совместимых с POSIX, большое количество сигналов, традиционно использовавшихся в UNIX (поддержка этих сигналов требуется для совместимости с переносимыми приложениями, ни один из системных процессов QNX их не генерирует), а также несколько сигналов, специфичных для самой системы QNX. По умолчанию любой сигнал, полученный процессом, приводит к завершению процесса (кроме нескольких сигналов, которые по умолчанию игнорируются). Но процесс с приоритетом уровня суперпользователя может защититься от нежелательных сигналов. В любом случае процесс может содержать обработчик для каждого возможного сигнала. Сигналы удобно рассматривать как разновидность программных прерываний.

Редиректор прерываний является частью ядра и занимается перенаправлением аппаратных прерываний в связанные с ними процессы. Благодаря такому подходу

возникает один побочный эффект — с аппаратной частью компьютера работает ядро, оно перенаправляет прерывания процессам — обработчикам прерываний. Обработчики прерываний обычно встроены в процессы, хотя каждый из них исполняется асинхронно с процессом, в который он встроен. Обработчик исполняется в контексте процесса и имеет доступ ко всем глобальным переменным процесса. При работе обработчика прерываний прерывания разрешены, но обработчик приостанавливается только в том случае, если произошло более высокоприоритетное прерывание. Если это позволяется аппаратной частью, к одному прерыванию может быть подключено несколько обработчиков, каждый из которых получит управление при возникновении прерывания.

Этот механизм позволяет пользователю избегать работы с аппаратным обеспечением напрямую и тем самым избегать конфликтов между различными процессами, работающими с одним и тем же устройством. Для обработки сигналов от внешних устройств чрезвычайно важно минимизировать время между возникновением события и началом непосредственной его обработки. Этот фактор существен в любой области применения: от работы терминальных устройств до обработки высокочастотных сигналов.

Блок планирования выполнения задач обеспечивает многозадачность. В этом плане операционная система QNX предоставляет разработчику огромный простор для выбора той дисциплины выделения ресурсов процессора задаче, которая обеспечит наиболее подходящие условия для выполнения критически важных приложений, а обычным приложениям обеспечит такие условия, при которых они будут выполняться за разумное время, не мешая работе критически важных приложений.

К выполнению своих функций как диспетчера ядро приступает в следующих случаях:

* какой-либо процесс вышел из блокированного состояния;
* истек квант времени для процесса, владеющего центральным процессором;
* работающий процесс прерван каким-либо событием.

Диспетчер выбирает процесс для запуска среди неблокированных процессов в порядке значений их приоритетов в диапазоне от 0 (наименьший) до 31 (наибольший). Обслуживание каждого из процессов зависит от метода его диспетчеризации (приоритет и метод диспетчеризации могут динамически меняться во время работы). В QNX существуют три метода диспетчеризации:

* очередь (First In First Out, FIFO) — раньше пришедший процесс раньше обслуживается;
* карусель (Round Robin, RR) — процессу выделяется определенный квант времени для работы, после чего процессор предоставляется следующему процессу;
* адаптивный метод (используется чаще других).

Метод FIFO наиболее близок к невытесняющей многозадачности. То есть процесс выполняется до тех пор, пока он не перейдет в состояние ожидания сообщения, в состояние ожидания ответа на сообщение или не отдаст управление ядру. При переходе в одно из таких состояний процесс помещается последним в очередь про-

цессов с таким же уровнем приоритета, а управление передается процессу с наибольшим приоритетом.

В методе RR все происходит так же, как и в предыдущем, с той разницей, что период, в течение которого процесс может работать без перерыва, ограничивается неким квантом времени.

Процесс, работающий в соответствии с адаптивным методом, ведет себя следующим образом:

* если процесс полностью использует выделенный ему квант времени, а в системе есть готовые к исполнению процессы с тем же уровнем приоритета, его приоритет снижается на 1;
* если процесс с пониженным приоритетом остается необслуженным в течение секунды, его приоритет увеличивается на 1;
* если процесс блокируется, ему возвращается исходное значение приоритета.

По умолчанию процессы запускаются в режиме адаптивной многозадачности. В этом же режиме работают все системные утилиты QNX. Процессы, работающие в разных режимах многозадачности, могут одновременно находиться в памяти и исполняться. Важный элемент реализации многозадачности — приоритет процесса. Обычно приоритет процесса устанавливается при его запуске. Но есть дополнительная возможность, называемая *вызываемым клиентом приоритетом*. Как правило, она реализуется для серверных процессов (исполняющих запросы на какое-либо обслуживание). При этом приоритет процесса-сервера устанавливается только на время обработки запроса и становится равным приоритету процесса-клиента.

Сетевой интерфейс в операционной системе QNX является неотъемлемой частью ядра. Он, конечно, взаимодействует с сетевым адаптером через сетевой драйвер, но базовые сетевые службы реализованы на уровне ядра. При этом передача сообщения процессу, находящемуся на другом компьютере, ничем не отличается с точки зрения приложения от передачи сообщения процессу, выполняющемуся на том же компьютере. Благодаря такой организации сеть превращается в однородную вычислительную среду. При этом для большинства приложений не имеет значения, с какого компьютера они были запущены, на каком исполняются и куда поступают результаты их работы.

Все службы операционной системы QNX, не реализованные непосредственно в ядре, работают как обычные стандартные процессы в полном соответствии с основными концепциями микроядерной архитектуры. С точки зрения операционной системы эти системные процессы ничем не отличаются от всех остальных. Как, впрочем, и драйверы устройств. Единственное, что нужно сделать, чтобы новый драйвер устройства стал частью операционной системы, — изменить конфигурационный файл системы так, чтобы драйвер запускался при загрузке.

Основные механизмы организации распределенных вычислений

QNX является сетевой операционной системой, которая позволяет организовать эффективные распределенные вычисления. Для этого на каждой машине, называ-

емой узлом, помимо ядра и менеджера процессов должен быть запущен уже упомянутый ранее менеджер Net. Менеджер Net не зависит от аппаратной реализации сети. Эта аппаратная независимость обеспечивается за счет сетевых драйверов. В операционной системе QNX имеются драйверы для сетей с различными технологиями: Ethernet и FastEthernet, Arcnet, IBM Token Ring и др. Кроме того, имеется возможность организации сети через последовательный канал или модем.

В QNX версии 4 полностью реализовано встроенное сетевое взаимодействие типа «точка-точка». Например, сидя за машиной *A,* вы можете скопировать файл с гибкого диска, подключенного к машине *B,* на жесткий диск, подключенный к машине *C.* По существу, сеть из машин с операционными системами QNX действует как один мощный компьютер. Любые ресурсы (модемы, диски, принтеры) могут быть добавлены к системе простым их подключением к любой машине в сети. QNX обеспечивает возможность одновременной работы в сетях Ethernet, Arcnet, Serial и Token Ring, более одного пути для связи и балансировку нагрузки в сетях. Если кабель или сетевая плата выходит из строя и связь через эту сеть прекращается, система автоматически перенаправит данные через другую сеть. Все это происходит в режиме подключения (on-line), предоставляя пользователю автоматическую сетевую избыточность и увеличивая эффективность взаимодействия во всей системе.

Каждому узлу в сети соответствует уникальный целочисленный идентификатор — логический номер узла. Любой поток выполнения в сети QNX имеет прозрачный доступ (при наличии достаточных привилегий) ко всем ресурсам сети; то же самое относится и к взаимодействию потоков. Для взаимодействия потоков, находящихся на разных узлах сети, используются те же самые вызовы ядра, что и для потоков, выполняемых на одном узле. В том случае, если потоки находятся на разных узлах сети, ядро переадресует запрос менеджеру сети. Для обмена в сети используется надежный и эффективный протокол транспортного уровня FLEET. Каждый из узлов может принадлежать одновременно нескольким QNX-сетям. В том случае, если сетевое взаимодействие может быть реализовано несколькими путями, для передачи выбирается менее загруженная и более скоростная сеть.

Сетевое взаимодействие является узким местом в большинстве операционных систем и обычно создает значительные проблемы для систем реального времени. Для того чтобы обойти это препятствие, разработчики операционной системы QNX создали собственную специальную сетевую технологию FLEET и соответствующий протокол транспортного уровня FTL (FLEET Transport Layer). Этот протокол не базируется ни на одном из распространенных сетевых протоколов вроде IPX или NetBios и обладает рядом качеств, которые делают его уникальным. Основные его качества зашифрованы в аббревиатуре FLEET, которая расшифровывается следующим образом:

* Fault-Tolerant Networking — QNX может одновременно использовать несколько физических сетей, при выходе из строя любой из них данные будут «на лету» перенаправлены через другую сеть;
* Load-Balancing on the Fly — при наличии нескольких физических соединений QNX автоматически распараллеливает передачу пакетов по соответствующим сетям;

* Efficient Performance — специальные драйверы, разрабатываемые фирмой QSSL для широкого спектра оборудования, позволяют использовать это оборудование с максимальной эффективностью;
* Extensable Architecture — любые новые типы сетей могут быть поддержаны путем добавления соответствующих драйверов;
* Transparent Distributed Processing — благодаря отсутствию разницы между передачей сообщений в пределах одного узла и между узлами нет необходимости вносить какие-либо изменения в приложения, для того чтобы они могли взаимодействовать через сеть.

Благодаря технологии FLEET сеть компьютеров с операционными системами QNX фактически можно представлять как один виртуальный суперкомпьютер. Все ресурсы любого из узлов сети автоматически доступны другим, и для этого не нужно создавать никаких дополнительных механизмов с использованием технологии RPC. Это значит, что любая программа может быть запущена на любом узле, причем ее входные и выходные потоки могут быть направлены на любое устройство на любых других узлах [18].

Например, утилита make в операционной системе QNX автоматически распараллеливает компиляцию пакетов из нескольких модулей на все доступные узлы сети, а затем собирает исполняемый модуль по мере завершения компиляции на узлах. Специальный драйвер, входящий в комплект поставки, позволяет использовать для сетевого взаимодействия любое устройство, с которым может быть ассоциирован файловый дескриптор, например последовательный порт, что открывает возможности для создания глобальных сетей.

Достигаются все эти удобства за счет того, что поддержка сети частично обеспечивается и микроядром (специальный код в его составе позволяет операционной системе QNX фактически объединять все микроядра в сети в одно ядро). Разумеется, за такие возможности приходится платить тем, что мы не можем получить драйвер для какой-либо сетевой платы от кого-либо еще, кроме фирмы QSSL, то есть использоваться может только то оборудование, которое уже поддерживается. Однако ассортимент такого оборудования достаточно широк и периодически пополняется новейшими устройствами.

Когда ядро получает запрос на передачу данных процессу, находящемуся на удаленном узле, он переадресовывает этот запрос менеджеру Net, в подчинении которого находятся драйверы всех сетевых карт. Имея перед собой полную картину состояния всего сетевого оборудования, Net может отслеживать состояние каждой сети и динамически перераспределять нагрузку между ними. В случае, когда одна из сетей выходит из строя, поток данных автоматически перенаправляется в другую доступную сеть, что очень важно при построении высоконадежных систем. Кроме поддержки собственного протокола, Net обеспечивает передачу пакетов TCP/IP, SMB (Server Message Block)[1] и многих других, используя то же сете-

[1] Сетевая технология взаимодействия клиента и сервера, разработанная фирмой IBM и активно используемая компанией Microsoft в своих операционных системах. В последнее время компания Microsoft стала называть ее CIFS (Common Internet File System).

вое оборудование. При этом производительность компьютеров с операционной системой QNX в сети приближается к производительности аппаратного обеспечения — настолько малы задержки, вносимые операционной системой.

При проектировании системы реального времени, как правило, необходимо обеспечить одновременное выполнение нескольких приложений. В QNX/Neutrino[1] параллельность выполнения достигается за счет использования *потоковой модели POSIX,* в которой процессы в системе представляются в виде совокупности потоков выполнения. Поток является минимальной единицей выполнения и диспетчеризации для ядра Neutrino; процесс определяет адресное пространство для потоков. Каждый процесс состоит минимум из одного потока. Операционная система QNX предоставляет богатый набор функций для синхронизации потоков. В отличие от потоков, само ядро не подлежит диспетчеризации. Код ядра исполняется только в том случае, когда какой-нибудь поток вызывает функцию ядра, или при обработке аппаратного прерывания.

Напомним, что операционная система QNX базируется на концепции *передачи сообщений.* Передачу и диспетчеризацию сообщений осуществляет ядро системы. Кроме того, ядро управляет временными прерываниями. Выполнение остальных функций обеспечивается задачами-администраторами. Программа, желающая создать задачу, посылает сообщение администратору задач (модуль task) и блокируется для ожидания ответа. Если новая задача должна выполняться одновременно с порождающей ее задачей, администратор задач task создает ее и, отвечая, выдает порождающей задаче идентификатор созданной задачи. В противном случае никакого сообщения не посылается до тех пор, пока новая задача не закончится сама по себе. Тогда в ответе администратора задач будут содержаться конечные характеристики закончившейся задачи.

Сообщения различаются количеством данных, которые передаются от одной задачи точно к другой задаче. Данные копируются из адресного пространства первой задачи в адресное пространство второй, и выполнение первой задачи приостанавливается до тех пор, пока вторая задача не вернет ответное сообщение. В действительности обе задачи кратковременно взаимодействуют во время выполнения передачи. Ничто, кроме длины сообщения (максимальная длина может достигать 64 Кбайт), не заботит QNX при передаче сообщения. Существует несколько протоколов, которые могут быть использованы для этой цели.

Основные операции над сообщениями: *послать, получить* и *ответить,* а также несколько их вариантов для обработки специальных ситуаций. Получатель всегда идентифицируется своим идентификатором задачи, хотя существуют способы ассоциировать имена с идентификатором задачи. Наиболее интересные варианты операций включают в себя возможность получать (копировать) только первую часть сообщения, а затем получать оставшуюся часть такими кусками, какие потребуются. Это может быть полезным, поскольку позволяет сначала узнать длину сообщения, а затем динамически распределить принимающий буфер. Если необходимо задержать ответное сообщение до тех пор, пока не будет получено и обра-

[1] Neutrino — один из проектов микроядерной ОС.

ботано другое сообщение, то чтение первых нескольких байтов дает вам компактный «обработчик», через который позже можно получить доступ ко всему сообщению. Таким образом, задача оказывается избавленной от необходимости хранить в себе большое количество буферов.

Другие функции позволяют программе получать сообщения только тогда, когда она уже ожидает их приема, а не блокироваться до тех пор, пока не прибудет сообщение. Можно также транслировать сообщение другой задаче без изменения идентификатора передатчика. Задача, которая транслировала сообщение, в транзакции невидима.

Кроме того, операционная система QNX обеспечивает объединение сообщений в структуру данных, называемую очередью. *Очередь сообщений* — это просто область данных в третьей, отдельной задаче, которая временно принимает передаваемое сообщение и немедленно отвечает передатчику. В отличие от стандартной передачи сообщений, передатчик немедленно освобождается для того, чтобы продолжить свою работу. Задача администратора очереди — хранить в себе сообщение до тех пор, пока приемник не будет готов прочитать его; делает он это, запрашивая сообщение у администратора очереди. Любое количество сообщений (ограничено только возможностью памяти) может храниться в очереди. Сообщения хранятся и передаются в том порядке, в котором они были приняты. Может быть создано любое количество очередей. Каждая очередь идентифицируется своим именем.

Помимо сообщений и очередей в операционной системе QNX для взаимодействия задач и организации распределенных вычислений имеются так называемые *порты,* которые позволяют формировать сигнал одного конкретного условия и механизм исключений, о котором мы уже упоминали ранее.

Порт подобен флагу, известному всем задачам на одном и том же узле (но не на разных узлах). Он имеет только два состояния, которые могут трактоваться как «присоединить» и «освободить», хотя пользователь может интерпретировать их по-своему, например «занят» и «доступен». Порты используются для быстрой простой синхронизации между задачей и обработчиком прерываний устройства. Они нумеруются от нуля до 32 максимум (на некоторых типах узлов возможно и больше). Первые 20 номеров зарезервированы для операционной системы.

С портом может быть выполнено три операции:

* присоединить порт,
* отсоединить порт,
* послать сигнал в порт.

Одновременно к порту может быть присоединена только одна задача. Если другая задача попытается «отсоединиться» от того же самого порта, то произойдет отказ при вызове функции, и управление вернется к задаче, которая в настоящий момент присоединена к этому порту. Это самый быстрый способ обнаружить идентификатор другой задачи, подразумевая, что задачи могут договориться использовать один номер порта. Напомним, что все рассматриваемые задачи должны находиться на одном и том же узле. При работе нескольких узлов специальные функции обеспечивают большую гибкость и эффективность.

Любая задача может посылать сигнал в любой порт независимо от того, была она присоединена к нему или нет (предпочтительно, чтобы не была). Сигнал подобен неблокирующей передаче пустого сообщения. То есть передатчик не приостанавливается, а приемник не получает какие-либо данные; он только отмечает, что конкретный порт изменил свое состояние.

Задача, присоединенная к порту, может ожидать прибытия сигнала или может периодически читать порт. Система QNX хранит информацию о сигналах, передаваемых в каждый порт, и уменьшает счетчик после каждой операции «приема» сигнала («чтение» возвращает счетчик и устанавливает его в нуль). Сигналы всегда принимают перед сообщениями, давая им тем самым больший приоритет над сообщениями. В этом смысле сигналы часто используются обработчиками прерываний для того, чтобы оповестить задачу о внешних (аппаратных) событиях. Действительно, обработчики прерываний не имеют возможности посылать сообщения и должны использовать сигналы.

В отличие от описанных выше методов, которые строго синхронизируются, *исключения* обеспечивают асинхронное взаимодействие. То есть исключение может прервать нормальное выполнение потока задачи. Они, таким образом, являются аварийными событиями. Операционная система QNX резервирует для себя 16 исключений, чтобы оповещать задачи о прерываниях с клавиатуры, нарушении памяти и подобных необычных ситуациях. Остальные 16 исключений могут быть определены и использованы прикладными задачами.

Системная функция может быть вызвана для того, чтобы позволить задаче реализовать собственный механизм обработки исключений и во время возникновения исключения выполнять свою внутреннюю функцию.

Заметим, что функция исключения задачи вызывается асинхронно операционной системой, а не самой задачей. Поэтому исключения могут негативно повлиять на операции (например, передачу сообщений), которые выполняются в это же время. Обработчики исключений должны быть написаны очень аккуратно.

Одна задача может установить одно или несколько исключений для другой задачи. Эти исключения могут быть комбинацией системных исключений и исключений, определяемых приложениями, обеспечивая другие возможности для межзадачного взаимодействия.

Благодаря такому свойству QNX, как возможность обмена посланиями между задачами и узлами сети, программы не заботятся о конкретном размещении ресурсов в сети. Это свойство придает системе необычную гибкость. Так, узлы могут произвольно добавляться в систему и изыматься из системы, не затрагивая системные программы. QNX имеет эту конфигурационную независимость благодаря концепции *виртуальных задач*. У виртуальных задач непосредственный код и данные, будучи на одном из удаленных узлов, возникают и ведут себя так, как если бы они были локальными задачами какого-то узла со всеми их атрибутами и привилегиями. Программа, посылающая сообщение в сеть, никогда не направляет его точно. Сначала она открывает *виртуальный канал*. *Виртуальный канал* связывает между собой все виртуальные задачи. На обоих концах такой связи имеются буферы, которые позволяют хранить самое большое послание из тех, которые канал

может нести в данном сеансе связи. Сетевой администратор помещает в эти буферы все сообщения для соединенных задач. Виртуальная задача, таким образом, занимает всего лишь пространство, необходимое для буфера и входа в таблице задач. Чтобы открыть виртуальный канал, необходимо знать идентификатор узла и задачи, с которой устанавливается связь. Для этого требуется идентификатор задачи-администратора, ответственного за данную функцию, или глобальное имя сервера. Не раскрывая здесь подробно механизм обмена посланиями, добавим лишь, что задача может вообще выполняться на другом узле, где, допустим, имеется более совершенный процессор.

Семейство операционных систем OS/2 Warp компании IBM

История появления, расцвета и практического ухода со сцены операционных систем под общим названием OS/2 и странна, и поучительна. Будучи одной из самых лучших операционных систем для персональных компьютеров по очень большому числу параметров и появившись существенно раньше систем своих основных конкурентов, она тем не менее не смогла стать самой распространенной, хотя могла бы, и с легкостью. Основная причина тому — законы бизнеса (умение рекламировать свой товар, всячески поддерживать его продвижение, вкладывать деньги в завоевание рынка), а не качество самой операционной системы. Во-первых, компания IBM не сочла необходимым продвигать свою операционную систему на рынок программного обеспечения, ориентированного на конечного пользователя, а решила продолжить свою практику работы исключительно с корпоративными клиентами. А этот рынок (корпоративного программного обеспечения) оказался существенно уже для персональных компьютеров, чем рынок программного обеспечения для конечного пользователя, ибо компьютеры типа IBM PC прежде всего являются персональными. Во-вторых, основные доходы компания IBM получала не от продажи системного программного обеспечения для персональных компьютеров, а за счет продаж дорогостоящих серверов и другого оборудования. Доходы от продажи операционной системы OS/2 не представлялись руководству компании IBM значимыми. Чтобы добиться успеха на рынке операционных систем для персональных компьютеров, необходимо было обеспечить всестороннюю поддержку своей системы соответствующей учебной литературой, широкой рекламой, заинтересовать разработчиков программного обеспечения. Увы, этого сделано не было, и сегодня уже практически мало кто знает о системах семейства OS/2. В то же время следует отметить, что те организации и предприятия, которые в свое время освоили эту систему и создали для нее соответствующее прикладное программное обеспечение, до сих пор не переходят на ныне чрезвычайно популярные операционные системы Windows NT/2000/XP, поскольку последние требуют существенно больше системных ресурсов. Любопытный факт: всем известные банкоматы работают под управлением OS/2.

Семейство 32-разрядных операционных систем OS/2 для IBM-совместимых персональных компьютеров начало свою историю с появления первой OS/2 v 2.0

в 1992 году. Ей предшествовала 16-разрядная операционная система с таким же названием — OS/2, которая была разработана для микропроцессора i80286. Этот микропроцессор, несмотря на множество принципиальных новаций, оказался неудачным. Защищенный режим работы этого 16-разрядного микропроцессора был несовершенным. Он обеспечивал работу с относительно небольшим объемом оперативной памяти, имел слабую аппаратную поддержку для организации виртуальной памяти, слишком низкое быстродействие (для того, чтобы выступать в качестве основы для построения мультизадачных операционных систем). Неудачная судьба 16-разрядной системы OS/2 1.x во многом повлияла ила 32-разрядную операционную систему, хотя по очень многим позициям архитектура 32-разрядной версии операционной системы OS/2 принципиально отличалась от своей предшественницы.

Компания IBM оставила этот проект, когда его версия имела номер 4.5. Сейчас из состава IBM отделилась небольшая компания, которая, выкупив проект OS/2, продолжает над ним работу и обеспечивает приверженцев этой операционной системы пакетами обновления и всевозможными добавлениями.

Все последние версии операционной системы OS/2 в своем названии имеют слово Warp, что переводится с английского как «основа». Операционная система OS/2 Warp 4.0 практически представляет собой OS/2 Warp 3.0 (вышедшую еще в 1994 году) с несколько улучшенной поддержкой DOS-задач и обновленными элементами объектно-ориентированного интерфейса. Для этой системы характерны:

* вытесняющая многозадачность (preemptive multitasking) и поддержка DOS- и Windows- (Win32s[1]) приложений;
* по-настоящему интуитивно понятный и действительно удобный объектный пользовательский интерфейс;
* поддержка стандарта открытого объектного документооборота OpenDoc;
* поддержка стандарта OpenGL;
* поддержка Java-апплетов и встроенных средств разработки на языке Java;
* поддержка шрифтов True Type (TTF);
* управление голосом без предварительной подготовки (технология Voice Type);
* полная поддержка сетевых технологий Интернет/интранет, доступ в сети CompuServe[2];
* средства построения одноранговых сетей и клиентские части для сетевых операционных систем IBM LAN Server, Windows, Lantastic, Novell Netware 4.1 (в том числе поддержка службы каталогов);
* система удаленного доступа через модемные соединения;
* файловая система Mobile File System для поддержки мобильных пользователей;
* стандарт автоматического распознавания аппаратных устройств (Plug-and-Play), но без столь навязчивого механизма, который реализован в Windows;

[1] Win32s — это одно из расширений интерфейса прикладного программирования систем Windows.
[2] Популярная американская служба.

* набор офисных приложений[1] (базы данных, электронные таблицы, текстовый процессор, генератор отчетов, деловая графика, встроенная система приема-передачи факсимильных сообщений, информационный менеджер);
* полная поддержка мультимедиа, включая средства работы с видеокамерой, расширенную систему помощи WarpGuide.

Однако наиболее заманчивы не перечисленные из рекламного буклета возможности системы, а удобная и надежная для работы с корпоративными базами данных и в сетях среда, предоставляющая клиентское рабочее место.

Операционная система OS/2 Warp предлагает единый интерфейс прикладного программирования (API), совместимый с рядом операционных систем, что позволяет снизить стоимость разработок. Все версии операционных систем OS/2 и LAN Server, включая текущие версии OS/2 Warp и OS/2 Warp Server 4.5, совместимы по восходящей линии, что позволяет экономить средства, необходимые для поддержания уже существующих прикладных программ.

Чрезвычайно важным для пользователей является тот факт, что компания IBM для всех версий своей операционной системы регулярно выпускает пакеты обновления (FixPak). Эти пакеты исправляют обнаруженные ошибки, а также вносят новые функции. Для пользователей такая практика сопровождения фирмой своей операционной системы, безусловно, более выгодна, нежели практика частого выпуска новых версий операционных систем (ей следует компания Microsoft).

Так, например, для одной из своих самых удачных операционных систем — Windows NT 4.0 — компания Microsoft выпустила всего 6 пакетов обновления (ServicePak), тогда как для уже совсем старой операционной системы OS/2 Warp 3.0, которая вышла в свет в 1994 году, компания IBM выпустила уже несколько десятков пакетов FixPak. Для операционной системы OS/2 Warp 4.0 вышло более 15 пакетов исправлений и обновлений.

Пакеты исправлений и обновлений пользователи получают бесплатно, тогда как за новую операционную систему приходится платить большие деньги. К тому же, длительная работа по исправлению имеющихся в системе ошибок приводит к тому, что количество последних со временем, как правило, уменьшается и система становится все более надежной и функциональной, в то время как новая версия операционной системы содержит не меньше ошибок, чем предыдущая. Последнее обстоятельство объясняется в том числе и тем, что объем ее исходного кода становится все больше и больше, а времени на создание операционной системы отводится столько же, если не меньше.

Немаловажным фактором является и то, что значительные капиталовложения требуются не только на приобретение новой операционной системы, но и на ее освоение. Для многих желательно, чтобы время жизни операционной системы составляло до 10 лет и более. В противном случае мы будем не только напрасно тратить

[1] Справедливости ради следует заметить, что этот набор приложений (называемый BonusPak) несовместим с современными версиями Microsoft Office, поэтому его используют, как правило, только в «закрытых системах», когда не предусматривается обмен документами, изготовленными посредством приложений Microsoft Office.

деньги на приобретение новых систем, но и не сможем обеспечить квалифицированную работу пользователей в этих системах. Современные операционные системы и прикладное программное обеспечение для своего освоения требуют длительного и дорогостоящего обучения пользователей. Поэтому желательно, чтобы все это программное обеспечение не требовало частого переобучения сотрудников (однако, с другой стороны, прогресс не стоит на месте, и большое количество конечных пользователей с нетерпением ожидают появления все более новых операционных систем и приложений).

Весьма полезным, как для управления приложениями, так и для создания несложных собственных программ, является наличие системы программирования на языке высокого уровня REXX, который иногда называют языком процедур. Можно сказать, что это встроенный командный язык, который служит для тех же целей, что и язык для пакетных (batch) файлов в среде DOS, но он обладает несравнимо большими возможностями. Это язык высокого уровня с нетипизированными переменными. Язык легко расширяем, любая программа OS/2 может добавлять в него новые функции. Помимо встроенного интерпретатора с языка REXX имеется система программирования Visual REXX. Имеется и объектно-ориентированная версия языка REXX с соответствующим интерпретатором.

Наиболее сильное впечатление при работе в операционной системе OS/2 оставляет объектно-ориентированный графический пользовательский интерфейс, а особой популярностью у программистов эта система пользовалась вследствие очень хорошей организации VDM-машин и высокого быстродействия при выполнении обычных DOS-приложений.

Особенности архитектуры и основные возможности

Строение и функционирование операционной системы OS/2 можно считать практически идеальными с точки зрения теории и довольно неплохими в реализации. В качестве подтверждения этому можно привести один пример, который представляется очень показательным: OS/2 до сегодняшних дней практически неизменна, начиная с версии 2.0, увидевшей свет в 1992 году. Этот факт говорит о глубокой продуманности архитектуры системы, ведь и по сей день OS/2 является одной из самых мощных и продуктивных операционных систем. Здесь самым показательным примером являются тесты серверов. В одной из вычислительных лабораторий Санкт-Петербургского государственного университета аэрокосмического приборостроения (ГУАП) с 1995 года в течение нескольких лет функции сервера кафедры вычислительных систем и сетей выполняла система OS/2 Warp Advanced Server. При переходе на сервер Windows NT 4.0 пришлось в два раза увеличить объем оперативной памяти и поменять процессор (с Pentium 90 на Pentium II300), и даже после этого скорость работы обычных приложений на рабочих станциях не достигла той производительности, какую имели пользователи при работе сервера под управлением OS/2. Аналогичные замечания не так давно можно было прочесть и в зарубежных публикациях — однопроцессорная машина под управлением OS/2 Warp Server обгоняет по производительности двухпроцессорную машину под управлением Windows NT.

Разработчики системы OS/2 решили не использовать всех возможностей защищенного режима, заложенных в микропроцессоры i80x86. Например, обработка прерываний чаще всего ведется через коммутаторы прерываний, а не через коммутаторы задач. Используется плоская модель памяти. Хорошо продуманная архитектура, в которой задействована модель клиент-сервер, и тщательное кодирование позволили получить систему, требующую очень небольших вычислительных ресурсов. Очень удачно реализована диспетчеризация задач. Представление различных системных информационных структур в статический форме (в виде таблиц) привело к более высокому быстродействию.

В OS/2 имеется несколько видов виртуальных машин для выполнения прикладных программ. Собственные 32- и 16-разрядные программы OS/2 выполняются на отдельных виртуальных машинах в режиме вытесняющей многозадачности и могут общаться между собой с помощью средств DDE OS/2. Прикладные программы DOS и Win16 могут запускаться на отдельных виртуальных машинах в многозадачном режиме. При этом они поддерживают полноценные связи DDE и OLE 2.0 друг с другом, а также связи DDE с 32-разрядными программами OS/2. Кроме того, при желании можно запустить несколько программ Win16 на общей виртуальной машине Win16, где они работают в режиме невытесняющей многозадачности, как в Windows 3.x. Конечно, нынче это уже неактуально, поскольку появилось огромное количество приложений, использующих API Win32, но в 90-е годы XX века эти факты имели существенное значение.

Разнообразные сервисные функции API OS/2, в том числе SOM (System Object Model — модель системных объектов), обеспечиваются с помощью системных библиотек DLL, к которым можно обращаться без требующих затрат времени переходов между кольцами защиты. Ядро операционной системы OS/2 предоставляет многие базовые сервисные функции API, обеспечивает поддержку файловой системы, управление памятью, имеет диспетчер аппаратных прерываний. В ядре виртуальных DOS-машин (Virtual DOS Machine, VDM), или в VDM-ядре, осуществляется эмуляция DOS и процессора 8086, а также управление VDM. Драйверы виртуальных устройств обеспечивают уровень аппаратной абстракции. Драйверы физических устройств напрямую взаимодействуют с аппаратурой.

Модуль реализации механизмов виртуальной памяти в ядре OS/2 поддерживает большие постраничные разбросанные адресные пространства, составленные из объектов памяти. Каждый объект памяти управляется так называемым *пейджером* — задачей вне ядра, обеспечивающей резервное хранение страниц объекта памяти. Адресные пространства управляются путем отображения или размещения объектов памяти внутри них. Ядро управляет защитой памяти и ее распределением на основе объектов памяти абстрактным образом, вне зависимости от каких-либо конкретных аппаратных средств трансляции процессорных адресов. В частности, ядро интенсивно использует режим копирования при записи для придания программам способности делить объекты памяти, не копируя множество страниц, когда новое адресное пространство получает доступ к объекту памяти. Новые копии страниц создаются, только когда программа в одном из адресных пространств обновляет их. Когда ядро принимает страничный сбой в объекте памяти и не име-

ет страницы памяти в наличии, или когда оно должно удалить страницы из памяти по требованию других работающих программ, ядро с помощью механизма IPC уведомляет пейджер об объекте памяти, в котором произошел сбой. После этого пейджер сервера приложений определяет, каким образом предоставить или сохранить данные. Это позволяет системе устанавливать различные семантики для объектов памяти, основываясь на потребностях программ, которые их используют.

Ядро управляет средами исполнения для программ, обеспечивая множественность заданий (процессов) и потоков выполнения. Каждое задание (процесс[1]) имеет свое собственное адресное пространство, или отображение. Ядро распределяет объекты памяти, которые задание отобразило на диапазон адресов внутри адресного пространства. Задание также является блоком размещения ресурсов и защиты, при этом заданиям придаются возможности и права доступа к средствам IPC системы. Для поддержки параллельного исполнения с другой программой в пределах одного адресного пространства ядро отделяет среду исполнения от реально выполняющегося потока. Таким образом, программа задания может быть загружена и исполнена в нескольких различных местах кода в одно и то же время на мультипроцессоре или параллельной машине. Это может привести к повышению быстродействия приложения.

Система IPC обеспечивает базовый механизм, позволяющий потокам работать в различных заданиях, взаимодействуя друг с другом, и надежную доставку сообщений в порты. *Порты* представляют собой защищенные каналы связи между заданиями. Каждому заданию, использующему порт, приписывается набор прав на этот порт. Права могут быть различными для разных заданий. Только одно задание может получить какой-либо порт, хотя любой поток внутри задания может выполнять операцию приема. Одно или более заданий могут иметь право посылать информацию в порт. Ядро позволяет заданиям применять систему IPC для передачи друг другу прав на порт. Оно также обеспечивает высокопроизводительный способ передачи больших объемов данных в сообщениях. Вместо того чтобы копировать данные, сообщение содержит указатель на них, который называется указателем на данные вне линии. Когда ядро передает сообщение от передатчика к приемнику, оно заставляет память, передаваемую через указатель, появиться в адресном пространстве приемника и, как вариант, исчезнуть из адресного пространства передатчика. Ядро само структурировано как задание с потоками, и большинство системных служб реализованы как механизмы IPC-обращений к ядру, а не как прямые системные вызовы.

Для поддержки операций ввода-вывода и доступа к внешним устройствам ядро операционной системы OS/2 обеспечивает доступ к ресурсам ввода-вывода, таким как устройства с отображаемой памятью, порты ввода-вывода и каналы прямого доступа к памяти (Direct Memory Access, DMA), а также возможность отображать прерывания на драйверы устройств, исполняемые в пользовательском пространстве. Службы ядра позволяют приоритетным программам получать устройства в свое владение: такими программами обычно являются программы, не связанные с заданиями, вроде серверов драйверов устройств, работающих как приложения. Поскольку ядро обязано обслужить все прерывания (в силу того, что прерывания обычно выдаются

[1] Здесь термины «задание» и известный нам «процесс» используются как синонимы.

в приоритетном состоянии компьютера, а также в целях поддержания целостности системы), оно имеет логику, которая определяет, должно ли оно обрабатывать прерывание или его следует отобразить на сервер. Если прерывание следует отобразить на приложение, это приложение должно быть зарегистрировано в ядре и содержать код, контролирующий отображение прерывания. Сразу после отображения в приложении запускается поток по обработке прерывания.

В соответствии с концепцией микроядерных операционных систем, непосредственно поверх ядра системы OS/2, которое построено с использованием этой архитектуры, располагается ряд служебных приложений, предоставляющих системные службы общего назначения, то есть службы, не зависящие от операционной среды, в которой выполняется приложение. Эти службы зависят только от ядра, некоторых вспомогательных служб, экспортируемых доминирующей задачей операционной системы, и от самих себя. В числе задачно-нейтральных служб имеются пейджер умолчания, мастер-сервер, который загружает другие задачно-нейтральные серверы в память, служба низкоуровневых имен, служба защиты, службы инициализации, набор драйверов устройств со связанным кодом поддержки, а также библиотечные подпрограммы для стандартной программной среды. Дополнительные задачно-нейтральные сервисы, например выделенный файловый сервер, могут быть просто добавлены.

С помощью ядра операционной системы и задачно-нейтральных сервисов приоритетная задача может обеспечить операционную системную среду типа UNIX. Поскольку приоритетная задача является прикладным сервером, можно добавлять другие серверы для различных задач, исполняющих программы, написанные в разных операционных системах, работающих на машине в одно и то же время.

Существуют некоторые операционные системные сервисы (вроде трансляции сообщений об ошибках), не обеспечиваемые задачно-нейтральными сервисами. Поскольку лучше не дублировать подобные сервисы, приоритетная задача предоставляет эти сервисы не только своим клиентским приложениям, но и любой другой задаче, исполняющейся в машине.

Особенности интерфейсов

В операционных системах OS/2 Warp в качестве стандартной графической оболочки используется среда Workplace Shell (WPS), организованная логичней и удобней, чем известный Windows-интерфейс. Оболочка Workplace Shell основана на модели системных объектов (SOM) фирмы IBM — мощной технологии, специально разработанной для решения таких проблем, как жесткая привязка объектов к их клиентам и необходимость использования одного и того же языка программирования. Объекты Workplace Shell работают в среде SOM, доступ в которую можно реализовать почти на всех языках программирования, где предусмотрены внешние процедуры, в том числе и на языке REXX.

В отличие от интерфейса GUI в Windows, где ярлыки (shortcuts) объектов никак не связаны между собой, в WPS объекты, имеющие аналогичные ярлыки (shadow[1]

[1] Shadow (по-английски «тень») — значок на рабочем столе OS/2, который является частью объекта, то есть имеется постоянная двухсторонняя связь между этим значком и собственно объектом.

в терминологии WPS), просто имеют дополнительное свойство — возможность многократно отображаться почти как самостоятельные объекты. Можно сделать несколько таких ярлыков из уже существующего ярлыка или из объекта. При этом любые ярлыки могут перемещаться в любое место и при этом их связи с основным объектом не теряются. Вроде бы то же самое происходит в GUI, но в WPS можно переместить основной объект, и его ярлыки тут же изменят свои параметры, тогда как в GUI произойдет разрушение связей, поскольку связи являются односторонними.

Про SOM можно сказать, что это не связанная ни с одним конкретным языком объектно-ориентированная технология для создания, хранения и использования двоичных библиотек классов. Ключевые слова здесь «двоичные» и «не связанная ни с одним конкретным языком». Хотя нынче многие считают OS/2 технологией прошлого, модель SOM на самом деле представляет собой одну из наиболее интересных разработок в области компьютерной индустрии даже на сегодняшний день. По существу, некоторые идеи, реализованные в OS/2 в начале 90-х годов прошлого столетия, сейчас только обещают реализовать в новом поколении операционных систем Windows с кодовым названием Whistler. Объектно-ориентированное программирование (ООП) заслужило безоговорочное признание в качестве основной парадигмы, однако его применению в коммерческом программном обеспечении препятствуют отсутствие в языках ООП средств обращения к библиотекам классов, подготовленным на других языках, и необходимость поставлять с библиотеками классов исходные тексты. Многим независимым разработчикам библиотек классов приходится продавать заказчикам исходные тексты, поскольку разные компиляторы по-разному отображают объекты. Настоящий потенциал модели SOM заключается в ее совместимости практически с любой платформой и любым языком программирования. Технология SOM соответствует спецификации CORBA (Common Object Request Broker Architecture — общая архитектура посредника объектных запросов), которая определяет стандарт условий взаимодействия между прикладными программами в неоднородной сети.

Графический интерфейс в системах OS/2 не единственный. Интересно отметить тот факт, что существует довольно много альтернативных оболочек для операционных систем OS/2, начиная с программы FileBar, которая хотя и кажется примитивной, но зато отлично работает на компьютерах с оперативной памятью объемом 4 Мбайт, и кончая мощной системой Object Desktop, которая значительно улучшает внешний вид экрана OS/2 и делает работу более удобной.

Помимо оболочек, улучшающих интерфейс операционной системы OS/2, имеется также ряд программ, расширяющих ее функциональность. В первую очередь, это Xfree86 for OS/2 — полноценная система X-Window, которая может использоваться как графический терминал при работе в сети с UNIX-машинами, а также для запуска программ, перенесенных из UNIX в OS/2. К сожалению, таких программ немного, однако большое количество UNIX-программ поставляется вместе с исходными кодами, которые, как правило, практически не нужно изменять для перекомпиляции под Xfree86/OS2.

Серверная операционная система OS/2 Warp 4.5

Серверная операционная система компании IBM, предназначенная для работы на персональных компьютерах и вышедшая в свет в 1999 году, носит название OS/2 WarpServer for e-Business, что подчеркивает ее основное назначение. Однако в процессе ее создания система носила кодовое название Аврора (Aurora), поэтому все ее так теперь и называют.

Как известно, предыдущие версии системы OS/2 могли предоставить программисту только 512 Мбайт виртуального адресного пространства для «родных» 32-разрядных приложений. В свое время это было очень много. Однако хотя задачи, требующие столь большого объема оперативной памяти, встречаются пока еще редко, некоторые считают ограничение в 512 Мбайт серьезным недостатком. Поэтому в последней версии системы это ограничение снято (напомним, что в операционной системе Windows NT 4.0 объем виртуального адресного пространства для задач пользователя составляет 2 Гбайт), и теперь максимальный объем виртуальной памяти для задачи в операционной системе OS/2 v. 4.5 по умолчанию составляет 2 Гбайт, но командой VIRTUALADDRESSLIMIT=3072 в конфигурационном файле CONFIG.SYS он может быть увеличен до 3 Гбайт.

В операционной системе OS/2 v. 4.5 разработчики постарались все «остатки» старого 16-разрядого кода, который еще частично оставался в предыдущих версиях системы, полностью заменить 32-разрядными реализациями, что повысило быстродействие системы. Прежде всего, обеспечена поддержка 32-раздядных драйверов устанавливаемых файловых систем (IFS), ибо в предыдущих системах работа с ними велась через трансляцию вызовов 32bit->16bit->32bit. В то же время для совместимости со старым программным обеспечением помимо 32-раздядного используется и 16-раздядный интерфейс API.

Создана новая файловая система JFS (Journaling File System — файловая система с протоколированием), призванная повысить надежность и живучесть файловой подсистемы по сравнению с файловой системой HPFS386. IFS. Файловая система JFS обеспечивает большую безопасность в структурах данных благодаря технике, разработанной для систем управления базами данных. Работа с JFS происходит в режиме транзакций с ведением журнала транзакций. В случае системных сбоев есть возможность обработки журнала транзакций с целью принятия или отмены изменений, произведенных во время системного сбоя. Эта система управления файлами также повышает скорость восстановления файловой системы после сбоя. Сохраняя целостность файловой системы, она, подобно файловой системе NTFS, не гарантирует восстановление пользовательских данных. Следует отметить, что файловая система JFS обеспечивает самую высокую скорость работы с файлами из всех известных систем, созданных для персональных компьютеров, что очень важно для серверной операционной системы.

Для работы с дисками создан специальный менеджер дисков — LVM (Logical Volume Manager — менеджер логических дисков). LVM хранит информацию обо всех устанавливаемых файловых системах и определяет имена дисков для программ, которые этого требуют. Это позволяет избирательно назначить любую букву любому разделу диска, что в ряде случаев можно считать удобным. И даже больше — теперь

операционной системе более не нужно использовать имена дисков. Менеджер логических дисков в совокупности с файловой системой JFS позволяет объединять несколько томов и даже несколько физических дисков в один большой логический том.

Контрольные вопросы и задачи

1. Изложите основные архитектурные особенности операционных систем семейства UNIX. Попробуйте объяснить основные различия между системами UNIX и Windows.
2. Перечислите и поясните основные понятия, относящиеся к UNIX-системам.
3. Что делает системный вызов fork()? Каким образом осуществляется в операционных системах семейства UNIX запуск новой задачи?
4. Изложите основные моменты, связанные с защитой файлов в UNIX.
5. Сравните разрешения NTFS, имеющиеся в Windows NT/2000/XP, с правами на доступ к файлам, реализованные в UNIX-системах.
6. Расскажите об особенностях семафоров в UNIX. Почему семафорные операции осуществляются сразу над множеством семафоров?
7. Что представляет собой вызов удаленной процедуры (RPC)?
8. Найдите в Интернете описание лицензии GNU и изучите его основные положения. Изложите их. Перечислите сильные и слабые стороны программного обеспечения с открытым исходным кодом.
9. Расскажите об операционной системе Linux. Какие проблемы, на ваш взгляд, наиболее важны для Linux? Расскажите об основных различиях между Linux и FreeBSD.
10. Что представляет собой X-Window? Что такое оконный менеджер? Какие оконные менеджеры для операционной системы Linux вы знаете?
11. Что представляет собой операционная система QNX? Перечислите ее основные особенности.
12. Почему про QNX часто говорят, что это «сетевая» операционная система? Что такое сетевой протокол FLEET?
13. Какие функции реализует ядро QNX?
14. В чем вы видите принципиальные различия между ядром Windows NT 4.0, которое считают построенным по микроядерным принципам, и ядром QNX?
15. Расскажите об основных механизмах взаимодействия для организации распределенных вычислений в операционной системе QNX.
16. Расскажите о проекте OS/2. Какие особенности архитектуры этой операционной системы представляются наиболее интересными?
17. Какие механизмы использует операционная система OS/2, чтобы уменьшить потребности в оперативной памяти и повысить производительность системы?