Государственный комитет Российской Федерации
по высшему образованию

Новосибирский государственный университет

Т.Б.Большаков Д.В.Иртегов

Операционные системы

Часть II

Новосибирск
2004

Учебное пособие по спецкурсу "Операционные системы" кафедры физико-технической информатики физического факультета НГУ. Спецкурс предполагает знакомство с архитектурой вычислительных систем, знание языка C и знакомство с языками ассемблера. Курс ориентирован на студентов 3-го курса физического и математического факультетов НГУ.

Пособие подготовлено с использованием материалов Областного Центра новых информационных технологий при НГУ.

Рецензенты: В.В.Городилов

Б.Н.Шувалов

Глава 1

Понятие операционной системы

1.1 Основные функции операционных систем

В работе [1] операционная система определяется так: "Я не знаю, что это такое, но всегда узнаю ее, если увижу". Эта фраза была сказана в первой половине 70-х, когда операционные системы действительно отличались большим разнообразием структуры и выполняемых функций.

С тех времен положение существенно изменилось. Современные ОС – по крайней мере, широко распространенные системы – во многом похожи друг на друга. Прежде всего это определяется требованием переносимости программного обеспечения. Именно для обеспечения этой переносимости был принят POSIX (Portable OS Interface based on uniX) – стандарт, определяющий минимальные функции по управлению файлами, межпроцессному взаимодействию и т.д., которые *должна* уметь выполнять система.

Кроме того, за четыре с лишним десятилетия, прошедших с момента разработки первых ОС, сообщество программистов достигло определенного понимания того, что:

- при разработке ОС возникает много стандартных проблем и вопросов;
- для большинства из этих проблем и вопросов существует набор стандартных решений;
- некоторые из этих решений намного лучше, чем все альтернативные.

Многие из таких наилучших решений были реализованы в операционных системах семейства Unix. *Поэтому среди адептов этой ОС ходит поговорка: "Если вы не понимаете* UNIX, *вы должны будете заново изобрести его". Опыт систем* OS/2 *и* Windows NT *отчасти подтверждает её.*

По современным представлениям, ОС должна уметь делать следующее:

- Обеспечивать загрузку пользовательских программ в оперативную память и их исполнение.

- Обеспечивать работу с устройствами долговременной памяти, такими как магнитные диски, ленты, оптические диски и т.д. Как правило, ОС управляет свободным пространством на этих носителях и структурирует пользовательские данные.

- Предоставлять более или менее стандартный доступ к различным устройствам ввода/вывода, таким как терминалы, модемы, печатающие устройства.

- Предоставлять некоторый пользовательский интерфейс. Слово *некоторый* здесь сказано не случайно – часть систем ограничивается командной строкой, в то время как другие на 90% состоят из средств интерфейса пользователя.

Существуют ОС, функции которых этим и исчерпываются. Одна из хорошо известных систем такого типа – дисковая операционная система MS DOS.

Более развитые ОС предоставляют также следующие возможности:

- Параллельное (точнее, псевдопараллельное, если машина имеет только один процессор) исполнение нескольких задач.

- Распределение ресурсов компьютера между задачами.

- Организация взаимодействия задач друг с другом.

- Взаимодействие пользовательских программ с нестандартными внешними устройствами.

- Организация межмашинного взаимодействия и разделения ресурсов.

- Защита системных ресурсов, данных и программ пользователя, исполняющихся процессов и самой себя от ошибочных и зловредных действий пользователей и их программ.

1.2 Классификация ОС

По тому, какие из вышеперечисленных функций реализованы и каким было уделено больше внимания, а каким меньше, системы можно разделить на несколько классов.

ДОС (Дисковые Операционные Системы)

Это системы, берущие на себя выполнение только первых четырех функций. Как правило, это просто некий резидентный набор подпрограмм, не более того. Он загружает пользовательскую программу в память и передает ей управление, после чего программа делает с системой все, что ей заблагорассудится. Считается желательным, чтобы после завершения программы машина оставалась в таком состоянии, чтобы ДОС могла продолжить работу. Если же программа приводит машину в какое-то другое состояние... что ж, ДОС ничем ей в этом не может помешать. Характерный пример – различные загрузочные мониторы для машин класса Spectrum. Как правило, такие системы работают одновременно только с одной программой.

Дисковая операционная система MS DOS для IBM PC-совместимых машин является прямым наследником одного из таких резидентных мониторов. Она, правда, умеет загружать несколько программ, но *не предоставляет* средств для исполнения этих программ. Более того, с точки зрения документированных функций, этим программам *нельзя* работать. Конечно, есть еще недокументированные *задние двери* (*backdoors*)...

Существование систем такого класса обусловлено их простотой и тем, что они потребляют мало ресурсов. Для машин класса Spectrum это более чем критичные параметры. Еще одна причина, по которой такие системы могут использоваться даже на довольно мощных машинах - требование программной совместимости с ранними моделями того же семейства компьютеров.

ОС

К этому классу относятся системы, берущие на себя выполнение всех вышеперечисленных функций. Разделение на ОС и ДОС идет, по-видимому, от систем IBM DOS/360 и OS/360 для больших компьютеров этой фирмы, клоны которых известны у нас в стране под названием ЕС ЭВМ серии 10XX. (Кстати, у IBM была еще TOS/360, Tape Operating System - Ленточная Операционная Система).

Здесь под ОС мы будем подразумевать системы "общего назначения", то есть рассчитанные на интерактивную работу одного или нескольких пользователей в режиме разделения времени, при не очень жестких требованиях на время реакции системы на внешние события. Как правило, в таких системах уделяется большое внимание защите самой системы, программного обеспечения и пользовательских данных от ошибочных и злонамеренных программ и пользователей. Обычно такие системы используют встроенные в архитектуру процессора средства защиты и виртуализации памяти. К этому классу относятся такие широко распространенные системы, как VAX/VMS, системы семейства Unix и OS/2, хотя последняя не обеспечивает одновременной работы нескольких пользователей и защиты пользователей друг от друга.

Системы виртуальных машин

Такие системы стоят несколько особняком. Это система, допускающая одновременную работу нескольких программ, но создающая при этом для каждой программы иллюзию, что машина находится в полном ее распоряжении, как при работе под ДОС. Примером такой системы является IBM VM, известная у нас в стране под названием СВМ (Система Виртуальных Машин). Аналогичным образом работают DesqView и другие **многозадачники** (**multitaskers**) для MS DOS.

Часто такие системы являются подсистемой ОС общего назначения: MS DOS и MS Windows-эмуляторы под UNIX и OS/2, окно DOS в MS Windows, эмулятор RT-11 в VAX/VMS.

В системах виртуальных машин часто приходится уделять много внимания эмуляции работы аппаратуры. Например, несколько программ могут начать программировать системный таймер. СВМ должна отследить такие попытки и создать для каждой из программ иллюзию, что она запрограммировала таймер именно так, как хотела. Разработка таких систем является сложным и часто неблагодарным делом и сильно зависит от аппаратуры, поэтому мы почти не будем обсуждать этот класс ОС.

Системы реального времени

Это системы, предназначенные для облегчения разработки так называемых **приложений реального времени**. Это программы, управляющие некомпьютерным по природе оборудованием, часто с очень жесткими ограничениями по времени. Примером такого приложения может быть программа бортового компьютера крылатой ракеты, системы управления ускорителем элементарных частиц или промышленным оборудованием. Такие системы обязаны поддерживать многопроцессность, *гарантированное* время реакции на внешнее событие, простой доступ к таймеру и внешним устройствам. Такие системы могут по другим признакам относиться как к классу ДОС (RT-11), так и к ОС (OS-9, QNX). Часто такие системы (например, VxWorks) рассчитаны на работу совместно с управляющей host-машиной, исполняющей "нормальную" операционную систему .

Гарантированное время реакции на внешнее событие является отличительным признаком систем РВ. Требование гарантированного времени реакции налагает специфические требования на архитектуру ОС; большинство современных ОС общего назначения непригодно для задач РВ.

Любопытно, что новомодное течение в компьютерной технике – **multimedia** – при качественной реализации предъявляет к системе те же требования, что и промышленные задачи реального времени. В multimedia основной проблемой является синхронизация изображения на экране со звуком. Именно в таком порядке. Звук обычно генерируется внешним аппаратным устройством с собственным таймером, и изображение синхронизуется с ним же. Человек способен заметить довольно малые временные неоднородности в звуковом потоке. Напротив, пропуск кадров в визуальном потоке не так заметен, а расхождение звука и изображения заметно уже при задержках около 30 мс. Поэтому системы качественного multimedia должны обеспечивать синхронизацию с такой же или более высокой точностью, что мало отличается от систем мягкого реального времени

Кросс-загрузчики

Это системы - полностью ориентированные на работу с host-машиной. Чаще всего они используются для написания и отладки кода, позднее прошиваемого в ПЗУ. Это системы программирования микроконтроллеров семейства Intel 8048 и подобных им, TDS (Transputer Development System) фирмы Inmos, и многие другие. Такие системы, как правило, включают в себя набор компиляторов и ассемблеров, работающих на host-системе (реже – загружаемых с host-машины в целевую систему), библиотеки, выполняющие большую часть функций ОС при работе программы (но не загрузку этой программы!), и средства отладки.

Системы промежуточных типов

Существуют системы, которые с первого взгляда нельзя отнести к одному из вышеперечисленных классов. Такова, например, система RT-11, которая, по сути своей, является ДОС, но позволяет одновременное исполнение нескольких программ с довольно богатыми средствами взаимодействия и синхронизации. Другим примером промежуточной системы являются MS Windows 3.x и Windows 95 которые, как ОС, используют аппаратные средства процессора для защиты и виртуализации памяти и даже могут обеспечивать некоторое подобие многозадачной работы, но *не защищают* себя и программы от ошибок других программ.

В последнее время вошел в употребление еще один термин: **сетевые ОС**, или сокращенно **NOS** (**Networking Operating System**). На взгляд авторов, сложившееся использование этого термина несколько неудачно. Его можно употреблять в двух различных смыслах:

1. Системы, предназначенные для предоставления сетевых услуг, аналогично тому, как ДОС предназначена для предоставления средств работы с диском. Под такое понимание NOS подходят узкоспециализированные системы, такие как Novell Netware, K9Q или программное обеспечение маршрутизаторов Cisco.

2. Системы, способные предоставлять сетевые услуги. Под такое определение подходят практически все современные ОС общего назначения.

Судя по тому, что большинство “обзоров сетевых операционных систем” в компьютерных журналах сравнивают не маршрутизатор Cisco с K9Q, а Windows NT с SunSoft Solaris или OS/2, термин NOS в этих публикациях понимается во втором смысле. Как уже говорилось, практически все современные ОС и некоторые ДОС способны предоставлять сетевые сервисы, поэтому этот термин почти эквивалентен словам “Современная ОС общего назначения” и, таким образом, почти не несет полезной информации.

1.3 Выбор операционной системы

Выбор типа операционной системы часто представляет собой нетривиальную задачу. Некоторые приложения накладывают жесткие требования, которым удовлетворяет только небольшое количество систем.

Например, задачи управления промышленным или исследовательским оборудованием в режиме жесткого реального времени вынуждают нас делать выбор между специализированными ОС реального времени и некоторыми ОС общего назначения, такими как Unix System V Release 4[1]. Другие приложения, например серверы баз данных, просто требуют высокой надежности и производительности, что отсекает системы класса ДОС и MS Windows.

Наконец, некоторые задачи, такие как автоматизация конторской работы, не предъявляют больших требований к надежности, производительности и времени реакции системы, что предоставляет широкий выбор между различными ДОС, MS Windows, Mac OS и многими системами общего назначения. При этом технические параметры системы перестают играть роль, и в игру вступают другие факторы. На заре персональной техники таким фактором была стоимость аппаратного обеспечения, вынуждавшая делать выбор в пользу ДОС и, позднее, MS Windows.

Сейчас же стоимость аппаратуры резко упала, а требования персонального программного обеспечения резко возросли. Последняя версия пакета MS Office занимает около 120 мегабайт на диске и требует для комфортабельной работы 8-16 мегабайт памяти и процессор класса 486/Pentium. Машины такого типа вполне достаточно для работы многих ОС общего назначения, таких как OS/2, Linux и некоторых коммерческих систем семейства Unix. Тем не менее, в большинстве ситуаций выбор попрежнему делается в пользу MS Windows. Обычно при этом ссылаются на отсутствие программного обеспечения для "альтернативных" систем, несмотря на то, что объективный анализ ситуации показывает несостоятельность этого утверждения.

Например, OS/2 способна исполнять практически все прикладное ПО, разработанное для DOS и MS Windows[2]; для этой системы существует также ряд офисных приложений – текстовые процессоры Describe, ClearLook, AmiPro/2, электронные таблицы Lotus-123 и т.д..

Для Linux приложений конторской направленности меньше, и они выглядят несколько странно для пользователя, привыкшего к MS Windows, однако они существуют – например, WYSIWYG текстовый процессор ez и электронные таблицы xess.

Мы не упоминаем здесь прикладное ПО, разработанное для Mac OS, потому что это ПО не

[1]Хотя Unix SVR4 теоретически способен обеспечивать гарантированное время реакции, системы этого семейства имеют ряд недостатков с точки зрения задач РВ, поэтому чаще всего предпочтительными оказываются специализированные ОС – QNX, VxWorks, OS-9 и т.д..

[2]Если быть точным, то ряд DOS и Windows-приложений, в первую очередь системные утилиты, некоторые игры, вирусы и некоторые программы multimedia, все-таки отказываются работать в эмуляции, но программы, необходимые в офисе – MS Word, Excel и им подобные – работают не хуже, чем под "родной" системой.

может исполняться на процессорах архитектуры x86, а машины фирмы Apple несколько дороже клонов IBM PC.

Утверждение о том, что MS Windows проще в эксплуатации, чем доступные альтернативы, также не соответствует действительности, особенно при работе в сети. Более убедительно звучат слова о том, что переход под другую ОС потребует переучивания пользователей и поиска или обучения специалистов, которые будут заниматься установкой и поддержкой системы. Большинство же современных специалистов по персональным компьютерам знакомы лишь с одной ОС – MS DOS/MS Windows...

Нужно отметить, впрочем, что MS Windows, несмотря на низкую надежность, сложность конфигурации и поддержки и ряд функциональных недостатков, вполне адекватна большинству задач конторской автоматизации. Проблемы возникают, когда задачи, стоящие перед организацией, выходят за пределы распечатки прайс-листов из MS Excel и набора писем в MS Word. Лучше всего проблемы этого рода выражены в следующей притче:

> Проблема. Организация имеет двенадцать велосипедов. Стоит задача: перевезти рояль. Что делать? Грузовик не предлагать...

Основная проблема MS Windows состоит вовсе не в том, что это не "настоящая" операционная система – "велосипед", в терминах процитированной притчи, а в том, что она не обеспечивает путей плавного и безболезненного перехода под другие платформы, даже если возникнет необходимость такого перехода. Строго говоря, тот же недостаток свойствен многим другим **закрытым** (**closed**) платформам, поставляемым одной фирмой и использующим нестандартные "фирменные" интерфейсы. Пока "закрытое" решение соответствует вашим требованиям, все хорошо, но когда вы выходите за пределы технологических возможностей данного решения, вы оказываетесь в тупике.

1.4 Открытые системы

Альтернативой закрытым решениям является концепция **открытых систем**. Идея открытых систем исходит из того, что для разных задач необходимы разные системы – как специализированные, так и системы общего назначения, просто по-разному настроенные и сбалансированные. Сложность состоит в том, чтобы:

- Обеспечить взаимодействие разнородных систем в гетерогенной сети.
- Обеспечить обмен данными между различными приложениями на разных платформах.
- Обеспечить переносимость прикладного ПО с одной платформы на другую, хотя бы путем перекомпиляции исходных текстов.
- Обеспечить по возможности однородный пользовательский интерфейс.

Эти задачи предполагается решать при помощи открытых стандартов – стандартных сетевых протоколов, стандартных форматов данных, стандартизации программных интерфейсов – **API** (**Application Program Interface** – интерфейс прикладных программ) и, наконец, стандартизации пользовательского интерфейса.

В качестве стандартного сетевого протокола предлагалась семиуровневая модель OSI, но прежде, чем на основе этой модели было разработано что-то полезное, получило широкое распространение семейство протоколов TCP/IP. Документация по протоколам этого семейства имеет статус **public domain** (**общественная собственность**); кроме того, есть по крайней мере одна программная реализация этого протокола, также имеющая статус **public domain** – сетевое ПО системы BSD Unix, поэтому TCP/IP является вполне приемлемым основанием для открытых систем.

Обсуждение стандартных форматов данных увело бы нас далеко от основной темы, но нужно отметить следующее: в настоящее время существует много общепризнанных стандартов представления изображений (особенно растровых) и звуковых данных, но некоторые типы данных так и не имеют признанной стандартной формы. Например, есть несколько открытых форматов представления форматированного текста: troff, LaTeXи другие пакеты макросов для системы TeX, и, наконец, стандарт SGML (Standard Generalized Markup Language), но ни один из этих стандартов не пользуется популярностью среди разработчиков коммерческих текстовых процессоров[3]. Причины такого отношения понятны: предоставление пользователю возможности без проблем обмениваться данными с текстовым процессором конкурента означает дать пользователю возможность выбирать между твоим процессором и его. Впрочем, пользователю от понимания не легче...

Для того чтобы как-то обеспечить переносимость программ между системами различных типов, принимались различные стандарты интерфейса между пользовательской (обычно говорят – прикладной, но это не всегда правильно) программой и ОС. Одним из первых таких стандартов был стандарт библиотек ANSI C. Он основан на системных вызовах ОС Unix, но функции MS DOS для работы с файлами (те, которые используют **file handle**) тоже достаточно близки к этому стандарту.

В конце 80-х - начале 90-х гг. было осознано, что средства для "тонкого" управления файлами и межпроцессного взаимодействия также нуждаются в стандартизации. Первоначально идея состояла в том, чтобы привести различных потомков системы UNIX к какому-то общему знаменателю. Для этой цели в ISO (International Standard Organisation) был создан ряд комитетов, принимавших стандарты для различных сфер деятельности (сетевое взаимодействие, работа в реальном времени и т.д.).

В настоящее время приняты только два документа этого стандарта из предполагаемых десяти. Эти документы регламентируют операции над файлами (стандарт POSIX.1) и интерпретатор команд (POSIX.2). Тем не менее, большинство производителей ОС объявили, что их системы, такие как VMS 5.4 и Windows NT, объявили о поддержке этого стандарта.

---

[3]Справедливости ради нужно отметить, что форматы troff и LaTeXочень неудобны для WYSIWYG текстовых процессоров, но SGML разрабатывался специально для них.

По оценкам пользователей NT, POSIX-*подсистема этой ОС является упражнением в том, насколько бесполезной можно сделать полную реализацию стандарта* POSIX, *или, наоборот, насколько полной может быть совершенно бесполезная реализация.*

Отчасти бесполезность этой подсистемы обусловлена неполнотой принятых в настоящее время документов стандарта POSIX*: например, они совершенно не охватывают сетевые интерфейсы, что позволило фирме MicroSoft реализовать собственный интерфейс* WinSock *вместо принятых в* Unix *механизмов. Это делает практически невозможным перенос сетевого программного обеспечения, и т.д..*

В качестве стандартного командного языка стандарт POSIX предлагает командный язык Bourne Shell. Со стандартизацией графического интерфейса пользователя ситуация несколько сложнее и будет обсуждаться в разделе 10.

Глава 2

Загрузка программ

Мы предполагаем, что вы знакомы с базовыми понятиями, т.е., вам не нужно объяснять, что оперативная память представляет собой последовательность ячеек-слов, и что номер каждой ячейки называется адресом, и т.д. С другой стороны, нам нужно пользоваться хорошо определенной терминологией. Поэтому часть этого раздела может оказаться пересказом известных для читателя сведений.

Как известно, существуют два типа адресов. Адреса первого типа называются виртуальными, или логическими. Это то число, которое вы увидите, если, скажем, распечатаете значение указателя. Говоря точнее, это тот адрес, который видит ваша программа, номер ячейки памяти в ее собственном адресном пространстве. У многих машин эти адресные пространства для разных программ различны.

Адреса другого типа называются физическими. Это тот адрес, который передается по адресным линиям шины процессора, когда этот процессор считывает или записывает данные в ОЗУ.

Вообще говоря, эти два адреса могут не иметь между собой ничего общего. Теоретически, могут существовать адреса физической памяти, которым не соответствует никакой виртуальный адрес ни у какой из программ. Это может просто означать, что в данный момент эта память никем не используется. Более интересная ситуация – это виртуальные адреса, которым не соответствует никакой физический адрес. Такая ситуация часто возникает в системах, использующих так называемую **страничную подкачку** (**page swapping** или просто **paging**). В этой ситуации сумма объемов адресных пространств всех программ в системе может превышать объем доступной физической памяти, то есть на машине с четырьмя мегабайтами ОЗУ вы можете исполнять программы, требующие 8 и более мег, например CorelDRAW![1]. Правда, организация такой подкачки – нетривиальная задача, но об этом в следующих разделах.

Для начала попробуем рассмотреть загрузку программы в виртуальную память. Для простоты мы будем считать, что эта виртуальная память представляет собой непрерывное адресное пространство. Это не всегда так, но на всех системах, которые мы будем серьезно обсуждать, это предположение

[1] Восклицательный знак перед точкой не опечатка, это программа так называется: CorelDRAW!

выполняется. Кроме того, будем считать, что программа была заранее собрана в некий единый самодостаточный объект, называемый **загрузочным** или **загружаемым модулем**. В ряде операционных систем программа собирается в момент загрузки из большого числа отдельных модулей, содержащих ссылки друг на друга, но об этом ниже.

2.1 Абсолютная загрузка

Первый, самый простой, вариант состоит в том, что мы всегда будем загружать программу с одного и того же адреса. Это возможно в следующих случаях:

- Система может предоставить каждой программе свое адресное пространство[2].
- Система может исполнять в каждый момент только одну программу. Так ведет себя CP/M, так же устроено большинство загрузочных мониторов для самодельных компьютеров. Похожим образом устроена система RT-11, но о ней чуть ниже.

Такой модуль называется **абсолютным загрузочным модулем**. Он представляет собой копию содержимого виртуального пространства программы в момент ее запуска. Точнее, наоборот, начальное содержимое адресного пространства формируется путем простого копирования модуля в память. В системе RT-11 такие файлы называются `.sav` от **saved** – сохраненный.

В системе UNIX на 32-разрядных машинах также используется абсолютная загрузка. Загружаемый файл начинается с заголовка, который содержит:

- “магическое число” – признак того, что это именно загружаемый модуль, а не что-то другое.
- число `TEXT_SIZE` – длину области кода программы ($TEXT$).
- `DATA_SIZE` – длину области инициализованных данных программы ($DATA$).
- `BSS_SIZE` – длину области неинициализованных данных программы (BSS).
- Стартовый адрес программы.

За заголовком следует содержимое областей TEXT и DATA. Затем может следовать отладочная информация. Она нужна символьным отладчикам, но самой программой не используется.

При загрузке система выделяет программе `TEXT_SIZE` байтов виртуальной памяти, доступной для чтения/исполнения, и копирует туда содержимое сегмента TEXT. Затем отматывается `DATA_SIZE`

[2]Это возможно только на процессорах, осуществляющих трансляцию виртуального адреса в физический. Такие процессоры сложнее, несколько медленнее и обладают рядом специфических и неустранимых недостатков по сравнению с процессорами, которые такой механики не имеют. Об этих недостатках будет в следующих разделах, пока скажем только, что транспьютеры фирмы Inmos не имеют диспетчера памяти и очень много на этом выигрывают.

байтов памяти, доступной для чтения/записи, и туда копируется содержимое сегмента DATA. Затем отматывается еще `BSS_SIZE` байтов памяти, доступной для чтения/записи, и прописываются нулями[3]. После этого программе выделяется пространство под стек, в стек помещаются позиционные аргументы и среда исполнения (**environment**), и управление передается на стартовый адрес. Программа начинает исполняться.

2.2 Относительная загрузка

Другой способ загрузки состоит в том, что мы грузим программу каждый раз с нового адреса. При этом мы должны настроить ее на новые адреса, а для этого...

Как известно, программа представляет собой последовательность команд. Каждая команда состоит из кода операции (который в данный момент нас не интересует) и одного или нескольких адресных полей. В процессорах, разработанных в 70-е гг. и в начале 80-х, у команды может быть переменное количество таких полей, от нуля до 5-6, а то и больше. У современных RISC-процессоров количество полей уменьшилось... Собственно, нас это тоже мало интересует. Зато нас очень интересует, каким образом вычисляется адрес на основании этого поля.

Существуют три способа вычисления адреса в команде. Первый способ очень прост – берется адресное поле, возможно, расширенное со знаком или нулями, и говорится, что это и будет наш виртуальный адрес. Это называется **абсолютной** адресацией. Очевидно, что если мы захотим "сдвинуть" программу по адресам виртуальной памяти так, чтобы она начиналась, скажем, не с адреса `01000`, а с адреса `02000`[4], то мы должны будем найти все команды с абсолютными адресными полями и прибавить ко всем этим полям разность нового и старого адресов.

Второй способ адресации состоит в том, что мы берем значение одного из регистров процессора, который указан соответствующем поле команды, прибавляем к нему значение адресного поля и получаем адрес. Такая адресация называется базовой. Если адрес формируется сложением двух или более регистров с адресным полем, то это называется базово-индексной адресацией.

Внимательный читатель может отметить, что способ формирования адреса в процессоре `8086`*, а также в реальном режиме более поздних процессоров той же линии, представляет собой именно базовую или базово-индексную адресацию, а вовсе никакую не сегментацию.*

В этом случае для перемещения программы нам нужно только изменить значения базовых

[3]Знатоки языка C могут вспомнить, что там неинициализованная статическая или внешняя переменная считается инициализованной значением 0. В некоторых реализациях, например в Turbo/Borland C, это не всегда так, но даже там существует соответствующая опция у линкера.

[4]Здесь и далее все числа записываются в формате языка C, т.е., число без всяких префиксов считается десятичным, число с незначащим нулем впереди – восьмеричным, а число, начинающееся с `0x`, как `0xFF` – шестнадцатиричным

регистров, и программа даже не узнает, что загружена с другого адреса, если только сама не будет перезагружать эти регистры.

Именно так происходит загрузка `.com`-файлов в системе MS DOS. Система выделяет свободную память, настраивает программе базовые регистры `DS` и `CS`, которые почему-то называются сегментными, и передает управление на стартовый адрес. Ничего больше делать не надо.

В более сложных ситуациях работа с базовой адресацией почти не отличается от абсолютной: мы должны запомнить все места в программе, где загружаются базовые регистры, и добавить к загружаемым значениям новый стартовый адрес. Единственное преимущество состоит в том, что таких мест в программе гораздо меньше, чем при абсолютной адресации. На практике разница здесь составляет десятки и сотни раз.

Для перемещения программы мы обязаны найти в ней и обработать все ссылки на абсолютные адреса. Такими ссылками могут быть не только адресные поля обычных команд и команды загрузки базовых регистров, но и статически инициализованные указатели, или даже изощрения на ассемблере вроде:

Пример 1

```
push dst_seg  ; Это и будет ссылкой на абсолютный адрес
push dst_offs
retf
```

Как бы то ни было, в перемещаемой программе мы вынуждены запоминать все ссылки на абсолютные адреса и в момент загрузки производить их настройку на реальный начальный адрес. Обычно это делается при помощи так называемой **таблицы перемещений** (**relocation table**), которая присоединяется к телу загружаемого модуля, и содержит смещения от начала модуля для каждой такой ссылки. Такой файл гораздо сложнее абсолютного загружаемого модуля, и носит название **относительного** или **перемещаемого** загрузочного модуля. Именно такой формат имеют `.exe`-файлы в системе MS DOS.

Наиболее поучительна в этом отношении система RT-11, в которой существуют загружаемые модули обоих типов. Обычные программы имеют расширение `.sav`, представляют собой абсолютные загружаемые модули и грузятся всегда с адреса 01000. Ниже этого магического адреса находятся вектора прерываний и стек программы. Сама операционная система вместе с драйверами размещается в верхних адресах памяти. Естественно, вы не можете загрузить одновременно два `.sav`-файла.

Однако, если вам *обязательно* нужно исполнять одновременно две программы, вы можете собрать вторую из них в виде относительного модуля: файла с расширением `.rel`. Такая программа будет загружаться в верхние адреса памяти, каждый раз разные, в зависимости от конфигурации ядра системы, количества загруженных драйверов устройств и других `.rel`-модулей.

2.3 Позиционно-независимый код

За всеми этими разговорами мы чуть было не забыли о третьем способе формирования адреса в программе. Это так называемая относительная адресация, когда адрес получается сложением адресного поля команды и *адреса самой этой команды* – значения счетчика команд (`IP` – Instruction Pointer или `PC` – Program Counter). Видно, что модуль, в котором используется только такая адресация, можно грузить с любого адреса без всякой перенастройки. Такой код называется позиционно-независимым.

Позиционно-независимые программы очень удобны для загрузки, но, к сожалению, их написание накладывает довольно жесткие ограничения на стиль программирования. Например, нельзя пользоваться статически инициализованными переменными указательного типа, нельзя делать на ассемблере фокусы, вроде того, который был приведен в примере 1, и т.д. На многих процессорах, например, на Intel 8080/8085 или многих современных RISC-процессорах позиционно-независимый код вообще невозможен – эти процессоры не поддерживают соответствующий режим адресации для данных. Возникают серьезные неудобства при сборке программы из нескольких модулей. Поэтому такой стиль программирования используют только в особых случаях. Например, многие вирусы под MS DOS и драйверы под RT-11 написаны именно таким образом.

Любопытное наблюдение. В эпоху RT-11 *хакеры писали драйверы. Сейчас они пишут вирусы. Еще любопытнее, что для некоторых персональных платформ, например, для* Amiga*, вирусов почти нет. Хакеры считают более интересным писать игры или демонстрационые программы для* Amiga*. Похоже, общение с* IBM PC *порождает у программиста какие-то агрессивные комплексы. Наблюдение это принадлежит не авторам: см.[3].*

2.4 Оверлеи (перекрытия)

Еще более интересный способ загрузки программы – это **оверлей** (**over-lay** – лежащий сверху) или, как это называли в старой русскоязычной литературе, **перекрытие**. Смысл оверлея состоит в том, чтобы не загружать программу в память целиком, а разбить ее на несколько модулей и затаскивать их в память по мере необходимости. При этом на одни и те же адреса в различные моменты времени будут отображены различные модули. Отсюда и название.

Потребность в таком способе загрузки появляется, если у нас виртуальное адресное пространство мало́, например 1 мегабайт или даже всего 64 килобайта (на некоторых машинах под RT-11 бывало и по 48K, и многие полезные программы нормально работали!), а программа относительно велика. На современных 32-разрядных системах виртуальное адресное пространство обычно измеряется гигабайтами, и большинству программ этого хватает, а проблемы с нехваткой можно решать совсем другими способами. Тем не менее, существуют различные системы, даже и 32-разрядные, в которых нет устройства управления памятью, и размер виртуальной памяти не может превышать объема микросхем ОЗУ, установленных на плате. Пример такой системы – упоминавшийся выше транспьютер.

Основная проблема при оверлейной загрузке состоит в следующем: прежде чем ссылаться на оверлейный адрес, мы должны понять, какой из оверлейных модулей в данный момент там находится. Для ссылок на функции это просто: вместо точки входа функции мы вызываем некую процедуру, называемую **менеджером перекрытий** (**overlay manager**). Эта процедура знает, какой модуль куда загружен, и при необходимости подкачивает то, что загружено не было. Перед каждой ссылкой на оверлейные данные мы должны выполнять аналогичную процедуру, что намного увеличивает и замедляет программу. Иногда такие действия возлагаются на программиста (MS Windows, Mac OS[5], иногда – на компилятор (*handle pointer* в Zortech C/C++ для MS DOS), но чаще всего с оверлейными данными вообще предпочитают не иметь дела. В таком случае оверлейным является только код.

В старых учебниках по программированию и руководствах по операционным системам уделялось много внимания тому, как распределять процедуры между оверлейными модулями. Действительно, подкачка модуля с диска представляет собой довольно длительный процесс, поэтому хотелось бы минимизировать эту подкачку. Для этого нужно, чтобы каждый оверлейный модуль был как можно более самодостаточным. Если это невозможно, стараются вынести процедуры, на которые ссылаются из нескольких оверлеев, в отдельный модуль, называемый резидентной частью или резидентным ядром. Это модуль, который всегда находится в памяти и не разделяет свои адреса ни с каким другим оверлеем. Естественно, оверлейный менеджер должен быть частью этого ядра.

Каждый оверлейный модуль может быть как абсолютным, так и перемещаемым. От этого несколько меняется устройство менеджера, но не более того. На архитектурах типа i80x86 можно делать оверлейные модули, каждый из которых адресуется относительно своего значения базового регистра `CS` и ссылается на данные, статически размещенные в памяти, относительно постоянного значения регистра `DS`. Такие модули можно загружать в память с любого адреса, может быть, даже вперемежку с данными. Именно так и ведут себя оверлейные менеджеры компиляторов Borland и Zortech.

2.5 Загрузка самой ОС

При загрузке самой ОС возникает специфическая проблема: в пустой машине, скорее всего, нет программы, которая могла бы это сделать.

В кросс-системах эта проблема решается просто. Как правило, значительная часть памяти разделяется между host-системой и целевой. Память целевой системы может быть видна host-машине как часть ее собственного физического адресного пространства (так называемая **backpane memory**), либо как специальное внешнее устройство. По принципу backpane memory организованы многие многопроцессорные машины, например транспьютерные “фермы”.

[5]Подробнее управение памятью в этих системах описывается в разделе 3.3.1

Например, в "Микропроцессорном Практикуме" память целевого компьютера на процессоре K580 (Советский клон процессора Intel 8080) загружалась из host-машины Электроника-60 (Советский клон PDP-11) через параллельный порт последней. Часто такая же техника используется при отладке программ для микроконтроллеров. Для этой цели на целевой плате должно стоять специальное устройство, **имитатор ПЗУ**, который имеет разъемы, совпадающие с кристаллом ПЗУ, который предполагается там поставить. Содержимое имитируемого ПЗУ может загружаться в такой имитатор через последовательный или параллельный порт, или какими-то другими методами.

В процессорах семейства транспьютер фирмы Inmos может использоваться похожая техника. Транспьютер имеет четыре или более высокоскоростных последовательных канала, называемых **линками**. Эти линки не являются внешними устройствами, они глубоко интегрированы в архитектуру процессора. Кроме того, транспьютер имеет специальный разъем, по которому передается сигнал `boot from link`. Если этот сигнал установлен, при включении питания или после получения сигнала сброса (`reset`) транспьютер начинает ждать, пока с какого-то из линков не придет байт, больший чем 1. Такой байт воспринимается как длина сообщения, которое сейчас должно прийти по этому же линку. Сообщение загружается в память с выделенного адреса и запускается как программа. В Inmos'овской документации такая программа называется первичным загрузчиком. Как правило, эта программа загружает следующую, более сложную, называемую вторичным загрузчиком, которая и занимается инициализацией всей системы. В случае сложной конфигурации она может затянуть в память еще один, третичный загрузчик, который и произведет инициализацию.

Похожий процесс происходит при загрузке современных операционных систем с диска. Это последовательное исполнение втягивающих друг друга загрузчиков возрастающей сложности называется **бутстрапом** (**bootstrap**), что можно перевести как "втягивание [себя] за шнурки от ботинок".

Большинство современных процессоров не имеют ничего похожего на линки. При включении питания или аппаратном сбросе такие процессоры исполняют команду, находящуюся по определенному адресу, что-нибудь типа `0xFFFFFFFA`. Обычно по этому адресу находится ПЗУ, в котором содержится программа первичного загрузчика. Кстати, транспьютер будет вести себя так же, если сигнал `boot from link` не установлен.

На многих системах в ПЗУ бывает прошито нечто большее, чем первичный загрузчик. Это может быть целая контрольно-диагностическая система, называемая консольным монитором. Такая система есть на всех машинах линии PDP-11/VAX и на VME-системах, рассчитанных на OS-9 или VxWorks. Такой монитор позволяет вам просматривать содержимое памяти по заданному адресу, записывать туда данные, запускать какую-то область памяти как программу, и многое другое. Он же позволяет выбирать устройство, с которого будет производиться дальнейшая загрузка. В PDP-11/VAX на таком мониторе можно даже писать программы, почти с таким же успехом, как на ассемблере. Нужно только уметь считать в уме в восьмеричной системе счисления.

На машинах фирмы Sun в качестве консольного монитора используется интерпретатор языка

Forth. На ранних моделях IBM PC в ПЗУ был прошит интерпретатор BASIC. Именно поэтому клоны IBM PC имеют огромное количество плохо используемого адресного пространства выше сегмента 0xC000. Вы можете убедиться в том, что BASIC там должен быть, вызвав из программы прерывание 0x60. Вы получите на мониторе сообщение вроде: `NO ROM BASIC. PRESS ANY KEY TO REBOOT`. Вообще говоря, этот BASIC не является консольным монитором в строгом смысле этого слова, так как получает управление не перед загрузкой, а лишь после того, как загрузка со всех устройств завершилась неудачей.

На многих больших компьютерах существует отдельная консольная подсистема. Это маломощная машина, используемая для инициализации центрального процессора и его загрузки. Так, у VAX-11/780 внутри стоит консольная PDP-11/03 (с которой и скопирована Электроника-60; у нее даже корпус точно такой же), которая загружает микропрограмму в процессор VAX и инициализует консольный терминал.

После запуска консольного монитора вы можете приказать системе начать загрузку. На IBM PC такое приказание отдается автоматически, и часто загрузка производится вовсе не с того устройства, с которого хотелось бы. На этом и основан жизненный цикл загрузочных вирусов.

Проще всего происходит загрузка с различных последовательных устройств – лент, перфолент, магнитофонов, перфокарточных считывателей и т.д. Первичный загрузчик считывает в память все, что можно считать с заданного устройства и передает управление на начало того, что прочитал.

В современных системах такая загрузка практически не используется. В них загрузка происходит с устройств с произвольным доступом, как правило – с дисков. При этом обычно в память считывается нулевой сектор нулевой дорожки диска. Содержимое этого сектора часто называют первичным загрузчиком, хотя в терминологии фирмы Inmos это будет уже вторичный загрузчик (первичный прошит в ПЗУ). Иначе этот загрузчик называют **загрузочным сектором**, или **boot-сектором**. Как правило, этот загрузчик ищет на диске начало файловой системы своей родной ОС, ищет в этой файловой системе файл с определенным именем, закачивает его в память и передает на него управление. Часто такой файл и является ядром операционной системы.

Размер вторичного загрузчика ограничен, чаще всего размером сектора на диске, то есть 512 байтами. Если файловая система имеет сложную структуру, иногда вторичному загрузчику приходится подкачать третичный загрузчик, размер которого может быть намного больше. Из-за большего размера этот загрузчик намного умнее и в состоянии разобраться в структурах данных файловой системы.

В старых версиях MS DOS (3.30 и ранее) эта проблема была решена еще проще: файлы IO.SYS и MSDOS.SYS обязаны занимать на диске непрерывное пространство. Загрузочный сектор находит корневую директорию, находит в ней ссылки на соответствующие файлы и их длину и считывает требуемое количество секторов, начиная со стартового. Это существенно проще, чем разбираться с типом FAT, размером кластера и всей относящейся к делу информацией. В ранних версиях DOS эти файлы вообще должны были лежать в начале диска.

Легко понять, что первичный и вторичный загрузчики должны быть абсолютными загружаемыми модулями, потому что возлагать на первичный загрузчик настройку и перемещение вторичного просто нечестно, а для первичного этого просто никто не может сделать. Как правило, третичный и последующие загрузчики также являются абсолютными модулями, потому что у вторичного загрузчика хватает своих проблем, кроме настройки адресов у загрузчика следующего уровня.

В современных системах возможен еще один интересный способ загрузки – загрузка с сети. Он происходит аналогично загрузке с диска – стартовое ПЗУ посылает в сеть пакет стандартного содержания, который содержит запрос к серверу удаленной загрузки. Этот сервер передает по сети вторичный загрузчик, и т.д. Такая механика может использоваться при загрузке бездисковых рабочих станций. Таким же образом умеют грузиться VAX/VMS, VxWorks и многие другие системы.

Когда ядро системы, наконец-то, окажется в памяти, оно обычно запускает некоторую специальную программу инициализации. В случае MS DOS такая программа содержится в модулях `MSDOS.SYS/IO.SYS`. Имеется в виду процедура интерпретации файла `CONFIG.SYS`. Эта процедура определяет параметры настройки системы, драйверы устройств, которые нужно загрузить, и т.д. В системе UNIX старых версий: System V старее, чем SVR4, или старых системах ветви BSD UNIX, все эти драйверы и параметры настройки намертво зашиты в ядро. Для изменения конфигурации системы вы должны собирать ядро заново. В случае BSD, которая поставляется в виде исходных текстов на C и ассемблере, вам, возможно, придется также перекомпилировать часть модулей.

Тем не менее, в UNIX имеется специальная инициализационная программа, которая так и называется - `init`. Эта программа запускает различные процессы-демоны, например `cron` – программу, которая умеет запускать другие заданные ей программы в заданные моменты времени, различные сетевые сервисы, программы, которые ждут ввода с терминальных устройств (`getty`), и т.д. То, что она запускает, вообще говоря, задается в специальном файле `/etc/inittab`[6]. Администратор системы может редактировать этот файл и устанавливать те сервисы, которые в данный момент нужны, избавляться от тех, которые не нужны, и т.д. Отчасти это похоже на группу `startup` в MS Windows. Вообще, аналогичный инициализационный сервис предоставляют все современные операционные системы.

Существуют операционные системы, которые не умеют самостоятельно выполнять весь цикл бутстрапа. Они используют более примитивную операционную систему, которая исполняет их вторичный (или какой это уже будет по счету) загрузчик, и помогает этому загрузчику затянуть в память ядро ОС. На процессорах i80x86 в качестве стартовой системы часто используется MS/DR DOS, а загрузчик новой ОС оформляется в виде `.EXE`-файла. Таким образом устроены MS Windows, DesqView и ряд других "многозадачников" для MS DOS. Таким же образом загружается сервер Nowell Netware, система Oberon для i386, программы, написанные для различных **расширителей DOS** (**DOS extenders**) и т.д.

Многие из перечисленных ОС, например MS Windows 3.x и Windows95, используют DOS и во

[6]В разных версиях системы этот файл может иметь разное имя. `/etc/inittab` используется в System V Release 3.

время работы в качестве дисковой подсистемы. Тем не менее, эти системы умеют самостоятельно загружать пользовательские программы и выполнять все перечисленные во введении функции и должны, в соответствии с нашим определением, считаться полноценными операционными системами.

2.6 Сборка программ

В предыдущем разделе шла речь о типах исполняемых модулей, но не говорилось ни слова о том, каким образом эти модули получаются. Вообще говоря, способ получения загружаемого модуля различен в различных ОС, но в настоящее время во всех широко распространенных системах этот процесс выглядит примерно одинаково. Это связано, прежде всего, с тем, что эти системы используют одни и те же языковые процессоры.

В большинстве современных языков программирования программа состоит из отдельных слабо связанных модулей. Как правило, каждому такому модулю соответствует отдельный файл исходного текста. Эти файлы независимо обрабатываются языковым процессором (компилятором), и для каждого из них генерируется отдельный файл, называемый **объектным модулем**. Затем запускается программа, называемая **редактором связей**, **компоновщиком** или **линкером** (**linker** – тот, кто связывает), которая формирует из заданных объектных модулей цельную программу.

Интегрированные среды, типа Turbo Pascal или Borland C/C++ организованы таким же образом, только в них компилятор и редактор связей по каким-то соображениям собраны вместе с текстовым редактором и “оболочкой” – модулями, которые перехватывают сообщения об ошибках компиляции и т.д.

Объектный модуль отчасти похож по структуре на перемещаемый загрузочный модуль. Действительно, мы, как правило, не знаем, в каком месте готовой программы окажутся объекты, определенные в нашем модуле. Объект в данном случае означает любую сущность, обладающую адресом. Это может быть переменная в смысле языка высокого уровня, точка входа функции, и т.д. Поэтому объектный модуль должен содержать структуру данных, похожую на таблицу перемещений в загрузочном модуле. Можно, конечно, потребовать, чтобы весь модуль был позиционно-независимым, но это, как говорилось выше, накладывает очень жесткие ограничения на стиль программирования, а на многих процессорах (например Intel 8085) просто невозможно. Кроме того, по причинам, описанным ниже, это невозможно *вообще*.

Кроме ссылок на собственные объекты, объектный модуль имеет право ссылаться на объекты, определенные в других модулях. Типичный пример такой ссылки – обращение к функции, которая определена в другом файле исходного текста. Самый простой способ отслеживать такие ссылки – собрать их в таблицу.

Таких таблиц должно быть две: внешние объекты, на которые ссылается модуль, и объекты, определенные внутри модуля, на которые можно ссылаться извне. Обычно с каждым таким объектом ассоциировано имя, называемое глобальным символом. Как правило, это имя совпадает с именем соответствующей функции или переменной в исходном языке.

Одна из трудностей состоит в том, что значение символа может быть определено двумя способами: как относительный адрес внутри модуля или как абсолютное значение. Чаще всего используется первый способ. Все адреса функций и переменных в обычных языках высокого уровня, таких как C, C++ или Fortran, определяются именно таким образом. Второй способ может использоваться в ассемблере:

Пример 2

```
     .asect
; Дальнейший код входит в специальный абсолютный .psect
     .org     0100
     .globl   SYMBOL
     SYMBOL: .word ?
```

Такая конструкция определит объект, который будет находиться по адресу `0100` независимо от положения соответствующего объектного модуля внутри программы, и даже от загрузочного адреса самой программы. Это может быть полезно при работе с отображенными на память внешними устройствами, межпрограммном взаимодействии и т.д.

Читатель, наверное, еще не понял, в чем же трудность. А трудность состоит в том, что при ссылке на символ мы не знаем, каким из двух способов он был определен. Поэтому при ссылках на внешние объекты мы должны использовать только абсолютную адресацию. Это условие, кроме того, значительно упрощает работу линкера. Но, с другой стороны, это делает невозможной сборку позиционно-независимых программ из нескольких модулей.

В большинстве существующих систем мы все-таки можем ссылаться на внешние символы при помощи относительной адресации. Например, мы можем написать на ассемблере нечто вроде:

Пример 3

```
        jmp SYMBOL - $
```

где `$` означает адрес текущей команды. Если наш линкер собирает абсолютный загрузочный модуль, такая конструкция породит вполне разумный код, даже если `SYMBOL` представляет собой абсолютный символ. В случае же перемещаемого модуля такая конструкция может создать большие сложности или вообще оказаться недопустимой. Действительно, в перемещаемом модуле мы добавляем начальный загрузочный адрес ко всем ссылкам на абсолютные адреса, а из такой ссылки мы должны его *вычесть*. Для этого мы должны, как минимум, уметь задавать такую операцию в таблице перемещений. Простая таблица на глазах превращается в целый командный язык. Насколько авторам известно, формат `.exe`-файла MS DOS такого не позволяет.

Если в загрузочном модуле превращение таблицы перемещений в программу на некотором командном псевдоязыке может быть нежелательно, то в объектном модуле оно неизбежно. Действительно, для каждой ссылки на внешний объект мы должны уметь сказать, является эта ссылка абсолютной или относительной, или это вообще должна быть разность или сумма двух или даже более адресов, и т.д. Для определения объекта, с другой стороны, мы должны уметь сказать, что это абсолютный или перемещаемый символ, или даже что он равен другому символу плюс заданное смещение, и т.д. Кроме того, в объектных файлах может содержаться отладочная информация, формат которой может быть очень сложным. Поэтому объектный файл представляет собой довольно сложную и рыхлую структуру. Размер собранной программы может оказаться в два или три раза меньше суммы длин объектных модулей.

Итак, типичный объектный модуль содержит следующие структуры данных:

- Таблицу перемещений, т.е. таблицу ссылок на перемещаемые объекты внутри модуля.
- Таблицу ссылок на внешние объекты. Иногда это называется таблицей или списком импорта.
- Таблицу объектов, определенных здесь, на которые можно ссылаться из других модулей. Иногда ее называют списком экспорта.

  Иногда эту таблицу объединяют с предыдущей и называют все это таблицей глобальных символов. В этом случае для каждого символа приходится указывать, определен он в данном модуле или нет, а если определен, то как.
- Различную служебную информацию, такую, как имя модуля, программу, которая его создала (например строка `"gcc compiled"`)
- Отладочную информацию.
- Собственно код и данные модуля. Как правило, эта информация разбита на именованные секции. В masm/tasm такие секции называются **сегментами**, в DEC'овских и UNIX'овых ассемблерах – **программными секциями** (**psect**). В готовой программе весь код или данные, принадлежащие к одной секции, собираются вместе.

Крупные программы часто состоят из сотен и более отдельных модулей. Кроме того, существуют различные пакеты подпрограмм, также состоящие из большого количества модулей. Один из таких пакетов используется практически в любой программе на языке высокого уровня – это так называемая стандартная библиотека. Для решения проблем, возникающих при поддержании порядка в наборах из большого количества объектных модулей, еще на заре вычислительной техники были придуманы **библиотеки объектных модулей**.

Библиотека, как правило, представляет последовательный файл, состоящий из заголовка, за которым последовательно уложены объектные модули. В заголовке содержится следующая информация:

- Список всех объектных модулей, со смещением каждого модуля от начала библиотеки. Это нужно для того, чтобы можно было легко найти требуемый модуль.
- Список всех глобальных символов, определенных в каждом из модулей, с указанием, в каком *именно* модуле он был определен.

Линкер обычно собирает в программу все объектные модули, которые были ему заданы в командной строке, даже если на этот модуль не было ни одной ссылки. С библиотечными модулями он ведет себя несколько иначе.

Встретив ссылку на глобальный символ, линкер ищет определение этого символа во всех модулях, которые ему были заданы. Если там такого символа нет, то линкер ищет этот символ в заголовке библиотеки. Если его нет и там, линкер говорит: `Не определен символ SYMBOL` и завершает работу. Некоторые линкеры, правда, могут продолжить работу и даже собрать загружаемый модуль, но, как правило, таким модулем пользоваться нельзя, так как в нем содержится ссылка на некорректный адрес. Если же определение символа в библиотеке есть, линкер вытаскивает соответствующий модуль и дальше работает так, будто этот модуль был задан ему наравне с остальными объектными файлами. Этот процесс повторяется до тех пор, пока не будут разрешены все глобальные ссылки, в том числе и те, которые возникли в библиотечных модулях – или пока не будет обнаружен неопределенный символ. Благодаря такому алгоритму в программу включаются только те модули из библиотеки, которые нужны.

Во многих современных системах с виртуальной памятью существует понятие разделяемой библиотеки. С точки зрения линкера она отличается от обычной тем, что он всегда обязан настраивать ее на одни и те же виртуальные адреса, и не имеет права производить перенастройку самого кода библиотеки. Кроме того, этот код хранится вовсе не в загружаемом модуле, а в отдельном файле. Часто этот файл представляет собой загружаемый модуль специальной структуры. Все программы, использующие такую библиотеку, в действительности работают с одной копией ее кода, но каждая из них создает свою копию ее данных. Это достаточно сильно экономит память и дисковое пространство, используемое для хранения программ, особенно в случае больших библиотек, таких, как X Window Toolkit.

В существуют также операционные системы, производящие сборку программ в момент загрузки. В таких системах исчезает различие между абсолютными и относительными загружаемыми модулями. Как правило, программа привязывается к тому адресу, с которого она будет загружена, как при абсолютной загрузке, но при этом сам адрес может изменяться.

Некоторые архитектуры процессоров поддерживают динамически пересобираемые программы, у которых вся настройка модуля вынесена в отдельную таблицу. В этом случае модуль может быть прилинкован одновременно к нескольким программам, использовать одновременно разные копии сегмента данных, и каждая инкарнация модуля при этом даже не будет подозревать о существовании других. Примером такой архитектуры является Pascal-система Lilith, разработанная Н.Виртом, и ее наследники Кронос/N9000.

В этих архитектурах каждый объектный модуль соответствует одному модулю в смысле языка высокого уровня Oberon (или NIL - N9000 Instrumental Language). Далее мы будем описывать архитектуру системы N9000, поскольку авторы с ней лучше знакомы.

Модуль может иметь не более 256 процедур, не более 256 переменных и ссылаться не более, чем на 256 других модулей. Код модуля является позиционно-независимым. Данные модуля собраны в отдельный сегмент, и для каждой инкарнации модуля, то есть для каждой программы, которая этот модуль использует, создается своя копия сегмента данных. В начале сегмента содержится таблица переменных. Строки этой таблицы содержат либо значения – для скалярных переменных, таких как целое число или указатель, либо адреса в сегменте данных. Кроме того, сегмент данных содержит ссылку на сегмент кода. Этот сегмент кода содержит в себе таблицу адресов точек входа всех определенных в нем функций.

Ссылки на все внешние модули собраны в таблицу, которая также содержится в сегменте данных. Внешний модуль определяется началом его сегмента данных.

Все ссылки на объекты в данном модуле осуществляются через индекс в соответствующей таблице. Ссылки на внешние модули имеют вид `индекс модуля: индекс объекта`.

Сегмент данных не может содержать никаких статически инициализованных данных. Вся инициализация производится специальной процедурой, которая вызывается при создании каждой новой инкарнации модуля.

Все эти свойства реализованы в системе команд, поэтому накладные расходы относительно невелики. Точнее, они невелики по сравнению с Intel 80286, но уже великоваты по сравнению с i386, а по сравнению с современными RISC-процессорами или системами типа транспьютера они становятся уже недопустимыми.

Видно, что в системе может существовать несколько программ, обращающихся к одним и тем же модулям и использующих одну и ту же копию кода модуля. Проблем с абсолютной/относительной загрузкой вообще не возникает.

Операционная система TC для N9000 была[7] основана на сборке программ в момент загрузки. В системе имелась специальная команда `load` – "загрузить все модули, используемые программой, и разместить для них сегменты данных, но саму программу не запускать". В памяти могло сидеть одновременно несколько программ; при этом модули, используемые несколькими из них, загружались в одном экземпляре. Это значительно ускоряло работу. Например, можно было загрузить в память текстовый редактор, и запуск его занимал бы доли секунды, вместо десятков секунд, которые нужны для загрузки с жесткого диска фирмы ИЗОТ.

Любопытно, что когда началась реализация системы программирования на языке С для этой машины, по ряду причин было решено не связываться с динамической сборкой, а собирать обычные перемещаемые загрузочные модули.

Вообще, среди современных ОС довольно много систем, использующих тот или иной способ сборки при загрузке. Таким образом устроен, например, Novell Netware. Таким же образом организован ряд систем реального времени, таких как OS-9 или VxWorks.

[7] Авторы не уверены, существует ли в настоящее время хотя бы одна работоспособная машина этого типа.

Сборка при загрузке существенно замедляет процесс загрузки программы, но упрощает, с одной стороны, разделение кода, а с другой стороны – разработку программ. Действительно, из классического цикла внесения изменения в программу: редактирование текста – перекомпиляция – пересборка – перезагрузка (программы, не обязательно всей системы), выпадает целая фаза. В случае большой программы это может быть длительная фаза. В случае Novell Netware решающим оказывается первое преимущество, в случае систем реального времени одинаково важны оба.

В системах MS Windows и OS/2 используется способ загрузки, промежуточный между сборкой в момент загрузки и сборкой заранее. Загрузочный модуль в этих системах может быть полностью самодостаточным, а может содержать ссылки на другие модули, называемые **DLL** (**Dynamically Loadable Library** – динамически загружаемая библиотека). Самое хорошее в этой схеме то, что модуль, по собственному желанию, может выбирать *различные* библиотеки. Единственное ограничение состоит в том, что такие библиотеки обязаны быть совместимыми по вызовам.

Например, программа CorelDRAW! может импортировать и экспортировать изображения в различных видах, начиная от собственного внутреннего формата `.CDR` или Windows Bitmap, и кончая сильноупакованным форматом Jpeg или специализированными форматами, вроде Targa-файлов. Импорт и экспорт каждого формата выполняется отдельной DLL.

DLL на первый взгляд кажутся удобным средством разделения кода и создания отдельно загружаемых программных модулей, но они имеют очень серьезное ограничение.

Это ограничение проявляется, когда мы пытаемся создать в многопроцессной среде DLL для работы с разделяемым ресурсом, например многооконную графическую систему для работы нескольких задач с разделяемым терминалом. Как будет показано в разделе 4.2, DLL, работающая с разделяемыми данными, не сможет обрабатывать одновременно два вызова из разных процессов. Это создаст много проблем, разрешение которых либо невозможно в принципе, либо сопряжено с большими неудобствами. Поэтому в многопроцессной среде нам придется выделить оконную систему в отдельный процесс и оформлять обращения к ней не как обычные вызовы процедур, а как посылку сообщений средствами межпроцессного взаимодействия. Например, таким образом реализована сетевая оконная система X Windows, реализованная на всех ОС семейства Unix и многих других системах.

Средства межпроцессного взаимодействия будут подробнее обсуждаться в разделе 5, а сейчас мы скажем только, что эти средства в современных ОС по мощности не уступают прямому вызову процедуры. Существует даже модель взаимодействия, которая так и называется – **RPC** (**Remote Procedure Call** – Удаленный вызов процедуры). Само собой, такое использование этих средств сопровождается бо̀льшими накладными расходами, чем прямой вызов; зато мы получаем намного большую свободу – например, мы можем обращаться к программе, исполняющейся на другом процессоре или вообще на другой машине, или разделять общий ресурс (например, общую базу данных) с другим процессом.

Поэтому то, что в MS Windows делается при помощи DLL, в большинстве современных ОС реализуется средствами межпроцессного взаимодействия.

Кроме того, использование DLL сильно замедляет процесс загрузки программ и несколько снижает общую производительность систем с виртуальной памятью. Системы семейства Unix использующие монолитный загрузочный имодуль и/или разделяемые библиотеки, заметно быстрее, чем OS/2 и Windows NT, использующие DLL.

Глава 3

Управление оперативной памятью

Основной ресурс системы, распределением которого занимается ОС – это оперативная память. Поэтому организация памяти оказывает большое влияние на структуру и возможности ОС. В настоящее время сложилась даже более интересная ситуация – переносимая операционная система UNIX, рассчитанная на машины со страничным диспетчером памяти, произвела жесткий отбор, и теперь практически все серьезные машины, начиная от i386 и заканчивая суперкомпьютерами или, скажем процессором Alpha, имеют именно такую организацию памяти.

3.1 Открытая память

Самый простой случай управления памятью - ситуация, когда диспетчер памяти отсутствует, и в системе может быть загружена только одна программа. Именно в таком режиме работают CP/M и RT-11 SJ (Single-Job, однозадачная).

В этих системах программы загружаются с фиксированного адреса `PROG_START`. В CP/M это `0x100`; в RT-11 – `01000`. В адресах от 0 до начала программы находятся вектора прерываний, а в RT-11 – также и стек программы. Сама система размещается в старших адресах памяти. Адрес `SYS_START`, с которого она начинается, зависит от количества памяти у машины и от конфигурации самой ОС.

В этом случае управление памятью со стороны системы состоит в том, что загрузчик проверяет, поместится ли загружаемый модуль в пространство от `PROG_START` до `SYS_START`. Если объем памяти, который использует программа, не будет меняться во время ее исполнения, то на этом все управление и заканчивается.

Однако программа может использовать динамическое управление памятью, например функцию `malloc()` или что-то в этом роде. В этом случае уже код `malloc()` должен следить за тем, чтобы не залезть в системные адреса. Как правило, динамическая память начинает размещаться с адреса `PROG_END = PROG_START + PROG_SIZE`. `PROG_SIZE` в данном случае обозначает полный размер программы, то есть размер ее кода, статических данных и области, выделенной под стек.

Функция `malloc()` поддерживает некоторую структуру данных, следящую за тем, какие блоки памяти из уже выделенных были освобождены. При каждом новом запросе она сначала ищет блок подходящего размера в своей структуре данных и, только когда этот поиск завершится неудачей, откусывает новый блок памяти у системы. Для этого используется переменная, которая в библиотеке языка C называется `brklevel`. Изначально эта переменная равна `PROG_END`, ее значение увеличивается при выделении новых блоков, но в некоторых случаях может и уменьшаться. Это происходит, когда программа освобождает блок, который заканчивается на текущем значении `brklevel`.

3.2 Алгоритмы динамического управления памятью

Динамическое распределение памяти (его еще иногда называют управлением **кучей** (**pool** или **heap**)) представляет собой нетривиальную проблему. Действительно, активное использование функций `malloc/free` может привести к тому, что вся доступная память будет разбита на блоки маленького размера, и попытка выделения большого блока завершится неудачей, даже если сумма длин маленьких блоков намного больше требуемой. Это явление называется фрагментацией памяти. Кроме того, большое количество блоков требует длительного поиска. Существует также много мелких трудностей разного рода. К счастью, человечество занимается проблемой распределения памяти уже давно, и найдено много хороших или приемлемых решений.

В зависимости от решаемой задачи используются различные алгоритмы поиска свободных блоков памяти. Действительно, программа может требовать множество блоков одинакового размера, или нескольких фиксированных размеров. Это сильно облегчает решение проблемы фрагментации и поиска. Возможны ситуации, когда блоки освобождаются в порядке, обратном тому, в котором они выделялись. Это позволяет свести выделение памяти к стековой структуре. Возможны ситуации, когда некоторые из занятых блоков можно переместить по памяти. Так, например, функцию `realloc()` в ранних реализациях системы UNIX можно было использовать именно для этой цели.

В стандартных библиотеках языков высокого уровня, таких как `malloc/free/realloc` в C, `new/dispose` в Pascal и т.д., как правило, используются алгоритмы, рассчитанные на худший случай: программа требует блоки случайного размера в случайном порядке и освобождает их также случайным образом. Возможен, правда, более неприятный случай:

Пример 4

```
while(TRUE) {
     void * b1 = malloc(random(10));
     /* Случайный размер от 0 до 10 байт */
     void * b2 = malloc(random(10)+10);
     /*   ........      от 10 до 20 байт */

     if(b1 == NULL && b2 == NULL) /* Если памяти нет */
        break;                    /* выйти из цикла  */

     free(b1);
 }
 void * b3 = malloc(150);

 /* Скорее всего, память не будет выделена */
```

В результате исполнения такой программы вся доступная память будет "порезана на лапшу": между любыми двумя свободными блоками будет размещен занятый блок меньшего размера. К счастью, пример 4 носит искусственный характер. В обычных программах такая ситуация встречается редко, и часто оказывается проще исправить программу, чем вносить изменения в универсальный алгоритм управления кучей.

Варианты алгоритмов распределения памяти исследовались еще в 50-е годы. Итоги многолетнего изучения этой проблемы приведены в [2] и многих других учебниках.

Возможны алгоритмы распределения памяти двух типов: когда размер блока является характеристикой самого блока, и когда его сообщают отдельно при освобождении. К первому типу относится `malloc/free`, ко второму – `GetMem/FreeMem` в Turbo Pascal. В первом случае с каждым блоком ассоциируется некоторый дескриптор, который содержит длину этого блока и, возможно, какую-то еще информацию. Этот дескриптор может храниться отдельно от блока, или быть его заголовком. Иногда такой дескриптор "окружает" блок, то есть состоит из двух меток – в начале блока и в его конце. Для чего это может быть полезно, будет обсуждаться ниже.

Обычно все свободные блоки памяти объединяются в двунаправленный связанный список. Список должен быть двунаправленным для того, чтобы из него в любой момент можно было извлечь любой блок. Впрочем, если все действия по извлечению блока производятся после поиска, то можно слегка усложнить процедуру поиска и всегда сохранять указатель на предыдущий блок. Это решает проблему извлечения и можно ограничиться однонаправленным списком. Беда только в том, что многие алгоритмы при объединении свободных блоков извлекают их из списка в соответствии с адресом, поэтому для таких алгоритмов двунаправленный список остро необходим.

При первом взгляде на проблему возникает желание отсортировать список по размеру блока.

На самом деле это бессмысленно: время поиска в сортированном списке улучшается всего в два раза по сравнению с несортированным (вот в массиве или в дереве – совсем другое дело), зато добавляется время вставки в список, пропорциональное $O(n)$, где n – размер списка. Помещать блоки в сортированный массив еще хуже – время вставки становится $O(n + \log(n))$ и появляется ограничение на количество блоков. Использование хэш-таблиц или двоичных деревьев требует больших накладных расходов и усложнений программы, которые себя в итоге не оправдывают. Поэтому используют несортированный список.

Поиск в списке может вестись двумя способами: до нахождения **первого подходящего** (**first fit**) блока или до блока, размер которого ближе всего к заданному – **наиболее подходящего** (**best fit**). Для нахождения наиболее подходящего мы обязаны просматривать весь список, в то время как первый подходящий может оказаться в любом месте, и среднее время поиска будет меньше. Насколько меньше – зависит от отношения количества подходящих блоков к общему количеству. (Читатели, знакомые с теорией вероятности, могут самостоятельно вычислить эту зависимость).

Кроме того, в общем случае best fit увеличивает фрагментацию памяти. Действительно, если мы нашли блок с размером больше заданного, мы должны отделить “хвост” и пометить его как новый свободный блок. Понятно, что в случае best fit средний размер этого хвоста будет маленьким, и мы в итоге получим большое количество мелких блоков, которые невозможно объединить, так как пространство между ними занято.

При использовании first fit с линейным двунаправленным списком возникает специфическая проблема. Если каждый раз просматривать список с одного и того же места, то большие блоки, расположенные ближе к началу, будут чаще удаляться. Соответственно, мелкие блоки будут иметь тенденцию скапливаться в начале списка, что увеличит среднее время поиска. Простой способ борьбы с этим явлением состоит в том, чтобы просматривать список то в одном направлении, то в другом. Более радикальный и еще более простой метод состоит в том, что список делается кольцевым, и поиск каждый начинается с того места, где мы остановились в прошлый раз. В это же место добавляются освободившиеся блоки. В результате список очень эффективно перемешивается и никакой “антисортировки” не возникает.

В ситуациях, когда мы размещаем блоки нескольких фиксированных размеров, алгоритмы best fit оказываются лучше. Однако библиотеки распределения памяти рассчитывают на худший случай, и в них обычно используются алгоритмы first fit.

В случае работы с блоками нескольких фиксированных размеров напрашивается такое решение: создать для каждого типоразмера свой список. Это избавляет программиста от необходимости выбирать между first и best fit, устраняет поиск в списках как явление… короче, решает сразу много проблем.

Интересный вариант этого подхода для случая, когда различные размеры являются степенями числа 2, как 512 байт, 1Кбайт, 2Кбайта и т.д., называется **алгоритмом близнецов**. Он состоит в том, что мы ищем блок требуемого размера в соответствующем списке. Если этот список пуст, мы берем список блоков вдвое большего размера. Получив блок вдвое большего размера, мы делим его пополам. Ненужную половину мы помещаем в соответствующий список свободных блоков. Любопытно, что нам совершенно неважно, получили ли мы этот блок просто из соответствующего списка, или же делением пополам вчетверо большего блока, и далее по рекурсии. Одно из преимуществ этого метода состоит в простоте объединения блоков при их освобождении. Действительно, адрес блока-близнеца получается простым инвертированием соответствующего бита в адресе нашего блока. Нужно только проверить, свободен ли этот близнец. Если он свободен, то мы объединяем братьев в блок вдвое большего размера, и т.д.

Алгоритм близнецов значительно снижает фрагментацию памяти и резко ускоряет поиск блоков. Наиболее важным преимуществом этого подхода является то, что даже в наихудшем случае время поиска не превышает $O(log(S_{max}) - log(S_{min}))$, где S_{max} и S_{min} обозначают соответственно максимальный и минимальный размеры используемых блоков. Это делает алгоритм близнецов трудносзаменимым для ситуаций, когда необходимо гарантированное время реакции – например, для задач реального времени. Часто этот алгоритм или его варианты используются для выделения памяти внутри ядра ОС. Например, функция `kmalloc`, используемая в ядре ОС `Linux`, основана именно на алгоритме близнецов.

Разработчик программы динамического распределения памяти обязан решить еще одну важную проблему, а именно – объединение свободных блоков. Действительно, обидно, если мы имеем сто свободных блоков по одному килобайту и не можем сделать из них один блок в сто килобайт. Но если все эти блоки расположены в памяти один за другим, а мы не можем их при этом объединить – это просто унизительно. Кроме того, если мы умеем объединять блоки и видим, что объединенный блок ограничен сверху значением `brklevel`, то мы можем вместо помещения этого блока в список просто уменьшить значение `brklevel` и, таким образом, вернуть ненужную память системе.

Рассмотрим способы решения этой проблемы. Один способ упоминался в описании алгоритма близнецов, но он пригоден только в особых случаях.

Представим себе для начала, что все, что мы знаем о блоке, – это его начальный адрес и размер. Такая ситуация возможна при выделении памяти “второго рода”, т.е. когда размер освобождаемого блока передается как параметр процедуры `FreeMem`. Кроме того, такое возможно и в случаях, когда дескриптор блока содержит только его длину.

Легко понять, что это очень плохая ситуация. Действительно, для объединения блока с соседями мы должны найти их в списке свободных, или же убедиться, что там их нет. Для этого мы должны просмотреть весь список. В порядке мозгового штурма можно высказать идею сортировать список свободных блоков по адресу...

Гораздо проще запоминать в дескрипторе блока указатели на дескрипторы соседних блоков. Немного развив эту идею, мы приходим к методу, который упоминался в начале раздела. Этот метод называется алгоритмом парных меток и состоит в том, что мы добавляем к каждому блоку по два слова памяти.

Именно сло́ва, а не байта. Дело в том, что требуется добавить достаточно места, чтобы хранить там размер блока в байтах или словах. Обычно такое число занимает столько же места, сколько и адрес, а размер слова обычно равен размеру адреса. На `x86` *в реальном режиме это не так, но это вообще довольно странный процессор.*

Итак, мы добавляем к блоку два слова – одно спереди, другое сзади. В оба слова мы записываем размер блока. Это и есть тот дескриптор, который окружает блок. При этом мы говорим, что значения длины будут положительными, если блок свободен, и отрицательными, если блок занят. Можно сказать и наоборот, важно только потом соблюдать это соглашение.

Представим, что мы освобождаем блок с адресом `addr`. Считаем, что addr имеет тип `word *`, и при добавлении к нему целых чисел результирующий адрес будет отсчитываться в словах, как в языке C. Для того чтобы проверить, свободен ли его сосед спереди, мы должны посмотреть слово с адресом `addr - 2`. Если оно отрицательно, то сосед занят, и мы должны оставить его в покое. Если же оно положительно, то мы можем легко определить адрес начала этого блока как `addr - addr[-2]`. Определив адрес начала блока, мы можем легко объединить этот блок с блоком `addr`, нам нужно только сложить значения меток-дескрипторов и записать их в дескрипторы нового большого блока. Нам даже не нужно будет добавлять освобождаемый блок в список и извлекать оттуда его соседа!

Похожим образом присоединяется и сосед сзади. Единственное отличие состоит в том, что этого соседа все-таки нужно извлекать из списка свободных блоков.

Дополнительное преимущество приведенного алгоритма состоит в том, что мы можем отлавливать такие ошибки, как многократное освобождение одного блока, запись в память за границей блока и иногда даже обращение к уже освобожденному блоку. Действительно, мы в любой момент можем проверить всю цепочку блоков памяти и убедиться в том, что *все* свободные блоки стоят в списке, что в нем стоят *только* свободные блоки, что сами цепочка и список не испорчены, и т.д.

Это действительно большое преимущество, так как оно значительно облегчает ловлю ошибок работы с указателями, о которых в руководстве по Zortech C/C++ *сказано, что "опытные программисты, услышав это слово ["pointer bug" – прим. авт.], бледнеют и прячутся под стол" ([4]).*

Итак, наилучшим из известных универсальных алгоритмов динамического распределения памяти является алгоритм парных меток с объединением свободных блоков в двунаправленный кольцевой список и поиском по принципу first fit. Этот алгоритм обеспечивает приемлемую производительность почти для всех стратегий распределения памяти, используемых в прикладных программах. Такой алгоритм используется практически во всех реализациях стандартной библиотеки языка C и во многих других ситуациях. Другие известные алгоритмы либо просто хуже, чем этот, либо проявляют свои преимущества только в специальных случаях.

К основным недостаткам этого алгоритма относится отсутствие верхней границы времени поиска подходящего блока, что делает его неприемлемым для задач реального времени.

Некоторые системы программирования используют специальный метод освобождения динамической памяти, называемый **сборкой мусора**. Этот метод состоит в том, что ненужные блоки памяти не освобождаются явным образом. Вместо этого используется некоторый более или менее изощренный алгоритм, следящий за тем, какие блоки еще нужны, а какие – уже нет.

Самый простой метод отличать используемые блоки от ненужных – считать, что блок, на который есть ссылка, нужен, а блок, на который ни одной ссылки не осталось – не нужен. Для этого к каждому блоку присоединяют дескриптор, в котором подсчитывают количество ссылок на него. Каждая передача указателя на этот блок приводит к увеличению счетчика ссылок на 1, а каждое уничтожение объекта, содержавшего указатель – к уменьшению.

Впрочем, при таком подходе возникает специфическая проблема. Если у нас есть циклический список, на который нет ни одной ссылки извне, то все объекты в нем будут считаться используемыми, хотя они и являются мусором. Если мы по тем или иным причинам уверены, что кольца не возникают или возникают очень редко, метод подсчета ссылок вполне приемлем; если же мы используем графы произвольного вида, необходим более умный алгоритм.

Все остальные методы сборки мусора так или иначе сводятся к поддержанию базы данных о том, какие объекты на кого ссылаются. Использование такой техники возможно практически только в интерпретируемых языках типа Lisp или Prolog, где с каждой операцией можно ассоциировать неограниченно большое количество действий.

3.3 Открытая память (продолжение)

Описанные выше алгоритмы распределения памяти используются не операционной системой, а библиотечными функциями, пришитыми к программе. Однако многозадачная или многопрограммная[1] ОС также должны использовать тот или иной алгоритм размещения памяти.

Отчасти такие алгоритмы могут быть похожи на работу malloc. Однако режим работы ОС может вносить существенные упрощения в алгоритм.

Так, например, процедура управления памятью MS DOS рассчитана на случай, когда программы выгружаются из памяти только в порядке, обратном тому, в каком они туда загружались[2]. Это позволяет свести управление памятью к стековой дисциплине.

Каждой программе в MS DOS отводится блок памяти. С каждым таким блоком ассоциирован дескриптор, называемый MCB – Memory Control Block. Этот дескриптор содержит размер блока, идентификатор программы, которой принадлежит этот блок и признак того, является ли данный блок последним в цепочке. Нужно

[1] MS DOS – типичный пример однозадачной многопрограммной ОС. Действительно, в ней может быть одновременно загружено несколько программ, но система выполняет только одно задание.

[2] На самом деле, программы могут выгружаться и в другом порядке, но обычно это приводит к проблемам.

отметить, что программе всегда принадлежит *несколько* блоков, но это уже несущественные детали. Другая малосущественная деталь та, что размер сегментов и их адреса отсчитываются в параграфах размером 16 байт. Знакомые с архитектурой процессоров 80x86 должны понять, что адрес MCB в этом случае будет состоять только из сегментной части с нулевым смещением.

Идентификатор программы в MS DOS *может служить темой для отдельного разговора. Во многих документах он носит громкое название* pid *(идентификатор процесса) – почти как в* UNIX, *но на самом деле это всего лишь адрес* PSP *(Program Segment Prefix).*

После запуска .com-файл получает сегмент размером 64K, а .exe – всю доступную память. Обычно .exe-модули сразу после запуска освобождают ненужную им память и устанавливают brklevel на конец своего сегмента, а потом увеличивают brklevel и наращивают сегмент по мере необходимости. Естественно, что наращивать сегмент можно только за счет следующего за ним в цепочке MCB, и MS DOS разрешит делать это только в случае, если этот сегмент не принадлежит никакой программе.

При запуске программы DOS берет *последний* сегмент в цепочке, и загружает туда программу, если этот сегмент достаточно велик. Если он недостаточно велик, DOS говорит Not enough memory и отказывается загружать программу.

При завершении программы DOS освобождает все блоки, принадлежавшие программе. При этом соседние блоки объединяются. Пока программы, действительно, завершаются в порядке, обратном тому, в котором они запускались, – все вполне нормально. Другое дело, что в реальной жизни возможны отклонения от этой схемы.

Например, неявно предполагается, что TSR-программы (Terminate, but Stay Resident) никогда не пытаются завершиться. Тем не менее любой уважающий себя хакер считает своим долгом сделать резидентную программу выгружаемой. У некоторых хакеров она, в результате, выбрасывает при выгрузке все резиденты, которые сели в память после них. Другой пример – отладчики обычно загружают программу в обход обычной DOS-овской функции LOAD & EXECUTE, а при завершении отлаживаемой программы сами освобождают память из-под нее.

Один из авторов в свое время занимался прохождением некоторой программы под отладчиком. Честно говоря, речь шла о взломе некоторой игрушки... Эта программа производила какую-то инициализацию, а потом вызывала функцию DOS LOAD & EXECUTE. Я об этом не знал и, естественно, провалился внутрь новой программы, которую и должен был, по-хорошему, взламывать. После нескольких нажатий <CTRL> <Break> я наконец-то попал в отладчик, но при каком-то очень странном состоянии программы. Полазив по программе некоторое время и убедившись, что она не хочет приходить в нормальное состояние, я вышел из отладчика и увидел следующую картину: системе доступно около 100Кбайт в то время, как сумма длин свободных блоков памяти более 300Кбайт, а размер наибольшего свободного блока около 200Кбайт. Отладчик, выходя, освободил свою память и память отлаживаемой программы, но не освободил память из-под нового загруженного модуля. В результате посредине памяти остался никому не нужный блок памяти изрядного размера, помеченный как используемый. Самым обидным было то, что DOS не пыталась загрузить ни одну программу в память *под* этим блоком, хотя там было гораздо больше места, чем *над* ним.

В качестве итога можно сказать, что основными проблемами многопрограммных систем без диспетчера памяти являются:

- Проблема выделения дополнительной памяти программе, которая загружалась не последней.
- Проблема освобождения памяти после завершения программы. В системах с монолитным загружаемым модулем иногда просто запрещают программам выгружаться. В MS DOS была сделана попытка запретить выгружать TSR и драйверы, но это привело к поиску задних дверей.
- Низкая надежность. Ошибка в одной из программ может привести к порче кода или данных других программ или самой системы.
- Проблемы безопасности. В системах с открытой памятью невозможны эффективные средства разделения доступа. Любая программная проверка прав доступа может быть легко обойдена прямым вызовом "защищаемых" модулей ядра. Даже криптографические средства не обеспечивают достаточно эффективной защиты, потому что можно посадить в память троянскую программу, которая будет анализировать код программы шифрования и считывать значение ключа...

В системах с динамической сборкой первые две проблемы не так остры, потому что память выделяется и освобождается небольшими кусочками, по блоку на каждый объектный модуль, поэтому код программы обычно не занимает непрерывного пространства. Соответственно, такие системы часто разрешают и данным программы занимать несмежные области памяти.

Такой подход используется многими системами с открытой памятью – AmigaDOS, Oberon, системами программирования для транспьютера и т.д. Однако в таких системах очень острую форму приобретает проблема фрагментации свободной памяти.

Количество фрагментов приблизительно пропорционально количеству операций выделения/освобождения памяти. Вероятность не найти блок подходящего размера из-за фрагментации оценить сложнее, но она также растет с количеством операций.

Понятно, что если эти операции осуществляются сразу многими программами, то фрагментация пропорционально возрастает. Вышеперечисленные системы не предоставляют никаких средств для борьбы с фрагментацией.

Напротив, в системе MacOS было предложено достаточно оригинальный метод борьбы с фрагментацией. Метод этот заслуживает отдельного обсуждения.

3.3.1 Управление памятью в MacOS и MS Windows

В этих системах предполагается, что пользовательские программы не сохраняют указателей на динамически выделенные блоки памяти. Вместо этого каждый такой блок идентифицируется целочисленным дескриптором или "ручкой" (**handle**). Когда программа непосредственно обращается к данным в блоке, она выполняет системный вызов `GlobalLock` (запереть). Этот вызов возвращает текущий адрес блока. Пока программа не исполнит вызов `GlobalUnlock` (отпереть), система не пытается изменить адрес блока. Напротив, если блок не заперт, система считает себя вправе передвигать его по памяти или даже сбрасывать на диск.

Таким образом, “ручки” представляют собой попытку создать программный аналог аппаратных диспетчеров памяти. Они позволяют решить проблему фрагментации и даже организовать некое подобие виртуальной памяти. Можно рассматривать их как средство организации оверлейных данных – поочередного отображения разных блоков данных на одни и те же адреса. Однако за это приходится платить очень дорогой ценой.

Использование “ручек” сильно усложняет программирование вообще и в особенности перенос ПО из систем, использующих линейное адресное пространство. Все указатели на динамические структуры данных в программе нужно заменить на “ручки”, а каждое обращение к таким структурам необходимо окружить вызовами `GlobalLock/GlobalUnlock`. Эти вызовы:

- сами по себе увеличивают объем кода и время исполнения;
- мешают компиляторам выполнять оптимизацию, прежде всего не позволяют оптимально использовать регистры процессора, потому что далеко не все регистры сохраняются при вызовах;
- требуют разрыва конвейера команд и перезагрузки командного кэша; в современных суперскалярных процессорах это может приводить к падению производительности во много раз.

Попытки уменьшить число блокировок требуют определенных интеллектуальных усилий. Фактически, к обычному циклу разработки ПО: проектирование, выбор алгоритма, написание кода и его отладка – добавляется еще две фазы: микрооптимизация использования “ручек” и отладка оптимизированного кода. Последняя фаза оказывается, пожалуй, самой сложной и ответственной.

Наиболее неприятной и опасной ошибкой, возникающей на фазе микрооптимизации, является вынос указателя на динамическую структуру за пределы скобок `GlobalLock/GlobalUnlock`. Эту ошибку очень сложно обнаружить при тестировании, так как она проявляется только если система пыталась передвигать блоки в промежутках между обращениями. Иными словами, ошибка может проявлять или не проявлять себя в зависимости от набора приложений, исполняющихся в системе и от характера деятельности этих приложений. В результате мы получаем то, чего больше всего боятся инженеры, механики и программисты – систему, которая работает *иногда*.

Не случайно фирма MicroSoft полностью отказалась от “ручечного” управления памятью в новой версии MS Windows – Windows 95, где реализована полноценная виртуальная память.

3.4 Системы с базовой виртуальной адресацией

Как уже говорилось, в системах с открытой памятью возникают большие сложности при организации многозадачной работы. Чтобы устранить их, необходимо предоставлять каждой задаче свое виртуальное адресное пространство. Наиболее простым способом организовать различные адресные пространства является так называемая **базовая адресация**. По-видимому, это исторически наиболее ранний способ.

Вы можете заметить, что термин **базовая адресация** уже занят – мы называли таким образом адресацию по схеме `reg[offset]`. Дело в том, что метод, о котором сейчас идет речь, состоит в формировании адреса по той же схеме. Отличие состоит в том, что регистр, относительно которого происходит адресация, не доступен прикладной программе. Кроме того, его значение прибавляется ко *всем* адресам, в том числе к "абсолютным" адресным ссылкам или переменным типа указатель. По существу, такая адресация является способом организации виртуального адресного пространства. Действительно, распечатав значение указателя, вы увидите число, равное, например, `0x000A25`. Однако такой адрес будет соответствовать физическому адресу `BASE + 0x000A25` и меняться вместе со значением регистра `BASE`.

Если говорить о разнице между двумя значениями слов "базовая адресация", нельзя не упомянуть о поучительной истории, связанной с управлением памятью в системах линии IBM System 360.

В этих машинах не было аппаратных средств управления памятью и все программы разделяли общее виртуальное адресное пространство, совпадающее с физическим. Адресные ссылки в программе задавались 12-битовым смещением относительно базового регистра. В качестве базового регистра мог использоваться, в принципе, любой из 16 32-битных регистров общего назначения. Предполагалось, что пользовательские программы не модифицируют базовый регистр, поэтому можно загружать их с различных адресов просто перенастраивая значение этого регистра. Таким образом была реализована одновременная загрузка многих программ в многозадачной системе OS/360.

Однако после загрузки программу уже нельзя было перемещать по памяти: например при вызове подпрограммы адрес возврата сохраняется в стеке в виде абсолютного 24-битного адреса[3] и при возврате базовый регистр не используется. Аналогично, базовый регистр не используется при ссылках на параметры подпрограмм языка FORTRAN, при работе с указателями в PL/I и т.д.. Перемещение программы, даже с перенастройкой базового регистра, нарушило бы все такие ссылки.

Разработчики фирмы IBM вскоре осознали пользу перемещения программ после их загрузки и попытались как-то решить эту проблему. Очень любопытный документ "Preparing to Rollin-Rollout Guideline"[4] описывает действия, которые программа должна была бы предпринять после перемещения. Фактически программа должна была найти в своем сегменте данных все абсолютные адреса и сама перенастроить их.

Естественно, никто из разработчиков компиляторов и прикладного программного обеспечения не собирался следовать этому руководству. В результате проблема перемещения программ в OS/360 не была решена вплоть до появления машин System 370 со страничным или странично-сегментным диспетчером памяти и ОС MVS.

Как правило, машины, использующие базовую адресацию, имеют два регистра. Один из ре-

[3] В System 360 под адрес отводилось 32-разрядное слово, но использовались только 24 младших бита адреса. В System 370 адрес стал полностью 32-разрядным.

[4] К сожалению, авторам не удалось найти полного текста этого документа.

гистров задает базу для адресов, второй устанавливает верхний предел. В системе ICL1900/Одренок эти регистры называются соответственно BASE и DATUM. Если адрес выходит за границу, установленную значением DATUM, возникает **исключительная ситуация** (**exception**) ошибочной адресации. Как правило, это приводит к тому, что система принудительно завершает работу программы.

При помощи этих двух регистров мы сразу решаем две важные проблемы.

Во-первых, мы можем изолировать программы друг от друга – ошибки в одной программе не приводят к разрушению или повреждению других программ или самой системы. Благодаря этому мы можем обеспечить защиту системы не только от ошибочных программ, но и от злонамеренных действий пользователей по разрушению системы или доступу к чужим данным.

Во-вторых, мы получаем возможность передвигать адресные пространства задач по физической памяти так, что сама программа не замечает, что ее передвинули. За счет этого мы решаем проблему фрагментации памяти и даем программам возможность наращивать свое адресное пространство. Действительно, в системе с открытой памятью программа может добавлять себе память только до тех пор, пока не упрется в начало следующей программы. После этого мы должны либо говорить, что памяти нет, либо мириться с тем, что программа может занимать несмежные области физического адресного пространства. Второе решение резко усложняет управление памятью, как со стороны системы, так и со стороны программы, и часто оказывается неприемлемым (подробнее связанные с этим проблемы обсуждаются в разделе 3.3). В случае же базовой адресации мы можем просто сдвинуть мешающую нам программу вверх по физическим адресам.

Часто системы, работающие на таких архитектурах, умеют сбрасывать на диск те задачи, которые долго не будут исполняться. Это самая простая из форм **своппинга** (**swapping** – обмен)[5].

В современных системах базовая виртуальная адресация используется редко. Дело не в том, что она плоха, а в том, что более сложные методы, такие как сегментная и страничная трансляция адресов, оказались намного лучше. Часто под словами "виртуальная память" подразумевают именно сегментную или страничную адресацию.

3.5 Сегментная и страничная виртуальная память

В системах с сегментной и страничной адресацией виртуальный адрес имеет сложную структуру. Он разбит на два битовых поля: номер страницы (сегмента) и смещение в нем. Соответственно, адресное пространство оказывается состоящим из дискретных блоков. Если все эти блоки имеют фиксированную длину и образуют вместе непрерывное пространство, они называются страницами. Если длина каждого блока может задаваться произвольно, а неиспользуемым частям блоков соответствуют "дыры" в виртуальном адресном пространстве, они называются сегментами. Как правило, один сегмент

[5] Термин "страничный обмен" довольно широко распространен, но в данном случае его использование звучало бы неправильно

соответствует коду или данным одного модуля программы. Со страницей или сегментом могут быть ассоциированы права чтения, записи и исполнения.

Такая адресация реализуется аппаратно. Процессор, как правило, имеет специальное устройство, называемое **диспетчером памяти**. В некоторых процессорах, например в MC68020 или MC68030 или в некоторых RISC-системах, это устройство реализовано на отдельном кристалле; в других, таких как i386 или Alpha, диспетчер памяти интегрирован в процессор.

Диспетчер памяти содержит регистр – указатель на **таблицу трансляции**. Эта таблица размещается где-то в физической памяти. Ее элементами являются **дескрипторы** каждой страницы/сегмента. Такой дескриптор содержит права доступа к странице, признак присутствия этой страницы в памяти и физический адрес страницы/сегмента. Для сегментов в дескрипторе также хранится длина сегмента.

Как правило, диспетчер памяти имеет также **кэш** (**cache**) дескрипторов – быструю память с ассоциативным доступом. В этой памяти хранятся дескрипторы часто используемых страниц.

Алгоритм доступа к памяти по виртуальному адресу `page:offset` получается следующим:

- Проверить, существует ли страница `page` вообще. Например, у машин семейства VAX адресное пространство *может* состоять из 8М страниц, что соответствует 4 Гбайтам, но реальные программы используют намного меньше памяти, и размеры их таблиц трансляции соответственно уменьшаются. Естественно, про страницу, для которой нет соответствующего элемента таблицы, можно сказать, что ее не существует. Такая проверка осуществляется простым сравнением номера страницы с длиной таблицы трансляции. Если страницы не существует, возникает **особая ситуация ошибки сегментации** (**segmentation violation**)
- Попытаться найти дескриптор страницы в кэше.
- Если его нет в кэше, загрузить дескриптор из таблицы в памяти.
- Проверить, имеет ли процесс соответствующее право доступа к странице. Иначе также возникает ошибка сегментации.
- Проверить, находится ли страница в оперативной памяти. Если ее там нет, возникает **особая ситуация отсутствия страницы** или **страничный отказ** (**page fault**). Как правило, реакция на нее состоит в том, что вызывается специальная программа-обработчик (**trap** – ловушка), которая подкачивает требуемую страницу с диска. В многозадачных системах во время такой подкачки может исполняться другой процесс.
- Если страница есть в памяти, взять из ее дескриптора физический адрес `phys_addr`.
- Если мы имеем дело с сегментной адресацией, сравнить смещение в сегменте с длиной этого сегмента. Если смещение оказалось больше, также возникает ошибка сегментации.

- Произвести доступ к памяти по адресу `phys_addr[offset]`.

Видно, что такая схема адресации довольно сложна. Однако в современных процессорах все это реализовано в аппаратуре и, благодаря кэшу дескрипторов и другим ухищрениям, скорость доступа к памяти получается почти такой же, как и при прямой адресации. Кроме того, эта схема имеет неоценимые преимущества при реализации многозадачных ОС.

Во-первых, мы можем связать с каждой задачей свою таблицу трансляции, а значит и свое виртуальное адресное пространство. Благодаря этому даже в многозадачных ОС мы можем пользоваться абсолютным загрузчиком. Кроме того, программы оказываются изолированными друг от друга, и мы можем обеспечить их безопасность.

Для переключения задачи нужно перегрузить регистры, управляющие этой таблицей.

Это не один регистр, а, как минимум, два – указатель на таблицу и ее длина. В большинстве современных процессоров диспетчер памяти еще сложнее. Часто он работает с несколькими таблицами, может иметь дополнительные управляющие регистры и т.д.

Во-вторых, мы можем сбрасывать на диск редко используемые области виртуальной памяти программ – не всю программу целиком, а только ее часть. Кроме того, в отличие от оверлеев, программа вообще не обязана знать, какая ее часть может быть сброшена.

Другое дело, что в системах реального времени программе может быть нужно, чтобы определенные ее части никогда не сбрасывались на диск. Действительно, система реального времени обязана *гарантировать время реакции, и это гарантированное время обычно намного меньше времени доступа к диску. Естественно, что код, обрабатывающий событие, и используемые при этом данные должны быть всегда в памяти.*

В-третьих, программа не обязана занимать непрерывную область физической памяти. При этом она вполне может видеть непрерывное виртуальное адресное пространство. Это резко упрощает борьбу с фрагментацией памяти, а в системах со страничной адресацией проблема фрагментации физической памяти вообще снимается.

Так, в VAX/VMS *свободная память отслеживается при помощи битовой маски физических страниц. В этой маске свободной странице соответствует 1, а занятой – 0. Если кому-то нужна страница, система просто ищет в этой маске установленный бит. Если учесть, что* VAX *имеет специальную команду для поиска установленного бита в массиве, все работает очень быстро и просто.*

В результате виртуальное пространство программы может оказаться отображено на физические адреса очень причудливым образом, но это никого не волнует – скорость доступа ко всем страницам одинакова.

В-четвертых, система может обеспечивать не только защиту программ друг от друга, но и защиту программы от самой себя – например, от ошибочной записи данных на место кода.

В-пятых, различные задачи могут использовать общие области памяти для взаимодействия или, скажем, просто для того, чтобы работать с одной копией библиотеки подпрограмм.

Перечисленные преимущества настолько серьезны, что считается невозможным реализовать многозадачную систему общего назначения, такую как UNIX или VMS (Virtual Memory System) на машинах без диспетчера памяти.

Отдельной проблемой при разработке системы со страничной или сегментной адресацией является выбор размера страницы или максимального размера сегмента. Этот размер определяется шириной соответствующего битового поля адреса и поэтому должен быть степенью двойки.

С одной стороны, страницы не должны быть слишком большими, так как это может привести к неэффективному использованию памяти и перекачке слишком больших объемов данных при сбросе страниц на диск. С другой стороны, страницы не должны быть слишком маленькими, так как это приведет к чрезмерному увеличению таблиц трансляции, требуемого объема кэша дескрипторов и т.д.

Что будет считаться "слишком" большим или, наоборот, маленьким, в действительности зависит от среднего количества памяти, используемого программой. Так, если вы работаете со старой UNIX-системой и пользуетесь только программами типа `ls`, `grep` или `sed`, а редактор `vi` считается чем-то почти сверхъестественным, то средний размер ваших программ будет измеряться десятками килобайт и редко выходить за сотню. Если же вы используете современные графические оконные интерфейсы, систему X Windows, `emacs` и прочие красоты, то типичная программа требует около мегабайта памяти, а некоторые и по несколько десятков. Казалось бы, размер страницы при этом также должен измениться на два-три порядка. На самом деле, здесь в игру вступает еще один параметр – размер сектора на диске, с которого осуществляется подкачка...

В реальных системах размер страницы меняется от 512 байт у машин семейства VAX до нескольких килобайт. Например, i3/486 имеет страницу размером 4 К. Некоторые диспетчеры памяти, например у MC6801/2/30, имеют переменный размер страницы – в том смысле, что система при запуске программирует диспетчер и устанавливает, помимо прочего, этот размер, и дальше работает со страницами выбранного размера. У процессора i860 размер страницы переключается между 4 К и 4 Мбайтами.

С сегментными диспетчерами памяти ситуация сложнее. С одной стороны, хочется, чтобы один программный модуль влезал в сегмент, поэтому сегменты обычно делают большими, от 32К и более. С другой стороны, хочется, чтобы в адресном пространстве можно было сделать много сегментов. Кроме того, может возникнуть проблема: как быть с *большими* неразделимыми объектами, например хэш-таблицами компиляторов, под которые часто выделяются сотни килобайт. С третьей стороны, при подкачке сегментов с диска не хочется качать за один раз много данных.

Третье обстоятельство вынуждает многих разработчиков идти на двухступенчатую виртуальную память – сегментную адресацию, в которой каждый сегмент, в свою очередь, разбит на страницы. Это дает ряд мелких преимуществ, например, позволяет раздавать права доступа сегментам, а подкачку с диска осуществлять постранично. Таким образом организована виртуальная память в IBM System 370 и ряде других больших компьютеров, а также в i3/486. Правда, в последнем виртуальная память используется несколько странным образом.

Вообще говоря, i386 может работать с двумя типами адресов:

- 32-разрядным адресом, в котором 16 бит задают смещение в сегменте, 14 бит – номер сегмента, и 2 бита используются для разных загадочных целей. При этом размер сегмента не более 64К, а общий объем виртуальной памяти не превышает 1 Гбайта.
- 48-разрядным адресом, в котором смещение в сегменте занимает 32 бита. В этом случае размер сегмента может быть до 4 Гбайт, а общий объем виртуальной памяти до 2^{48} байт.

В обоих случаях сегмент может быть разбит на страницы по 4 К. При этом сегментная часть адреса и его смещение лежат в разных регистрах, и с ними можно работать раздельно. В **реальном режиме** возможность такой работы порождает весь зоопарк "моделей памяти", с которыми знакомы те, кто писал на С для MS DOS. В случае i386 большинство систем программирования выделяют программе один сегмент с 32-разрядным смещением, и программа живет там так, будто это обычная машина с 32-разрядным линейным адресным пространством. Так поступают все известные авторам реализации Unix для i386, ряд так называемых **расширителей ДОС** (**DOS extenders**), Oberon/386, Novell Netware и т.д.

С другой стороны, MS Windows в `enhanced` режиме используют i386 именно как машину с двухслойной сегментно-страничной адресацией, то есть загружают программу по сегментам, и права доступа ей выдают на сегменты, а подкачку с диска делают постранично. Это обусловлено не столько настоящей потребностью в сегментации, сколько требованием совместимости со `standard` режимом, в котором MS Windows работают на процессоре 80286

3.6 Страничный обмен

Подкачка, или **своппинг** (**swapping** – обмен), – это процесс сброса редко используемых областей виртуального адресного пространства программы на диск или другое устройство массовой памяти. Такая массовая память всегда намного дешевле оперативной, хотя и намного медленнее. Во многих учебниках по ОС приводятся таблицы следующего вида:

При разработке системы всегда есть желание сделать память как можно быстрее. С другой стороны, потребности в памяти очень велики и постоянно растут. Современные персональные системы имеют около 500 Мбайт дисковой памяти, и этого часто оказывается недостаточно, особенно если идет работа с Multimedia или просто с высококачественными изображениями.

Существует эмпирическое наблюдение, что *любой* объем дисковой памяти будет полностью занят за две недели. Жизненный опыт обоих авторов подтверждает его.

Тип памяти	Время доступа	Цена 1Мбайта	Способ использования
Статическая память	15нс	$200	Регистры, кэш-память
Динамическая память	70 нс	$30 *(4Mb SIMM)*	Основная память
Жесткие магнитные диски	1-10 мс	$3 *(1.2Gb EIDE)*	Файловые системы, устройства своппинга
Магнитные ленты	секунды	$0.025 *(8mm Exabyte)*	Устройства резервного копирования

Очевидно, что система с 500 М статического ОЗУ будет иметь стоимость, скажем так, совершенно не персональную, не говоря уже о габаритах, потребляемой мощности и прочем. К счастью, далеко не все, что хранится в памяти системы, используется одновременно. В каждый заданный момент исполняется только часть программного обеспечения, и оно работает только с частью данных.

Статистика утверждает, что в пределах одной программы 90% времени исполняется код, который занимает 10% места, а остальные 90% кода исполняются только 10% времени. Для данных разница в частоте использования, по-видимому, не столь резкая, но также существует.

Это приводит нас к идее многослойной или многоуровневой памяти, когда в быстрой памяти хранятся часто используемые код или данные, а редко используемые постепенно мигрируют на более медленные устройства. В случае дисковой памяти такая миграция осуществляется вручную, когда администратор системы сбрасывает на ленты редко используемое "барахло" и заполняет освободившееся место чем-то нужным. Для больших и сильно загруженных систем существуют специальные программы, которые определяют, что является барахлом, а что нет. Управление миграцией из ОЗУ на диск иногда осуществляется пользователем, но часто это оказывается слишком утомительно. В случае кэш-памяти и ОЗУ делать что-то вручную просто физически невозможно.

Естественно, для того чтобы автоматизировать процесс удаления "барахла" – редко используемых данных и программ, – мы должны иметь какой-то легко формализуемый критерий, по которому определяется, какие данные считаются редко используемыми. Для ручного переноса данных критерий очевиден – нужно удалять[6] то, что дольше всего не будет использоваться в будущем. Конечно, любые предположения о будущем имеют условный характер, все может неожиданно измениться. Тем не менее, обычно человек располагает такой информацией о системе и ее использовании, которая принципиально недоступна самой этой системе. А для автомата будущее неизвестно, и он должен делать догадки о будущем только на основании данных об использовании системы в прошлом.

Алгоритм автомата должен быть как можно более простым. Например, кэш-контроллер обыч-

[6]Вообще говоря, под **удалением** (**removal**) почему-то обычно понимают разрушение данных. В данном разделе мы временно вернем этому слову исходный смысл: **удалить** что-либо будет означать: перенести это на более медленный физический носитель, из кэша – в ОЗУ, из ОЗУ – на диск, с диска – на магнитную ленту или другое дешевое и медленное устройство массовой памяти.

но делается на основе “жесткой логики”, потому что он должен быть очень быстрым. Даже микропрограммный автомат с точки зрения скорости оказывается недопустимым, не говоря уже о сложных алгоритмах, использующих динамические структуры данных. Для алгоритма управления страничной подкачкой простота также важна – очень не хочется, чтобы большую часть времени система занималась размышлениями о том, какую страницу можно сбросить на диск. Сравните с армейским афоризмом: “командир должен уметь принимать *быстрое* решение, а если оно случайно окажется правильным, то это будет вообще замечательно” (цит. по [5]).

Один простой критерий выбора очевиден – при прочих равных условиях, в первую очередь, мы должны выбирать в качестве **жертвы** (**victim**) для удаления тот объект, который не был изменен за время жизни в быстрой памяти. Действительно, вы скорее удалите с винчестера саму игрушку (если у вас есть ее копия на дискетах), чем файлы сохранения!

Самый простой алгоритм – выкидывать случайно выбранный объект. Такой алгоритм хорош тем, что очень прост – не надо набирать никакой статистики о частоте использования и т.д. Очевидно, при этом удаленный объект совершенно необязательно будет ненужным...

Можно также удалять то, что дольше всего находится в данном слое памяти. Это называется алгоритмом **FIFO** (**First In - First Out** – первый вошел - первый вышел). Видно, что это уже чуть сложнее случайного удаления – нужно запоминать, когда мы что загружали. Понятно также, что это лишь очень грубое приближение к тому, что нам требуется.

Наиболее честным будет удалять тот объект, к которому дольше всего не было обращений в прошлом **LRU** (**Least Recently Used**). Это требует набора статистики обо всех обращениях. Для страничного или сегментного управления памятью это также требует аппаратной поддержки – мы ведь не в состоянии программно отслеживать все обращения ко всем страницам без катастрофического падения производительности системы.

Такая поддержка на практике должна состоять в том, что диспетчер памяти поддерживает в дескрипторе каждой страницы счетчик обращений, и при каждой операции чтения или записи над этой страницей увеличивает этот счетчик на единицу. Это требует довольно больших накладных расходов – в ряде работ ([6]) утверждается, что они будут недопустимо большими. Поэтому такая техника применяется только в экспериментальных установках, используемых для оценки производительности тех или иных алгоритмов. Она также применяется в некоторых контроллерах кэш-памяти и программно реализованных дисковых кэшах.

Остроумным приближением к алгоритму LRU является так называемый **clock**-алгоритм. Он состоит в следующем:

- Дескриптор каждой страницы содержит бит, указывающий, что к данной странице было обращение. Этот бит иногда называют clock-битом.

- При первом обращении к странице, в которой clock-бит был сброшен, диспетчер памяти устанавливает этот бит.

- Программа, занимающаяся поиском жертвы, циклически просматривает все дескрипторы страниц. Если clock-бит сброшен, данная страница объявляется жертвой, и просмотр заканчивается – до появления потребности в новой странице. Если clock-бит установлен, то программа сбрасывает его и продолжает поиск.

Название **clock**, по-видимому, происходит от внешнего сходства процесса циклического просмотра с движением стрелки часов. Подумав, можно убедиться, что вероятность оказаться жертвой для страницы, к которой часто происходят обращения, существенно ниже.

Практически все известные авторам диспетчеры памяти предполагают использование clock-алгоритма. Такие диспетчеры хранят в дескрипторе страницы или сегмента два бита – clock-бит и признак модификации. Признак модификации устанавливается при первой записи в страницу/сегмент, в дескрипторе которой этот признак был сброшен.

Экспериментальные исследования показывают любопытный факт: реальная производительность системы довольно слабо зависит от применяемого алгоритма поиска жертвы. Статистика исполнения реальных программ говорит о том, что каждая программа имеет некоторый набор страниц, называемый **рабочим множеством**, который ей в данный момент *действительно* нужен. Размер такого набора сильно зависит от алгоритма программы, он изменяется на различных этапах исполнения и т.д., но в большинство моментов мы можем довольно точно указать его. Если все страницы рабочего набора попадают в память, то частота ошибок отсутствия страницы резко снижается. В случае, когда памяти не хватает, программе почти на каждой команде требуется новая страница, и производительность системы катастрофически – в тысячи раз – падает. В случае машин типа IBM PC/AT386 с дисковым контроллером IDE, в которых процессор задействуется при операциях с диском, это может привести, практически, к блокировке системы. Это состояние по-английски называется **overswap** или **thrashing** – чрезмерный своппинг и является крайне нежелательным.

В системах коллективного пользования размер памяти часто выбирают так, чтобы система балансировала где-то между состоянием, когда все программы держат свое рабочее множество в ОЗУ, и оверсвопом. Точное положение точки балансировки определяется в зависимости от соотношения скорости процессора со скоростью обмена с диском и с потребностями прикладных программ. Во многих старых учебниках рекомендуется подбирать объем памяти так, чтобы канал дискового обмена был загружен на 50% [6].

Еще одно эмпирическое правило приводится в документации фирмы Amdahl: сбалансированная система должна иметь по мегабайту памяти на каждый MIPS (Million of Instructions Per Second – миллион операций в секунду) производительности центрального процессора. Если система не использует память, определенную по этой формуле, есть основания считать, что процессор также работает с недогрузкой. Иными словами, это означает, что вы купили слишком мощный для ваших целей процессор и заплатили лишние деньги.

Это правило было выработано на основе опыта эксплуатации больших компьютеров четвертого поколения, в основном на задачах управления базами данных. Скорость дисковой подсистемы в этих машинах была примерно сравнима с дисковыми контроллерами современных персоналок, поэтому аналогичный критерий оценки применим и к ПК, особенно работающим под управлением систем с виртуальной памятью – OS/2, Windows NT и системами семейства Unix.

В соответствии с этим правилом, машины с процессором i486 должны иметь 8-16 мегабайт памяти, а с процессором Pentium – от 20 и более. Эти цифры подтверждаются опытом реальной эксплуатации систем на основе соответствующих процессоров.

В "персональных" операционных системах, таких как Windows 3.x и Windows 95, ситуация с памятью, и вообще с производительностью, несколько иная.

Дело в том, что пользователя ПК интересует не производительность в целом, а только время реакции текстового редактора или электронных таблиц на нажатие кнопки. С одной стороны, это вынуждает устанавливать в систему больше памяти, потому что страничный обмен во время отработки команды в редакторе существенно снижает время реакции и, соответственно, наблюдаемую скорость системы. Поэтому память часто подбирают так, чтобы используемые программы и их данные входили в память целиком, а не только рабочие множества. Но, так как пользователь обычно работает только с одной программой, часто оказывается возможным ограничиться 4 или 8 мегабайтами памяти. С другой стороны, низкокачественные прикладные программы часто вынуждают пользователей покупать чрезмерно мощные процессоры, чтобы обеспечить приемлемое время реакции. Так например, редактор MS Word 6.0 требует процессора класса i486/Pentium для редактирования текстов – работы, с которой вполне справляется даже 8-разрядный процессор типа Z-80.

Поэтому во многих конторах часто можно увидеть машину класса Pentium с 8 мегабайтами памяти, используемую только для печати бумажек, то есть очень сильно разбалансированную и очень сильно недогруженную по стандартам систем общего назначения.

Глава 4

Параллельное и псевдопараллельное исполнение

4.1 Выгоды многозадачности и многопроцессности

Вычислительные системы первого поколения были полностью однозадачными. Программист набирал свою программу тумблерами на консоли, а чуть позже – вводил ее с пакета перфокарт. Все это время система была в его личном распоряжении, то есть в некотором смысле, являлась персональным компьютером. По мере того как загрузка компьютеров росла, у системных администраторов возникло желание как-то распараллелить этот процесс – пока один пользователь вводит свои перфокарты, у другого исполняется программа, у третьего печатается результат и т.д.

Развитие этой идеи привело к системам коллективного пользования – много пользователей одновременно вводят свои пакеты перфокарт на нескольких считывателях, несколько принтеров печатают "простыни", а в системе одновременно живет несколько заданий, каждое из которых ждет окончания выдачи строки на принтер, или пока промотается до нужного места лента на магнитофоне, или пока другая задача освободит центральный процессор, или какого-то другого внешнего или внутреннего события.

Это резко увеличило эффективность использования дорогостоящих вычислительных систем и, с другой стороны, сделало редкие тогда компьютеры доступными более широкому кругу пользователей. К концу 60-х годов появились так называемые системы разделенного времени, когда несколько пользователей интерактивно работали с одной системой при помощи терминалов. Для мощных систем число таких пользователей доходило до нескольких сотен одновременно. Естественно, обеспечивать нормальную работу в таком режиме может только многопроцессная ОС.

Разработчикам интерактивных программ известно, что самым медленным элементом интерактивной системы обычно является человек. Большую часть своей жизни интерактивная программа ожидает ввода с клавиатуры или движения мышки. Если в системе исполняется только одна программа, то система при этом, в сущности, простаивает. Поэтому даже для однопользовательских – **персональных** – систем часто оказывается удобным иметь возможность работы в многозадачном режиме.

В системах реального времени возможность заниматься каким-то полезным делом во время ожидания внешнего события часто оказывается жизненно необходимой.

Теоретически можно возложить задачу распараллеливания на пользовательскую программу – так поступают некоторые разработчики программ реального времени для MS DOS или в некоторых кросс-системах.

Однако многопроцессность создает ряд специфических проблем, и оказывается целесообразным реализовать решения этих проблем один раз, собрать код, решающий их, в единый модуль и объявить этот модуль ядром ОС. Именно таким образом построено ядро систем RT-11, OS-9 и ОС, базирующихся на технологии **микроядра** (**microkernel**) – QNX, UNIX System V R4, HURD.

В других ОС в ядро помещен также код, занимающийся работой с внешними устройствами, и даже менеджеры файловых систем. Такие системы сейчас называются "монолитными". Выгоды и недостатки обоих решений будут обсуждаться ниже.

4.2 Проблемы при параллельной работе

Первой из проблем является проблема разделения ресурсов. Действительно, представим себе две программы, пытающиеся печатать что-то на одном принтере. Если они будут делать это произвольным образом, их вывод на бумаге будет перемешан и скорее всего окажется совершенно нечитаемым. Одним из разумных решений может быть монопольный захват принтера одной из программ. При этом другая программа будет вынуждена ждать, пока принтер не освободится. Значит, нужны средства для захвата ресурсов и ожидания их освобождения[1].

Другая проблема – это проблема **реентерабельности** (**reenterability** – повторной входимости, от re-enter) разделяемых программ.

Представим себе, что две задачи используют общий программный модуль. Примером такого модуля может являться само ядро ОС. Представим себе, что одна из программ вызвала процедуру из разделяемого модуля. После этого произошло переключение задач, и вторая задача обращается к тому же модулю.

Существует техника реализации этого модуля, при которой такой вызов не создаст никаких проблем. Такая техника состоит в том, что все данные, с которыми работает разделяемая программа, хранятся в ее локальных переменных. Для каждой инкарнации программы создается собственная копия таких данных. Такое требование легко выполняется для функций, не имеющих побочного эффекта, таких как вычисление синуса или поиск символа в текстовой строке.

[1]На практике для разделения принтеров используют более сложные и удобные средства, обсуждаемые в разделе 7.3.7.

При других техниках программирования могут возникнуть серьезные проблемы, вплоть до развала системы. Модули, которые можно вызывать многократно, называются **реентерабельными**. Соответственно программы, которые так вызывать нельзя, называются нереентерабельными. Типичный пример нереентерабельной программы – ядро дисковой операционной системы MS DOS.

Легко понять, что программы, управляющие внешними устройствами или файловыми системами и вообще работающие с разделяемыми объектами, *не могут* быть реентерабельными.

Во многих случаях оказывается выгодно рассматривать нереентерабельную программу как разделяемый ресурс и распределять доступ к нему теми же способами, что и к принтеру. Так поступает большинство известных авторам средств организации параллельной работы под MS DOS. Таким же образом микроядро организует доступ пользовательских программ к модулям ОС.

Очень широкий класс проблем – проблемы синхронизации, возникающие при попытках организовать взаимодействие нескольких программ.

Первая, самая простая из них, состоит в вопросе – если одна задача производит данные, а вторая их потребляет, то как задача-потребитель узнает, что готова очередная порция данных? Или, что еще интереснее, как она узнает, что очередная порция данных еще не готова? Типичный случай такого взаимодействия – асинхронное чтение с диска, когда программа дает дисковому драйверу запрос: “читай с такого-то сектора в такой-то блок памяти”, и продолжает заниматься своими делами. Это режим работы, поддерживаемый всеми ОС линии RT-11 - RSX-11 - VAX/VMS - OpenVMS и многими системами реального времени.

Другая проблема называется проблемой критических секций. Например, одна программа вставляет данные в разделяемый двунаправленный список, а другая достает их оттуда. Те, кто знаком с алгоритмом вставки в список, легко поймут, что есть момент, когда указатели элементов показывают вовсе не туда, куда надо. Попытка произвести в этот момент какую-то другую операцию изменения списка приведет к полному разрушению его структуры, а чтение или поиск закончатся аварией. Поэтому изменяющая программа должна каким-то образом блокировать доступ к списку на время изменения. Часто это делается теми же средствами, что и разделение ресурсов.

Большинство решений всех вышеперечисленных проблем сводится к созданию какого-то средства, сообщающего программе, что произошло то или иное внешнее или внутреннее событие. При этом программа может остановиться, ожидая заданного события.

4.3 Методы синхронизации

4.3.1 Прерывания и сигналы

Я слышу крик в темноте,
Возможно, это сигнал

Nautilus Pompilius

Исторически первым методом сообщения системе о внешнем событии является аппаратное прерывание. Идея прерывания состоит в том, чтобы в момент события вызывать некоторую подпрограмму, которая и выполнит действия по обработке события.

Слово **прерывание** представляет довольно неудачную, на взгляд авторов, кальку с англоязычного термина **interrupt** (существительное от глагола **прервать**).

Прерывания, вызванные внутренними событиями, часто называют **исключениями** (**exceptions**). Мы далее будем разделять эти два понятия, то есть внешние прерывания будем называть просто прерываниями, а внутренние – исключениями. Исключения возникают при делении на ноль, неопределенном коде команды, ошибках обращения к памяти и т.д.

Реализации прерываний и исключений у разных процессоров немного отличаются.

Для примера рассмотрим организацию прерываний в машинах семейства PDP-11. Машины данной архитектуры сейчас почти не используются, но ряд архитектурных решений не потерял актуальности и поныне. В частности, подход к реализации прерываний считается классическим.

Процессоры семейства PDP-11 различают 128 типов прерываний и исключений. Каждому типу соответствует процедура - обработчик. Адреса точек входа всех процедур собраны в таблицу, называемую **таблицей векторов прерываний**. Эта таблица занимает 256 слов физической памяти, начиная с нулевого адреса. Каждый элемент таблицы (**вектор**) содержит адрес обработчика и новое слово состояния процессора. Ниже будет объяснено, для чего это сделано.

Процессор узнает о возникновении прерывания, если на один из входов запроса прерывания подан сигнал. Обычно этот сигнал генерируется одним из внешних устройств. Например, прерывание может сигнализировать о завершении операции перемещения головки дисковода, и т.д.

Каждый вход соответствует определенному уровню приоритета. PDP-11 имеет восемь уровней приоритета прерывания. Прерывание происходит только когда уровень приоритета процессора ниже приоритета запрашиваемого прерывания. Если у процессора установлен приоритет 7, внешние прерывания запрещены. Приоритет процессора задается его словом состояния.

Получив запрос, процессор завершает исполнение текущей команды и выставляет сигнал готовности к прерыванию. После этого внешнее устройство выставляет на шине данных номер вектора прерывания.

Процессор считывает номер и вызывает соответствующую процедуру из таблицы. При этом вызов обработчика прерывания отличается от вызова обычной процедуры: при обычном вызове в стеке сохраняется только адрес команды, на которую следует возвратить управление. При прерывании же в стеке сохраняются два значения: адреса команды и слова состояния процессора. Новое слово состояния берется из таблицы векторов.

При этом приоритет процессора автоматически устанавливается равным тому значению, которое разработчик программы обработки считает правильным. Обратите внимание: не равным приоритету обрабатываемого прерывания, а тому, которое требует разработчик.

При завершении процедуры обработки вызывается команда `RTI` (ReTurn from Interrupt – возврат из прерывания). Эта команда выталкивает из стека адрес прерванной команды и старое слово состояния, тем самым восстанавливая приоритет процессора.

Для сравнения: в процессорах семейства i80x86 вектор прерывания содержит только адрес программы-обработчика, а приоритет процессора задается не словом состояния процессора, а регистром внешнего устройства – **контроллера прерываний**. Контроллер прерываний обычно устанавливает приоритет равным приоритету прерывания, обрабатываемого в данный момент. Чтобы повысить или понизить этот уровень, обработчик прерывания должен программировать контроллер. Перед завершением обработчика необходимо вернуть контроллер прерываний в исходное состояние, выполнив над ним серию магических команд – **эпилог прерывания**.

Обработка прерываний в системах с виртуальной памятью несколько усложняется: ведь кроме адреса обработчика нам надо еще задать адресное пространство, в котором этот адрес определен.

В моделях PDP-11, имеющих диспетчер памяти, эта проблема решается просто: для процессора в каждый момент времени заданы два адресных пространства: пользовательское и системное. Все прерывания обрабатываются в системном адресном пространстве.

В защищенном режиме процессоров i80x86 использован более гибкий механизм установки адресного пространства для обработчика. По существу, с каждым обработчиком может быть ассоциировано свое виртуальное адресное пространство. О способе, которым это достигается, лучше прочитать в литературе по соответствующим процессорам, например [20].

Аналогичное прерываниям средство реализовано в ряде ОС – например, в UNIX – для чисто программных событий. Это средство называется **сигналами** и используется для обработки исключительных ситуаций и, в некоторых случаях, для межпроцессной коммуникации. Сигналы в UNIX часто используются для принудительного завершения программы, и даже команда посылки сигнала называется `kill` – **убить**.

Легко видеть, что прерывания и сигналы могут служить для оповещения программы о событии, но не решают ни одной из проблем, перечисленных в разделе 4.2, а напротив, создают их. Действительно, подпрограмму обработки прерывания во многих отношениях можно рассматривать как параллельно исполняемый процесс, и к ней вполне приложимо все, что сказано в предыдущем разделе.

К счастью, аппаратные реализации позволяют *запрещать* все или некоторые прерывания, и мы можем решить таким образом проблему критической секции или даже доступа к нереентерабельной процедуре. Однако, такое решение часто оказывается неудовлетворительным или просто нереализуемым. Например, в системах реального времени нельзя запрещать прерывания надолго. Поэтому программа обработчика прерывания в плохо спроектированной ОС, в ДОС или на "голой"машине часто вынуждена заниматься тонкой игрой аппаратными уровнями прерывания. Такая игра доставляет много удовольствия молодым хакерам, но может приводить к тонким и очень труднообнаружимым ошибкам.

Эти ошибки состоят в неправильном определении границ критических секций. Они очень плохо воспроизводятся при тестировании, потому что для срабатывания ошибки необходимо возникновение прерываний в строго определенной последовательности и в заданные относительные моменты времени. Поймать такую ошибку под интерактивным отладчиком практически невозможно.

Кроме того, программу, работающую совместно с процедурами обработки прерываний, нельзя представить в виде детерминированного конечного автомата. Это усложняет анализ алгоритмов и доставило в свое время много волнений теоретикам программирования. Например, в [7] Дийкстра очень эмоционально описывает свою реакцию при первом столкновении с системой, использующей прерывания.

С практической точки зрения, наиболее серьезным недостатком прерываний является то, что прерывания предоставляют только механизм сообщения программе о том, что какое-то событие произошло.

Но как программа узнает, что события еще не происходило?

Действительно, вернемся к примеру из предыдущего раздела: программа **A** производит данные, а программа **B** их потребляет. Наиболее простым решением было бы объединить эти программы в один процесс. Тогда программа **A**, произведя очередную порцию данных, вызывала бы программу **B**, и ждала бы, пока та пережует эту порцию. Или наоборот, программа **B**, обнаружив, что ей нужны очередные данные, вызывала бы программу **A**. Понятно, что в этом случае мы отказываемся от всех преимуществ многопроцессности.

Если мы хотим, чтобы программы **A** и **B** исполнялись параллельно, программа **B** должна быть способна узнать, что очередная порция данных готова. Например, мы можем создать флаг, который равен нулю, если данные не готовы, и единице, если готовы. Тогда модель взаимодействия выглядит так:

- Программа **B**, когда ей нужны очередные данные, проверяет флаг. Если флаг равен 1, данные готовы. Если не равен, то нужно заснуть до получения сигнала.
- Программа **A**, подготовив очередную порцию данных, посылает сигнал программе B.
- Обработчик сигнала устанавливает флаг в 1 и возобновляет выполнение программы **B**.

Если мы запишем алгоритм программы **B** на псевдокоде, он будет выглядеть примерно так:

Пример 5

```
extern boolean flag;

  set_signal_handler(DATA_READY, sighandler);
  while(1) {
    .... // Обработка очередной порции данных
    if (flag != TRUE) {
     /* Обратите внимание, что проверка флага *
      * и засыпание - это разные операторы!   */
      pause();
    }
    .... // Данные готовы!
  } // End while

void sighandler() {
  flag = TRUE;
}
```

Для простоты мы опускаем обсуждение вопроса о том, кем и как флаг устанавливается в 0. Нам и без этого хватит тем для обсуждения. Внимательный читатель должен был задаться вопросом: что будет, если обработчик сигнала активизируется в интервале между операторами `if` и `pause()`? Хотя такое событие маловероятно, оно приведет к засыпанию программы **B** навсегда!

Напрашивающееся решение состоит в том, чтобы запретить сигнал `DATA_READY` на время проверки флага. Но мы все равно должны разрешить сигнал перед засыпанием, поэтому это, на самом деле, вовсе не решение.

Одно из решений состоит в переделке программы таким образом:

Пример 6

```
extern boolean flag;

  set_signal_handler(DATA_READY, sighandler);
  while(1) {
    .... // Обработка очередной порции данных
    label:
    if (flag != TRUE) {
      pause();
    }
    .... // Данные готовы!
  } // End while

void sighandler() {
  flag = TRUE;
  goto label;
}
```

Для простоты мы не обсуждаем вопрос о том, как можно делать передачу управления между функциями. В языке C это может быть сделано библиотечными функциями `setjmp/longjmp`. В языках Ada и C++ для этого используется механизм обработки исключений. На ассемблере мы должны не только передать управление, но и восстановить стек в то состояние, которое он имел во время вызова целевой функции. Желающие исследовать этот вопрос подробнее могут прочитать руководство по соответствующим языкам программирования или исследовать исходные тексты функций `setjmp/longjmp`, входящие в поставку Borland C/C++ 3.x или 4.x, ОС BSD UNIX, Linux и т.д.

Текст 6, на самом деле, содержит еще более серьезную ошибку: что будет, если обработчик активизируется до окончания обработки данных? Эта проблема также имеет решение, но мы не будем далее обсуждать эту идею, а скажем лишь, что код, содержащий решения всех возникающих проблем, занял бы около страницы.

Именно поэтому Справочное руководство программиста ОС UNIX не рекомендует пользоваться сигналами для синхронизации процессов.

4.3.2 Семафоры

Если мы хотим спроектировать простой и удобный механизм синхронизации, лучше всего объединить проверку флага и засыпание процесса в единую операцию, которая не может быть прервана исполнением другого процесса. Распространив эту идею на случай взаимодействия более чем двух процессов, мы приходим к механизму, известному под названием **семафоров Дийкстры**.

Семафор Дийкстры представляет собой целочисленную переменную, с которой ассоциирована очередь ожидающих процессов. Пытаясь пройти через семафор, процесс пытается вычесть из значения переменной 1. Если значение переменной больше или равно 1, процесс проходит сквозь семафор успешно (семафор открыт). Если переменная равна нулю (семафор закрыт), процесс останавливается и ставится в очередь.

Закрытие семафора соответствует захвату ресурса, доступ к которому контролируется этим семафором. Процесс, закрывший семафор, захватывает ресурс. Если ресурс захвачен, остальные процессы вынуждены ждать его освобождения. Закончив работу с ресурсом, процесс увеличивает значение семафора на единицу, открывая его. При этом первый из стоявших в очереди процессов активизируется, вычитает из значения семафора единицу, и снова закрывает семафор. Если же очередь была пуста, то ничего не происходит, просто семафор остается открытым. Тогда первый процесс, подошедший к семафору, успешно пройдет через него. Это действительно похоже на работу железнодорожного семафора, контролирующего движение поездов по одноколейной ветке.

Наиболее простым случаем семафора является **двоичный** семафор. Начальное значение флаговой переменной такого семафора равно 1, и вообще она может принимать только значения 1 и 0. Двоичный семафор соответствует случаю, когда с разделяемым ресурсом в каждый момент времени может работать только одна программа.

Семафоры общего вида могут принимать любые неотрицательные значения. Это соответствует случаю, когда несколько процессов могут работать с ресурсом одновременно, или когда ресурс состоит из нескольких независимых, но равноценных частей – например, несколько одинаковых принтеров. При работе с такими семафорами часто разрешают процессам вычитать и добавлять к флаговой переменной значения, большие единицы. Это соответствует захвату/освобождению нескольких частей ресурса.

При доступе к нескольким различным ресурсам с использованием семафоров возникает специфическая проблема, называемая **мертвой блокировкой** (**dead lock**). На этой проблеме стоит остановиться подробнее, потому что она возникает и при многих других методах синхронизации, но в случае семафоров ее легче объяснить.

Рассмотрим две программы, использующие доступ к двум различным ресурсам. Например, один процесс копирует данные со стримера на кассету Exabyte, а другой – в обратном направлении. Доступ к стримеру контролируется семафором `sem1`, а к кассете – семафором `sem2`.

Первая программа сначала закрывает семафор `sem1`, затем `sem2`. Вторая программа поступает наоборот. Поэтому, если вторая программа получит управление и защелкнет `sem2` в промежутке между соответствующими операциями первой программы, то мы получим мертвую блокировку – первая программа никогда не освободит `sem1`, потому что стоит в очереди у `sem2`, занятого второй программой, которая стоит в очереди у `sem1`, занятого первой... Все остальные программы, пытающиеся получить доступ к стримеру или кассете, также будут становиться в соответствующие очереди и ждать, пока администратор не убьет одну из защелкнувшихся программ.

Эта проблема может быть решена несколькими способами. Первый способ – разрешить программе в каждый момент времени держать закрытым только один семафор – прост и решает проблему в корне, но часто оказывается неприемлемым. Более приемлемым оказывается соглашение, что семафоры всегда должны закрываться в определенном порядке. Этот порядок может быть любым, важно только чтобы он всегда соблюдался. Третий, наиболее радикальный, вариант состоит в предоставлении возможности объединить семафоры и/или операции над ними в неразделяемые группы. При этом программа может выполнить операцию закрытия семафоров `sem1` и `sem2` единой командой, во время исполнения которой никакая другая программа не может получить доступ к этим семафорам.

Многие ОС предоставляют для синхронизации семафоры Дийкстры или похожие на них механизмы.

Так, например, в системах RSX-11 и VMS основным средством синхронизации являются **флаги событий** (**event flags**). Процессы и система могут **очищать** (**clear**) или **взводить** (**set**) эти флаги. Флаги делятся на локальные и глобальные. Локальные флаги используются для взаимодействия между процессом и ядром системы, глобальные – между процессами. Процесс может остановиться, ожидая установки определенного флага, поэтому флаги во многих ситуациях можно использовать вместо двоичных семафоров. Кроме того, процесс может связать с флагом события процедуру-обработчик **AST** (**Asynchronous System Trap** – Асинхронно [вызываемый] системный обработчик). AST во многом напоминают сигналы или аппаратные прерывания.

В частности, флаги событий используются для синхронизации пользовательской программы с асинхронным исполнением запросов на ввод/вывод. Исполняя запрос, программа задает локальный флаг события. Затем она может остановиться, ожидая этого флага, который будет взведен после исполнения запроса. При этом мы получаем псевдосинхронный ввод/вывод, напоминающий синхронные операции чтения/записи в UNIX и MS DOS. Но программа может и не останавливаться! При этом запрос будет исполняться параллельно с исполнением самой программы, и она будет оповещена о завершении операции соответствующей процедурой `AST`.

Любопытно, что в этом случае процедуре `AST` не нужно устанавливать никаких флаговых переменных, а программе не нужно проверять их – она может просто посмотреть значение флага события. Поэтому ничего похожего на проблему в программе 5 просто не возникает.

Асинхронный ввод/вывод часто жизненно необходим в программах реального времени, но бывает полезен и в других случаях. Например, реализованная С.Ковалевым программа просмотра файлов для VAX/VMS при запуске считывает и форматирует столько данных, чтобы заполнить экран. Затем она ожидает команд пользователя и одновременно считывает и форматирует данные на несколько экранов вперед. Если пользователь дает команду "следующий экран", то показываются заранее отформатированные данные. За счет этого достигается быстрый запуск и очень малое время реакции на команды, то есть очень высокая субъективная "скорость".

Можно показать, что любые проблемы взаимодействия и синхронизации процессов могут – с большими или меньшими трудностями – быть решены при помощи семафоров. На практике трудности иногда оказываются не то, чтобы слишком большими, но нежелательными. Поэтому большинство ОС кроме семафоров предоставляет и другие средства синхронизации.

Так например, семафоры удобны при синхронизации доступа к единому ресурсу, такому как принтер. Если же нам нужна синхронизация доступа к ресурсу, имеющему внутреннюю структуру, такому как файл с базой данных, удобнее оказываются другие методы.

4.3.3 Блокировка участков файлов

Если говорить именно о файле с базой данных, оказывается удобно блокировать доступ к участкам файла. При этом целесообразно ввести два типа блокировок: **на чтение** и **на запись**. Блокировка на чтение разрешает другим процессам читать из заблокированного участка и даже ставить туда такую же блокировку, но запрещает писать в этот участок и, тем более, блокировать его на запись. Этим достигается уверенность в том, что структуры данных, считываемые из захваченного участка, никем не модифицируются, поэтому гарантирована их целостность и непротиворечивость.

В свою очередь, блокировка на запись запрещает всем, кроме блокирующего процесса, любой доступ к заблокированному участку файла. Это означает, что данный участок файла сейчас будет модифицироваться, и целостность данных в нем не гарантирована.

Блокировка участков файла в качестве средства синхронизации была известна еще с шестидесятых годов, но в том виде, который описан в стандартах ANSI и POSIX, она была реализована в ОС UNIX в начале 70-х. Этот стандарт поддерживается практически всеми современными многопроцессными ОС и даже MS/DR DOS при работе в сети или в MS Windows.

В UNIX возможны два режима блокировки: **допустимая** (**advisory**) и **обязательная** (**mandatory**). Как та, так и другая блокировка может быть блокировкой на чтение либо на запись. Допустимая блокировка является "блокировкой для честных": она не оказывает влияния на подсистему ввода/вывода, поэтому программа, не проверяющая блокировок или игнорирующая их, сможет писать или читать из заблокированного участка без проблем. Обязательная блокировка требует бо̀льших накладных расходов, но запрещает физический доступ к файлу: чтение или запись, в зависимости от типа блокировки.

При работе с разделяемыми структурами данных в ОЗУ было бы удобно иметь аналогичные средства, но их реализация ведет к большим накладным расходам, даже на системах с виртуальной памятью, поэтому ни одна из известных авторам систем таких средств не предоставляет. Впрочем, в современных версиях системы UNIX есть возможность отображать файл на виртуальную память. Используя при этом допустимую блокировку участков файла, программы могут синхронизовать доступ к нему (обязательная блокировка делает невозможным отображение на память).

4.3.4 Гармонически взаимодействующие последовательные процессы

Разделяемые структуры данных являются предметом ненависти теоретиков и источником серьезных ошибок при разработке программ. Легко показать, что критические секции может иметь только программа, работающая с разделяемыми структурами данных. И этими критическими секциями как раз и являются места, где программа модифицирует такие структуры или просто обращается к ним. Синхронизация доступа к разделяемым структурам часто приводит к усложнению программы, а стремление сократить участки исключительного доступа – к ошибкам.

Желание устранить эти проблемы привело в свое время Дийкстру к концепции, известной как **гармонически взаимодействующие последовательные процессы**. Эта концепция состоит в следующем:

- Каждый процесс представляет собой независимый программный модуль, для которого создается иллюзия чисто последовательного исполнения.
- Процессы не имеют разделяемых данных.
- Все обмены данными, и вообще взаимодействие, происходят в выделенных точках процессов. В этих точках процесс, передающий данные, останавливается и ждет, пока его партнер будет готов эти данные принять. В некоторых реализациях процесс-передатчик может не ожидать приема, а просто складывать данные в системный буфер. Аналогично, процесс, принимающий данные, ожидает, пока ему передадут данные. Иными словами, все передачи данных неразрывно связаны с синхронизацией.
- Синхронизация, не сопровождающаяся передачей данных, просто лишена смысла – процессы, не имеющие разделяемых структур данных, совершенно независимы и не имеют ни критических точек, ни нереентерабельных модулей.

Концепция гармонически взаимодействующих процессов очень привлекательна с теоретической точки зрения и позволяет легко писать правильные программы. Однако часто по соображениям производительности оказывается невозможно отказаться от разделяемой памяти. В этом смысле разделяемая память напоминает предмет ненависти структурных программистов – оператор `goto`. И то, и другое является потенциальным источником ошибок и проблем, но часто без них оказывается нельзя обойтись.

В современных системах реализован целый ряд средств, которые осуществляют передачу данных одновременно с синхронизацией: **почтовые ящики** (**mailboxes**) в системах линии RSX-11 - VMS, **трубы** (**pipes**)[2] в UNIX, **рандеву** (**rendesvous** – свидание) в языке Ada, транспьютерные линки и т.д. Они будут обсуждаться в разделе 5.

[2]В русскоязычной литературе их часто называют программными каналами

Глава 5

Межзадачное взаимодействие

5.1 Определения

В предыдущих разделах мы достаточно свободно использовали термины “программа”, “процесс” и “задача”. Пришло время, когда мы вынуждены ввести определения.

Понятие программы интуитивно очевидно – это набор команд, который можно загрузить в память и на который можно передать управление. Часто программой называют цельный загрузочный модуль. Для систем типа MS DOS или UNIX такое определение может быть разумным, и в этом смысле мы можем назвать MS DOS многопрограммной системой, но для систем с динамической сборкой оно просто не имеет смысла.

Процесс представляет собой программу, которая исполняется последовательно. При этом ее исполнение может быть прервано передачей управления другому процессу, но после этого оно должно быть возобновлено с той точки, где прервалось. В операционных системах OS/2 и Windows NT определенному таким образом процессу соответствует понятие **нити** (**thread**).

Задача представляет собой сочетание программы и данных. При этом задачи изолированы друг от друга. Если позволяет аппаратура процессора, то задачи обычно имеют отдельные виртуальные адресные пространства. Однако мы можем говорить о задачах и на процессорах без диспетчера памяти, например, на транспьютере. Там все задачи имеют общее виртуальное адресное пространство, совпадающее с физическим, но из-за особенностей архитектуры процессора и инструментального языка Occam разделение памяти между процессами крайне затруднено и практически не используется.

Точнее говоря, задачи могут разделять код программы, но каждая задача обязана иметь собственную копию локальных (`auto` в терминах языка C) и статических (`static` и `extern`) данных. Вообще-то задачи могут иметь разделяемые данные, но такое разделение организуется специальными средствами.

Понятно, что в рамках одной задачи может исполняться несколько процессов. С другой стороны, можно привести пример многозадачной однопроцессной операционной системы – MS Windows 3.x.

Действительно, в MS Windows 3.1 в `standard` и `enhanced` режимах каждая запущенная программа формирует отдельную задачу со своим адресным пространством. Однако для этой программы не создается иллюзии последовательного исполнения. Напротив, программа представляет собой набор **обработчиков событий** – объектов, часто представляющих собой простые конечные автоматы. С каждым таким объектом связан соответствующий модуль программы. В документации такой модуль называется **callback**.

По мере возникновения внешних событий ядро системы – **менеджер событий** – вызывает их обработчиков. При этом каждый обработчик вызывается вполне синхронно, то есть менеджер передает управление и ждет, пока тот его возвратит, как при обычном вызове процедуры в последовательной программе. Поэтому, в соответствии с нашими определениями, менеджер событий вместе с обработчиками является единственным в системе последовательный процессом, который исполняется попеременно различными задачами. Подробнее архитектура Windows описана в 5.4.

Во многих операционных системах, например в UNIX, понятия задачи и процесса почти совпадают – с каждым процессом ассоциировано свое виртуальное адресное пространство. Новый процесс формируется системным вызовом `fork`, который также формирует и новое виртуальное адресное пространство.

Вообще, вызов `fork` является достаточно любопытной конструкцией. Этот вызов создает две задачи, которые в первый момент полностью идентичны, у них различается только значение, возвращенное вызовом `fork`. Типичная программа, использующая этот вызов, выглядит так:

Пример 7

```
int pid; /* Идентификатор порожденного процесса */

switch(pid = fork())
{
    case 0: /* Порожденный процесс */
       ......
    break;
    case -1: /* Ошибка */
       perror("Cannot fork");
       exit(1);
    default: /* Родительский процесс */
       ......
    /* Здесь мы можем ссылаться на порожденный процесс,
     *  используя значение pid */
}
```

При этом каждый из процессов имеет свою копию всех локальных и статических переменных. На процессорах со страничным диспетчером памяти физического копирования не происходит. Изначально оба процесса используют одни и те же страницы памяти, а дублируются только те из них, которые были изменены.

При этом, если мы хотим запустить другую программу, то мы должны исполнить системный вызов из семейства `exec`. Вызовы этого семейства различаются только способом передачи параметров. Все они прекращают исполнение текущего процесса и создают новый процесс/задачу с новым виртуальным адресным пространством, но с тем же идентификатором процесса. При этом у нового процесса будет тот же приоритет, будут открыты те же файлы (это часто используется), и он унаследует ряд других важных характеристик, поэтому часто говорят, даже в документации, что это тот же самый процесс. Во всяком случае, после `exec` формируется совершенно новая задача. Поэтому запуск другой программы в UNIX выглядит примерно так:

Пример 8

```
int pid; /* Идентификатор порожденного процесса */

switch(pid = fork())
{
   case 0: /* Порожденный процесс */
      dup2(1, open("ls.log", O_WRONLY | O_CREAT));
      /* Перенаправить открытый файл #1
       * (stdout) в файл ls.log */

      execl("/bin/sh", "sh", "-c", "ls", "-l", 0);

   /* Сюда мы попадаем только при ошибке! */
   /* fall through */
   case -1: /* Ошибка */
       perror("Cannot fork or exec");
       exit(1);
   default: /* Родительский процесс */
       ......
   /* Здесь мы можем ссылаться на порожденный
    * процесс, используя значение pid */
}
```

Программа в примере 8 запускает командный интерпретатор `/bin/sh`, известный как Bourne shell, приказывает ему исполнить команду `ls -l` и перенаправляет стандартный вывод этой команды в файл `ls.log`.

Техника программирования, основанная на `fork/exec`, несколько менее красива, чем принятая в современных системах реального времени или в языке Ada, где новый процесс исполняет свою программу, а не ту же самую, что и родительский. Кроме того, на системах без диспетчера памяти `fork` требует копирования адресных пространств, что приводит к большим накладным расходам, да и просто не всегда возможно.

Наиболее простым аргументом в пользу `fork` является то, что на основе `fork` достаточно легко можно смоделировать описанную выше функцию `processCreate(void (*processBody)())`, а на основе `processCreate` смоделировать `fork` нельзя. Такой аргумент звучит не очень серьезно, пока мы не вспомним, что на основе этой техники написано довольно много простых и удобных прикладных и системных программ, например, тот же Bourne shell. Поэтому функция `fork` внесена в стандарт POSIX и, вообще, охраняется законом и традициями.

Более важная причина, которая, по-видимому, и заставила отцов-основателей Unix остановиться на `fork/exec` вместо `processCreate` – это… количество параметров процедуры `processCreate`, которое могло бы оказаться необходимым. Действительно, в промежутке между вызовами `fork` и `exec` родительская программа может настроить многие свойства порождаемой задачи. Каждому из настраиваемых таким образом свойств соответствует параметр эквивалентной функции `processCreate`. Давайте посмотрим на список настраиваемых свойств:

- Открытые файлы. При исполнении системного вызова `fork`, порожденный процесс наследует все дескрипторы файлов, открытых родителем. При вызове `exec` сохраняются все дескрипторы файлов, у которых не был установлен флаг `CLOSE_ON_EXEC`. По умолчанию, этот флаг не устанавливается.

  Переназначение первых трех файловых дескрипторов (`0 - stdin`, `1 - stdout`, `2 - stderr`) широко используется командными процессорами Unix и другими программами для переназначения ввода/вывода и объединения процессов в конвейеры.

  Наследование остальных дескрипторов используется реже, но такие программы известны – например, демон HTTP версии Netscape.

- Управляющий терминал и идентификаторы сессии и группы процессов. Многие командные процессоры используют эти идентификаторы для управлении заданиями.

- **Среда** (**environment**) – список текстовых строк формата `NAME=VALUE`. Программа может получить значение переменной с заданным именем прямым поиском в этом списке или вызовом библиотечной функции `getenv`. Некоторые из переменных среды оказывают влияние на поведение всех программ; например, переменная `TERM` обозначает тип используемого терминала и используется многими программами, особенно экранными редакторами, для настройки на этот терминал. Другие переменные оказывают влияние только на некоторые программы: так, например, переменная `LIBPATH` указывает редактору связей `ld`, в каких директориях искать объектные библиотеки.

  Пользователи MS/DR DOS и OS/2 должны быть знакомы с переменными среды.

  По умолчанию, все переменные среды родительской задачи наследуются порожденной. Однако существуют два способа, которым родитель может изменить среду новой задачи. Во-первых, некоторые формы вызова `exec` получают в качестве параметра новый список переменных среды. Во-вторых, порожденный процесс перед вызовом `exec` может сделать один или несколько вызовов функции `putenv`, или модифицировать свою среду “вручную". При этом все изменения автоматически унаследуются порожденной задачей, а среда родительской задачи не изменится.

- Сигнальная маска. Задача может игнорировать отдельные сигналы, задерживать их обработку или устанавливать функцию-обработчик. Операции игнорирования и задержки определяются сигнальной маской. При `fork` сигнальная маска и обработчики наследуются; при `exec` сигнальная маска наследуется, а обработчики сбрасываются: ведь в адресном пространстве новой задачи уже не будет кода обработчиков.

- Текущая директория.

- Маска прав доступа `umask`. Права доступа к файлам в Unix задаются девятибитовой маской: права чтения, записи и исполнения для хозяина файла, группы и всех остальных. При создании файла программа задает права доступа к нему. Заданная маска прав объединяется логическим И с обращенным значением `umask`. Эта, на первый взгляд немного странная, процедура предоставляет относительно удобный способ управления правами доступа к вновь создаваемым файлам.

Эти параметры изменяются почти при каждом вызове `fork/exec`. Параметры, перечисленные ниже, изменяют не так часто, но они оказывают серьезное воздействие на исполнение порожденной задачи, и нередко бывает *необходимо* настроить их.

- Приоритет процесса.
- Корневая директория. Обычно она совпадает с корневой директорией файловой системы, но Unix позволяет переместить корневую директорию текущей задачи. Тогда задача будет видеть только поддерево общего дерева директорий. Это часто используется при создании “гостевых"входов в систему с сильно ограниченными правами.
- `ulimit` – максимальная длина файла, который может быть создан задачей. Этот параметр бывает полезен при отладке.
- Ряд других ограничительных параметров, например максимальный размер адресного пространства новой задачи.

Итак, мы получили более десяти параметров, которые следовало бы передать функции `processCreate`, не считая имени запускаемой программы и позиционных аргументов (`argc/argv`). Необходимо также учесть, что многие аргументы представляют собой не скалярные значения, а массивы, каждый элемент которых также нуждается в настройке.

В свете сказанного преимущество `fork/exec` перед `processCreate` представляется очевидным.

В соответствии с нашим определением, каждая задача имеет собственный сегмент данных. Поэтому для организации взаимодействия между задачами необходимы отдельные средства.

5.2 Разделяемая память

Самым простым из таких средств может быть организация сегмента памяти, разделяемого между задачами. В системах с виртуальной памятью такой сегмент организуется просто – мы отображаем одни и те же физические страницы на адресные пространства различных задач – и получаем разделяемый сегмент.

Самое интересное состоит в том, что можно организовать разделяемую память и для задач, исполняющихся на различных машинах. Для этого необходим блок двухпортовой памяти, подключенный к системным шинам обеих машин. Такое решение часто используется в многопроцессорных системах, особенно когда нам нужно организовать быстрый обмен большими (десятки мегабайт) объемами данных. Например, по такой схеме организован обмен между основным(и) процессорами и видеопроцессором в графических станциях Silicon Graphics.

Можно создать разделяемую между машинами память и в том случае, если машины не имеют физической общей памяти. Для этого необходимо использовать диспетчер памяти. Реализация относительно проста: диспетчер памяти отслеживает все записи в “разделяемый” сегмент, а затем системы тем или иным способом обеспечивают синхронизацию содержимого измененных сегментов. Беда только в том, что этот метод требует относительно больших накладных расходов.

Теоретики не очень любят разделение памяти, потому что оно влечет за собой все проблемы, перечисленные в предыдущем разделе: необходимость синхронизации, критические секции и т.д. На практике, однако, оно часто оказывается необходимым, так как обеспечивает наибольшую скорость обмена данными – до сотни мегабайт в секунду для современных компьютеров. Тем не менее, используя разделяемую память, мы должны также использовать для синхронизации какие-либо другие средства. В OS/2, Windows NT и системах типа VxVorks чаще всего используются семафоры.

5.3 Средства для гармонического межпроцессного взаимодействия

Ряд систем предоставляет более простые в использовании средства, предоставляющие обмен данных одновременно с синхронизацией. Одним из наиболее типичных средств такого рода является **труба** (**pipe**) или **программный канал** – основное средство взаимодействия между процессами в ОС семейства Unix.

5.3.1 Трубы (програмные каналы)

Труба представляет собой поток байтов. С одного конца в этот поток можно записывать данные, из другого – считывать. Процесс, который пытается считать данные из пустой трубы, будет задержан, пока там не появятся данные. Напротив, пишущий процесс может записать в трубу некоторое количество данных, прежде чем труба заполнится и дальнейшая запись будет заблокирована. На практике труба реализована в виде небольшого (несколько килобайт) кольцевого буфера. Пишущий процесс заполняет этот буфер, пока там есть место. Читающий процесс считывает данные, пока буфер не опустеет.

Читающий процесс может проверить наличие данных в трубе и предпринять какие-то действия в случае их отсутствия. Пишущий процесс также может проверять трубу на переполнение.

По-видимому, трубы являются одной из первых техник, реализующих гармонически взаимодействующие процессы по Дийкстре.

Самым интересным свойством трубы является то, что чтение данных из трубы и запись в нее осуществляется теми же самыми системными вызовами `read` и `write`, что и работа с обычным файлом или внешним устройством типа терминала. На этом основана техника переназначения ввода-вывода, широко используемая в командных интерпретаторах UNIX. Эта техника состоит в том, что большинство системных утилит получают данные из потока стандартного ввода (`stdin`) и выдают их в поток стандартного вывода (`stdout`). При этом, подсовывая в качестве этих потоков терминальное устройство, файл или трубу, мы можем использовать в качестве ввода, соответственно: текст, набираемый с клавиатуры, содержимое файла или стандартный вывод другой программы. Аналогично мы можем выдавать данные сразу на экран, в файл или передавать их на вход другой программы.

Так, например, компилятор GNU C состоит из трех основных проходов: препроцессора, собственно компилятора, генерирующего текст на ассемблере, и ассемблера. При этом внутри компилятора, на самом деле, также выполняется несколько проходов по тексту (в описании перечислено **восемнадцать**), в основном для оптимизации, но нас это в данный момент не интересует, поскольку все они выполняются внутри одной задачи. При этом все три задачи объединяются трубами в единую линию обработки входного текста – **конвейер** (**pipeline**). При этом любая из программ может также брать входные данные из файла.

Трубы широко используются системами семейства Unix и они внесены в стандарт POSIX. Ряд операционных систем, не входящих в семейство Unix, например VxWorks, также предоставляют этот сервис.

Система VMS предоставляет средства, отчасти аналогичные трубам, называемые *почтовые ящики (mailbox)*. Почтовый ящик также представляет собой кольцевой буфер, доступ к которому осуществляется теми же системными вызовами, что и работа с внешним устройством. Системная библиотека языка VAX C использует почтовые ящики для реализации труб, в основном совместимые с UNIX и стандартом POSIX.

В системе UNIX труба создается системным вызовом `pipe(int fildes[2])`. Этот вызов создает трубу и помещает дескрипторы файлов, соответствующие входному и выходному концам трубы, в массив `fildes`. Затем мы можем выполнить `fork`, в различных процессах переназначить соответствующие концы трубы на место `stdin` и `stdout` и запустить требуемые программы. При этом мы получим типичный конвейер – две задачи, стандартный ввод и вывод которых соединены трубой.

Можно понять, что такие трубы можно использовать только для связи родственных задач, т.е. таких, которые связаны отношением родитель-потомок или являются потомками одного процесса.

Для связи между неродственными задачами используется другое средство – **именованные трубы** (**named pipes**) в System V и **UNIX domain sockets** в BSD UNIX. В разных системах именованные трубы создаются различными системными вызовами, но очень похожи по свойствам. Поэтому стандарт POSIX предлагает для создания именованных труб библиотечную функцию `mkfifo(const char * name, mode_t flags);`. Эта функция создает специальный файл. Открывая такой файл, программа получает доступ к одному из концов трубы. Когда две программы откроют именованную трубу, они смогут использовать ее для обмена данными точно так же, как и обычную.

5.3.2 Линки

В транспьютерах микропрограммно реализован **линки** (**links** – связь): механизм, отчасти похожий на трубы.

Линки бывают двух типов – физические и логические. Операции над линками обоих типов осуществляются одними и теми же командами.

Физический линк представляет собой последовательный интерфейс RS432, реализованный на кристалле процессора. С линком также ассоциировано одно слово памяти, смысл которого будет объяснен ниже.

Современные транспьютеры имеют четыре физических линка. Физические линки могут передавать данные со скоростью до 20 Mbaud и могут использоваться как для соединения транспьютеров между собой, так и для подключения внешних устройств. Поэтому физический линк может использоваться как для связи между процессами на разных транспьютерах, так и для синхронизации процесса с внешними событиями, и даже просто для ввода/вывода.

Логический линк – это просто структура данных, выделенная в физическом адресном пространстве процессора. С точки зрения программы физический и логический линки ничем не отличаются, кроме того, что описатель физического линка привязан к определенному физическому адресу. Логический линк может использоваться только для связи между процессами, исполняющимися на одном транспьютере.

Транспьютер `T9000` предоставляет также виртуальные линки – протокол, позволяющий двум транспьютерам организовать несколько линий взаимодействия через один физический линк, или даже через цепочку маршрутизаторов.

Как уже говорилось, процессы взаимодействуют через физические, логические и виртуальные линки совершенно одинаково. Поэтому мы можем отладить параллельную программу в однопроцессорной системе, а потом запустить ее в гиперкубе из 64K транспьютеров – и она будет вести себя точно также, только начнет работать в 64K раз быстрее.

При передаче данных в линк, процесс должен исполнить команду `out`. Эта команда имеет три операнда: адрес линка, адрес массива данных и количество данных. Для передачи операндов используется регистровый стек процессора, но это несущественно. Процесс, исполнивший такую команду, задерживается до тех пор, пока все данные не будут переданы.

Аналогично, при приеме данных из линка, процесс должен исполнить команду `in`. Эта команда также имеет три операнда – адрес линка, адрес буфера, куда необходимо поместить данные и размер буфера. При исполнении такой команды процесс блокируется до тех пор, пока буфер не будет заполнен данными. При этом приемник и передатчик могут использовать буфера разного размера, т.е. приемник может считывать большой массив данных в несколько приемов и т.д.

Существует также команда `alt`, позволяющая процессу ожидать данных из нескольких линков одновременно. В качестве одного из ожидаемых событий можно также использовать сигнал от системного таймера.

Слово, связанное с линком, содержит указатель на дескриптор процесса, ожидающего на линке. Кроме того, это слово может принимать значение `NotProcessP`, указывающее, что линка никто не ждет. Остальная информация, такая, как указатель на буфер и размер буфера, хранится в дескрипторе процесса.

Направление передачи данных определяется командой, которую исполнит очередной процесс при обращении к этому линку. Например, если исполняется команда `out`, предназначенные к записи данные копируются в буфер ожидающего процесса. При этом указатель буфера продвигается, а счетчик размера уменьшается на количество скопированных данных. Если же в линке записано значение `NotProcessP`, процесс переводится в состояние ожидания и указатель на его дескриптор помещается в линк.

Аналогично обрабатываются запросы на чтение. Если мы имеем более двух процессов, пытающихся использовать один линк, то возникает серьезная проблема: внимательный читатель должен был заметить, что мы не сказали, где хранится информация о том, чего ожидает текущий процесс: чтения или записи. Проблема состоит в том, что эта информация *нигде* не хранится. Если процесс попытается записать данные в линк, на котором кто-то уже ожидает записи, то данные второго процесса будут записаны поверх данных ожидавшего. Если размеры буферов совпадут, то ожидавший процесс будет пребывать в убеждении, что он успешно передал все данные. Поэтому линки рекомендуется использовать только для однонаправленной передачи данных между двумя (не более!) процессами.

При работе с физическим линком данные не копируются, а передаются или принимаются через физический канал в режиме прямого доступа к памяти.

В отличие от труб, линк не содержит внутреннего буфера. Это означает, что передающий процесс по окончании передачи может быть уверен, что данные получены (если не произошло описанного выше конфликта). Поэтому линки можно использовать как средство синхронизации двух процессов. Трубы для этого непригодны именно из-за наличия буфера. С другой стороны, по линку нельзя передавать данные по принципу “выстрелил и забыл”, как это часто делают с трубами.

Как уже говорилось, довольно хорошую имитацию трубы можно реализовать при помощи двух линков и буферного процесса.

Интересно, что в версиях ОС `Unix`*, основанных на микроядре, трубы реализованы именно таким образом: микроядро предоставляет низкоуровневые операции* `send` *и* `receive`*, функционально очень похожие на команды* `in` *и* `out` *транспьютера. Системные вызовы* `read` *и* `write` *представляют собой относительно высокоуровневый интерфейс к командам микроядра. Эти вызовы обмениваются данными через буферный процесс, создаваемый в ядре системы при инициализации трубы.*

Напротив, реализовать имитацию линка при помощи труб невозможно из-за наличия буфера и соответствующей невозможности использовать трубы для синхронизации. В этом смысле линк является средством более низкого уровня, чем труба.

5.4 Системы, управляемые событиями

В начале 70-х годов появилась новая архитектура многозадачных систем, довольно резко отличающаяся от вышеописанной модели последовательных процессов. Речь идет о так называемых **системах, управляемых событиями** (**event-driven systems**).

Впервые эта архитектура была реализована в экспериментальных настольных компьютерах Alto, разработанных в 1973 году в исследовательском центре PARC фирмы Xerox. Целью эксперимента было создание операционной среды, удобной для создания интерактивных программ.

В этих системах впервые была реализована многооконная графика, когда пользователь одновременно видит на экране графический вывод нескольких программ и может активизировать любую из них, указав на соответствующее окно при помощи манипулятора-“мыши”.

При каждом движении мыши, нажатии на ее кнопки или клавиши на клавиатуре генерируется **событие**. События могут также генерироваться системным таймером или пользовательскими программами. Существуют также так называемые “визуальные” события: например, сдвинув или закрыв одно из окон, мы открыли часть окна, находившегося внизу. Этому окну посылается событие, говорящее о том, что ему нужно перерисовать часть себя.

Каждое событие представляет собой структуру данных, которая содержит код, обозначающий тип события: движение мыши, нажатие кнопки и т.д., а также поля, различные для различных типов событий: для “мышиных” событий это текущие координаты мыши и битовая маска, обозначающая состояние кнопок (нажата/отпущена). Для клавиатурных событий это код нажатой клавиши – обычно, ASCII для алфавитно-цифровой клавиатуры и специальные коды для стрелок и других “расширенных” клавиш – и битовая маска, обозначающая состояние различных модификаторов, таких как `SHIFT, CNTRL, ALT` и т.д.; для визуальных событий это координаты прямоугольника, который нужно перерисовать, и так далее.

Все события помещаются в очередь в порядке их возникновения.

В системе существует понятие **обработчика событий**. Обработчик событий представляет собой объект, то есть структуру данных, с которой связано несколько программных модулей – **методов**. Один из методов вызывается при поступлении события и называется **callback** (**дословно – “вызов назад”**).

Представим себе простой объект, такой как меню. При нажатии на кнопку мыши в области этого меню вызывается callback. Он разбирается, какой из пунктов меню был выбран, и вызывает соответствующую функцию обработки этого пункта. Таким образом, вместо последовательно исполняющейся программы, время от времени вызывающей систему для исполнения той или иной функции, мы получаем набор callback’ов, вызываемых системой в соответствии с желаниями пользователя. Отсюда, видимо, и происходит термин “callback” – это “системный вызов, идущий в обратном направлении”.

Специальная программа, **менеджер событий**, непрерывно просматривает очередь и передает поступающие события обработчикам. События, связанные с экранными координатами, передаются обработчику, ассоциированному с соответствующим окном. Клавиатурные события передаются **фокусу клавиатуры** – текущему активному обработчику.

Легко понять, что менеджер и обработчики событий достаточно органично вписываются в традиционную многозадачную ОС. События становятся дополнительным средством синхронизации и передачи данных, удобным для организации пользовательского интерфейса. В других ситуациях программист может пользоваться другими средствами. Так устроены Windows NT и оконная подсистема OS/2 – Presentation Manager. Особенно любопытен в этом отношении опыт сетевой оконной системы X Windows, где передача событий реализована на основе стандартных средств межпроцессного взаимодействия: труб или разделяемой памяти при работе в пределах одной машины и **сокетов**[1] при взаимодействии с удаленной машиной.

Однако наиболее распространенные в настоящее время событийные системы: MS Windows 3.x и MacOS реализуют иной подход.

Эти системы, занимающие промежуточное положение между ОС и ДОС, реализуют кооперативную многозадачность. В большинстве же случаев даже кооперативная многопроцессность не используется: менеджер событий синхронно вызывает callback'и, и все программы работают как единый процесс. Предполагается, что обычно callback исполняется быстро, а возможность переключать фокус между различными задачами дает пользователю многие из преимуществ многопроцессной системы. Если долго исполняющиеся callback'и регулярно вызывают функцию `GetNextEvent`, то можно создать довольно приличную иллюзию полноценной ОС.

Одним из основных недостатков такой архитектуры является то, что система оказывается зависимой от качества реализации приложений: чем реже приложение вызывает `GetNextEvent`, тем менее работа системы похожа на многопроцессную. Другие недостатки кооперативной многопроцессности обсуждались в разделе 6.1. Кроме того, кооперативное переключение процессов существенно усложняет реализацию сетевых приложений и **multimedia**: чувствительные к времени реакции программы не могут исполняться как обычные задачи, они вынуждены отбирать управление у системы.

В Windows 3.x для этой цели предоставляется возможность регистрировать так называемые VxD – модули, исполняющиеся с привилегиями ядра системы. Такие модули могут непосредственно обрабатывать внешние события и реагировать на них, не отдавая управления ядру. В Windows 3.x и Windows 95 VxD являются единственным средством, пригодным для реализации приложений, нуждающихся в жесткой синхронизации, например, для синхронизации звука и изображения в multimedia.

Это техническое решение очень характерно для фирмы MicroSoft: вместо полноценной реализации той или иной функции предоставлять разработчикам приложений недокументированные "задние двери". Занятно, что большинство таких задних дверей так или иначе относятся к многопроцессности, синхронизации и межпроцессному взаимодействию.

[1] Средство сетевого обмена данными, во многом похожее на трубы.

Так, вместо защиты ядра MS DOS от повторных вызовов и нормального сервиса для обработки прерываний, MicroSoft реализовал недокументированный флаг занятости ДОС.

Вместо вытесняющей многопроцессности в Windows 3.x предоставлен системный вызов `getNextEvent`

Вместо полноценных средств синхронизации и быстрой реакции на события в Windows 3.x предоставлены VxD, то есть возможность искажения работы ядра системы.

Самым интересным во всем этом является тот печальный факт, что программы, использующие VxD, не могут исполняться в ОС Windows NT, которую MicroSoft рекламирует в качестве "системы будущего"...

Глава 6

Реализация многопроцессности на традиционных (однопроцессорных) компьютерах

6.1 Кооперативная многопроцессность

По-видимому, самой простой реализацией многопроцессной системы была бы библиотека подпрограмм, которая определяет следующие объекты:

`struct Process;` В тексте будет обсуждаться, что должна представлять собой эта структура.

`Process * processCreate(void (*processBody)(void));` Создать процесс, исполняющий функцию processBody.

`void processSwitch();` Вызывается изнутри процесса. Эта функция приостанавливает текущий процесс и активизирует очередной процесс, готовый к исполнению.

`void processExit();` Вызывается изнутри процесса. Эта функция прекращает исполнение текущего процесса.

Сейчас мы не обсуждаем методов синхронизации процессов и взаимодействия между ними. Нас интересует только вопрос: что же мы должны сделать, чтобы переключить процессы?

Функция `processSwitch` называется **диспетчером процессов** или **планировщиком** (**scheduler**) и ведет себя следующим образом:

- Она передает управление на следующий активный процесс.
- Текущий процесс остается активным, и через некоторое время снова получит управление.

- При этом он получит управление так, как будто `processSwitch` представляла собой обычную функцию и возвратила управление в точку, из которой она была вызвана.

Очевидно, что функцию `processSwitch` *нельзя честным образом реализовать на языке высокого уровня, вроде* C, *потому что это должна быть функция, которая* **не возвращает [немедленно] управления в ту точку, из которой она была вызвана**. *Она вызывается из одного процесса, а передает управление в другой.*

Самым простым вариантом, казалось бы, будет простая передача управления на новый процесс, например командой безусловной передачи управления по указателю. При этом весь описатель процесса (`struct Process`) будет состоять только из адреса, на который надо передать управление. Беда только в том, что этот вариант не будет работать.

Действительно, каждый из процессов, вообще говоря, исполняет программу, состоящую из вложенных вызовов процедур. Для того, чтобы процесс нормально продолжил исполнение, нам нужно не только восстановить адрес текущей команды, но и структуру данных, которая содержит информацию обо всех этих вложенных вызовах: **стек вызовов**.

Поэтому мы приходим к следующему варианту:

- Каждый процесс имеет свой собственный стек вызовов.
- При создании процесса выделяется область памяти под его стек, и указатель на эту область помещается в дескриптор процесса.
- Регистр-указатель стека (Stack Pointer – `SP`) процесса настраивается на эту область.
- `processSwitch` сохраняет указатель стека текущего процесса в его дескрипторе и восстанавливает `SP` из дескриптора следующего активного процесса. При этом, когда функция `processSwitch` выполняет оператор `return`, она автоматически возвращает управление в то место, из которого она была вызвана *в этом процессе*, потому что адрес возврата сохраняется в стеке.

Если система программирования предполагает, что при вызове функции должны сохраняться определенные регистры[1], то они также сохраняются в стеке. Поэтому предложенный нами вариант также будет автоматически сохранять и восстанавливать все необходимые регистры.

Понятно, что кроме стека процесса, `struct Process` должна содержать еще некоторые поля. Как минимум, она должна содержать указатель на следующий активный процесс. Система должна хранить указатели на текущий процесс и на конец списка. При этом `processSwitch` переставляет текущий процесс в конец списка, а текущим делает следующий за ним в списке. Все вновь активизируемые процессы также ставятся в конец списка. При этом список не обязан быть двунаправленным, ведь

---

[1] Как, например, C-компиляторы для Intel 80x86 сохраняют при вызовах регистры `SI` и `DI` (`ESI/EDI` в i386)

мы извлекаем элементы только из начала, а добавляем только в конец. Подобным образом работает микропрограммная реализация планировщика в транспьютерах.

Часто в литературе такой список называют **очередью процессов** (**process queue**). Такая очередь присутствует во всех известных авторам реализациях многопроцессных систем. Кроме того, очереди процессов используются и при организации очередей ожидания различных событий, например, при реализации семафоров Дийкстры.

Планировщик, основанный на `processSwitch`, то есть на принципе переключения процессов по инициативе активного процесса, реализован в ряде экспериментальных и учебных систем. Этот же принцип, называемый **кооперативной многопроцессностью**, реализован в библиотеках языков Simula 67 и Modula-2. MS Windows 3.x также имеют средство для организации кооперативного переключения задач – системный вызов `taskIdle` или, как это называется в MS Windows, `GetNextEvent`.

Часто кооперативные процессы называют не процессами, а сопрограммами – ведь они не переключаются принудительно, а вызывают друг друга. Единственное отличие такого вызова от вызова подпрограммы состоит в том, что такой вызов не иерархичен – вызванная программа может вновь передать управление исходной и остаться при этом активной.

Основным (и едва ли не единственным) преимуществом кооперативной многопроцессности является простота отладки планировщика. Кроме того, снимаются все коллизии, связанные с критическими секциями и тому подобными трудностями – ведь процесс может просто не отдавать никому управления, пока не будет готов к этому.

С другой стороны, кооперативная многопроцессность имеет и серьезные недостатки.

Во-первых, необходимость включать в программу вызовы `processSwitch` усложняет программирование вообще и перенос программ из однопроцессных или иначе организованных многопроцессных систем в частности.

Особенно неприятно требование регулярно вызывать `processSwitch` для вычислительных программ. Чаще всего такие программы исполняют относительно короткий внутренний цикл, скорость работы которого определяет скорость всей программы. Для "плавной" многозадачности необходимо вызывать `processSwitch` изнутри этого цикла. Делать вызов на каждом цикле нецелесообразно, поэтому необходимо будет написать что-то вроде:

Пример 9

```
int counter; // Переменная-счетчик.

  while{condition) {

    // Вызывать processSwitch каждые rate циклов.
    counter++;
    if (counter % rate == 0) processSwitch();

    .... // Собственно вычисления
  }
```

Условный оператор и вызов функции во внутреннем цикле сильно усложняют работу оптимизирующим компиляторам и приводят к разрывам конвеера команд, что может привести к очень заметному снижению производительности. Вызов функции на каждом цикле приводит к еще большим накладным расходам и, соответственно, к еще большему замедлению.

Во-вторых, злонамеренный процесс может захватить управление и никому не отдавать его. Просто не вызывать `processSwitch`, и все. Это может произойти не только из-за злых намерений, но и просто по ошибке. Поэтому такая схема оказывается непригодна для многопользовательских систем и часто не очень удобна для интерактивных однопользовательских.

Почему-то большинство коммерческих программ для MS Windows, в том числе и продаваемые самой фирмой MicroSoft, недостаточно используют вызов `GetNextEvent`. Вместо этого такие программы монопольно захватывают процессор и рисуют известные всем пользователям этой системы “песочные часы”. В это время система никак не реагирует на запросы и другие действия пользователя, кроме нажатия кнопок `RESET` или `CTRL-ALT-DEL`.

В-третьих, такая схема непригодна для систем реального времени. Действительно, система реального времени обязана давать *гарантированное* время реакции на внешнее событие. Кооперативная система не обладает таким свойством, потому что никак не может воздействовать на время, проходящее между вызовами `processSwitch` из пользовательской программы.

Простой анализ показывает, что кооперативные многопроцессные системы пригодны только для учебных проектов или ситуаций, когда программисту на скорую руку необходимо склепать многопроцессное ядро. Вторая ситуация кажется несколько странной – зачем для серьезной работы может потребоваться быстро склепанное ядро, если существует много готовых систем реального времени, а также общедоступных (freeware или public domain) в виде исходных текстов реализаций таких ядер?

6.2 Вытесняющая многопроцессность

Все вышесказанное подводит нас к идее вызывать `processSwitch` не из пользовательской программы, а каким-то иным способом. Например, повесить вызов такой функции на прерывание от системного таймера.

Тогда мы получим следующую схему:

- Каждому процессу выделяется квант времени.
- Если процесс не освободил процессор в течение этого кванта, то его снимают и переставляют в конец очереди. При этом все готовые к исполнению процессы более или менее равномерно получают управление.

Этот механизм, называемый **time slicing** или **разделение времени**, реализован в микрокоде транспьютера и практически во всех современных ОС, включая и OS/2. Общим названием для всех методов переключения задач по инициативе системы является термин **вытесняющая** (**preemptive**) многопроцессность. Таким образом, вытесняющая многопроцессность противопоставляется кооперативной, где переключение происходит только по инициативе самой задачи. Разделение времени является частным случаем вытесняющей многопроцессности, но используется чаще всего.

При этом вопрос выбора кванта времени является нетривиальной проблемой. С одной стороны, чрезмерно короткий квант приведет к тому, что большую часть времени система будет заниматься переключением процессов. С другой стороны, в интерактивных системах или системах реального времени слишком большой квант приведет к недопустимо большому времени реакции.

В системе реального времени мы можем объявить процессы, которым надо быстро реагировать, высокоприоритетными и на этом успокоиться. Однако мы не можем так поступить с интерактивными программами в многопользовательской или потенциально многопользовательской, как UNIX на настольной машине типа AT/386 или Sun системе.

Из психологии восприятия известно, что человек начинает ощущать задержку ответа при величине этой задержки около 100 мс. Поэтому в системах разделенного времени, рассчитанных на интерактивную работу, квант обычно выбирают равным десяткам миллисекунд. В старых системах, ориентированных на пакетную обработку вычислительных задач, таких как ОС ДИСПАК на БЭСМ-6, квант может достигать десятых долей секунды или даже секунд. Это повышает эффективность системы, но делает невозможной – или, по крайней мере, неудобной – интерактивную работу.

Вытесняющая многопроцессность имеет много преимуществ, но если мы просто будем вызывать описанный в предыдущем разделе `processSwitch` по прерываниям от таймера или другого внешнего , то такое переключение будет непоправимо нарушать работу прерываемых процессов.

Действительно, пользовательская программа может быть не готова к переключению. Например, она может в данный момент использовать какой-то из регистров, который не сохраняется при вызовах.

Поэтому, например, обработчики аппаратных прерываний сохраняют в стеке *все* используемые ими регистры. Кстати... Если наша функция `processSwitch` будет сохранять в стеке все регистры, то будет происходить именно то, чего мы хотим. `processSwitch` вызывается по прерыванию, сохраняет регистры текущего процесса в текущем стеке, переключается на стек нового процесса, восстанавливает из его стека его регистры, и новый процесс получает управление так, будто и не терял его.

Полный набор регистров, которые нужно сохранить, чтобы процесс не заметил переключения, называется **контекстом процесса**. Такими регистрами, как минимум, являются все регистры общего назначения, указатель стека, счетчик команд и слово состояния процессора. Если система использует виртуальную память, то в контекст процесса входят также регистры диспетчера памяти, управляющие трансляцией виртуального адреса.

Чаще всего оказывается неудобным сохранять контекст именно в стеке. Тогда его сохраняют в какой-то другой области памяти, чаще всего в дескрипторе процесса. Многие процессоры имеют специальные команды сохранения и загрузки контекста процесса.

Для реализации вытеснения достаточно сохранить контекст текущего процесса и загрузить контекст следующего активного процесса в очереди. Необходимо предоставить также и функцию переключения процессов по инициативе текущего процесса, аналогичную `processSwitch`. Это нужно для реализации межпроцессной синхронизации. Чаще всего, однако, вместо `processSwitch` система предоставляет высокоуровневые примитивы синхронизации, например семафоры или функции `send/receive` микроядра[2]. Вызов `processSwitch` оказывается скрыт внутри таких высокоуровневых функций.

Легко понять, что вытесняющий планировщик с разделением времени ненамного сложнее кооперативного планировщика – и тот, и другой реализуются несколькими десятками строк на ассемблере. В работе [11] приводится полный ассемблерный текст приоритетного планировщика системы VAX/VMS *занимающий одну страницу*[3].

Понятно, что у современных процессоров, имеющих десятки регистров общего назначения и виртуальную память, контекст процесса будет измеряться сотнями байт. Например, у процессора VAX контекст процессора состоит из 64 32-разрядных слов, т.е. 256 байт. При этом VAX имеет только 16 регистров общего назначения, а большая часть остальных регистров так или иначе относится к системе управления виртуальной памятью.

На этом основано решающее преимущество транспьютера перед процессорами традиционных и RISC архитектур. Дело в том, что транспьютер не имеет диспетчера памяти, и у него вообще очень мало регистров. В худшем случае, при переключении процессов должно сохраняться 7 32-разрядных регистров. В лучшем случае сохраняются только два регистра – счетчик команд и статусный регистр. Кроме того, перенастраивается регистр `wptr`, который выполняет по совместительству функции указателя стека, базового регистра сегмента статических данных процесса и указателя на дескриптор процесса.

[2] Концепция микроядра подробнее обсуждается в разделе 6.4.

[3] Авторам неизвестно, как публикация этого текста соотносится с авторскими правами фирмы DEC.

Транспьютер имеет три арифметических регистра, организованных в регистровый стек. При этом обычное переключение процессов может происходить только, когда этот стек пуст. Такая ситуация возникает довольно часто; например, этот стек обязан быть пустым при вызовах процедур и даже при условных и безусловных переходах, поэтому циклическая программа не может не иметь точек, в которых она может быть прервана. Упомянутые в предыдущем разделе команды обращения к линкам также исполняются при пустом регистровом стеке. Поэтому, действительно, оказывается достаточно перезагрузить три управляющих регистра, и мы передадим управление следующему активному процессу.

Операция переключения процессов, а также установка процессов в очередь при их активизации полностью реализованы на микропрограммном уровне. Деактивация процесса происходит только по его инициативе, когда он начинает ожидать сигнала от таймера или готовности линка. При этом процесс исполняет специальную команду, которая вызывает микропрограмму, которая, в свою очередь, устанавливает процесс в очередь ожидающих соответствующего события, и переключается на очередной активный процесс. Когда приходит сигнал таймера или данные по линку, то также вызывается микропрограмма, которая устанавливает активизированный процесс в конец очереди активных.

У транспьютера также существует микропрограммно реализованный режим разделения времени, когда по сигналам внутреннего таймера активные процессы циклически переставляются внутри очереди. Такие переключения, как уже говорилось, могут происходить только, когда регистровый стек текущего процесса пуст, но такие ситуации возникают довольно часто.

Кроме обычных процессов в системе существуют также так называемые высокоприоритетные процессы. Если такой процесс получает управление в результате внешнего события, то текущий низкоприоритетный процесс будет прерван независимо от того, пуст его регистровый стек или нет. Чтобы при этом не разрушить прерванный процесс, его стек и весь остальной контекст спасаются в быструю память, расположенную на кристалле процессора. Это и есть тот самый худший случай, о котором говорилось выше. Весь цикл переключения занимает 640нс по сравнению с десятками и, порой, сотнями микросекунд у традиционных процессоров.

Все это вместе представляет собой довольно нетривиальную и красивую конструкцию, о которой подробнее нужно читать в литературе, посвященной этому предмету ([8], [9]).

Благодаря такой организации транспьютер не имеет равных себе по времени реакции на внешнее событие. На первый взгляд, микропрограммная реализация такой довольно сложной конструкции, как планировщик, снижает гибкость системы. В действительности, в современных системах планировщики реализуются довольно стандартным образом, и реализация, выполненная в транспьютере, очень близка к этому стандарту, известному как **микроядро** (**microkernel**). Кроме того, аппаратная схема взаимодействия между процессами по линкам является достаточно мощной, и на ее основе можно смоделировать практически любой другой механизм, например те же семафоры Дийкстры. А все возможные накладные расходы будут скомпенсированы очень высокой производительностью базовой системы.

6.3 Планировщики с приоритетами

В многопроцессных системах часто возникают вопрос: в каком порядке исполнять готовые процессы? Часто бывает очевидно, что одни из процессов важнее других. Например, в системе может существовать три готовых к исполнению процесса: процесс - сетевой файловый сервер, интерактивный процесс - текстовый редактор и процесс, занимающийся плановым резервным копированием с диска на ленту. Очевидно, что хотелось бы в первую очередь разобраться с сетевым запросом, затем – отреагировать на нажатие клавиши в текстовом редакторе, а резервное копирование может подождать сотню-другую миллисекунд. С другой стороны, мы должны гарантировать пользователя от ситуаций, когда какой-то процесс вообще не получает управления, потому что система постоянно занята более приоритетными заданиями. Действительно, вы запустили то же самое резервное копирование, и сели играть в тетрис или писать письмо любимой женщине, а после получаса игры обнаружили, что ни одного байта не скопировано – процессор все время был занят.

Самым простым и наиболее распространенным способом распределения процессов по приоритетам является организация нескольких очередей – в соответствии с приоритетами. При этом процесс из низкоприоритетной очереди получает управление тогда и только тогда, когда все очереди с более высоким приоритетом пусты.

Простейшим случаем такой организации является транспьютер, имеющий две очереди. При этом система не может отобрать управление у высокоприоритетного процесса. В этом смысле низкоприоритетные задачи вынуждены полагаться на “порядочность” высокоприоритетных, т.е. на то, что те освобождают процессор в разумное время.

Отчасти похожим образом организован планировщик системы VAX/VMS. Он имеет 32 приоритетных очереди, из которых старшие 16 называются процессами реального времени, а младшие – разделенного. При этом процесс реального времени исполняется всегда, когда готов к исполнению, и в системе нет более приоритетных процессов. В этом смысле ОС и процессы разделенного времени также вынуждены полагаться на его порядочность. Поэтому привилегия запускать такие процессы контролируется администратором системы.

В свою очередь, процессы разделенного времени периодически переключаются между собой.

Легко понять, что разделение времени обеспечивает более или менее справедливый доступ к процессору для задач с одинаковым приоритетом. В случае транспьютера, который имеет только один приоритет и, соответственно, одну очередь для задач разделенного времени, этого оказывается достаточно. Однако современные ОС, как общего назначения, так и реального времени, имеют много уровней приоритета. Для чего это нужно, и как достигается в этом случае честное распределение времени процессора?

Дело в том, что в системах такого типа приоритет процессов разделенного времени является динамической величиной. Он изменяется в зависимости от того, насколько активно задача использует процессор и другие системные ресурсы.

В системах с пакетной обработкой, когда для задачи указывают верхнюю границу времени процессора, которое она может использовать, часто более короткие задания идут с более высоким приоритетом. Кроме того, более высокий приоритет дают задачам, которые требуют меньше памяти.

В системах разделенного времени часто оказывается сложно заранее определить время, в течение которого будет работать задача. Например, вы отлаживаете программу. Для этой цели вы запускаете символьный отладчик и начинаете исполнять вашу программу в пошаговом режиме. Естественно, что вы не можете даже приблизительно предсказать как астрономическое время отладки, так и время центрального процессора, занятое при этом. Поэтому обычно такие системы не ограничивают время исполнения задачи и другие ресурсы и вынуждены делать предположения о поведении программы в будущем на основании ее поведения в прошлом.

Так, если программа начала вычисления, не прерываемые никакими обращениями к внешней памяти или терминалу, мы можем предположить, что она будет заниматься такими вычислениями и дальше. Напротив, если программа сделала несколько запросов на ввод/вывод, можно ожидать, что она и дальше будет активно выдавать такие запросы.

Предпочтительными для системы будут те программы, которые захватывают процессор на короткое время и быстро отдают его, переходя в состояние ожидания внешнего или внутреннего события. Поэтому таким процессам система стремится присвоить более высокий приоритет. Если программа ожидает завершения запроса на обращение к диску, то это также выгодно для системы – ведь на большинстве машин чтение и запись на диск происходит параллельно с работой центрального процессора. IBM PC с контроллером IDE или MFM в этом смысле представляет собой досадное исключение.

Поэтому система динамически повышает приоритет тем заданиям, которые освободили процессор в результате запроса на ввод-вывод или ожидание события и, наоборот, снижает тем заданиям, которые были сняты с процессора по истечению кванта времени. Однако приоритет не может превысить определенного значения – стартового приоритета задачи.

При этом наиболее высокий приоритет автоматически получают интерактивные задачи и программы, занятые интенсивным вводом-выводом. Поэтому процессор часто оказывается свободен, и управление получают низкоприоритетные вычислительные задания. Поэтому системы UNIX и VAX/VMS даже при очень высокой загрузке обеспечивают как приемлемое время реакции для интерактивных программ, так и приемлемое астрономическое время исполнения для пакетных заданий. А благодаря возможности создавать процессы реального времени[4] эти же ОС можно использовать и в задачах реального времени, вроде управления атомным реактором.

VMS повышает приоритет также и тем задачам, которые остановились в ожидании подкачки страницы. Это сделано из тех соображений, что если программа несколько раз выскочила за пределы своего рабочего набора, то она будет делать это и далее, а процессор на время подкачки она освобождает.

Нужно отметить, что процесс разделенного времени может повысить свой приоритет до максимального в классе разделения времени, но никогда не сможет стать процессом реального времени. А для процессов реального времени динамическое изменение приоритетов обычно не делают.

[4]Из ОС семейства Unix процессы реального времени бывают только в System V Release 4 и HURD.

Любопытно реализовано динамическое изменение приоритета в OS-9. Там каждый процесс имеет статически[5] определенный приоритет, и **возраст** (**age**) – количество просмотров очереди с того момента, когда этот процесс в последний раз получал управление. Обе эти характеристики представлены 16-разрядными беззнаковыми числами. При этом управление каждый раз получает процесс с наибольшей суммой статического приоритета и динамически изменяющегося возраста. Если у двух процессов такие суммы равны, то берется процесс с бо̀льшим приоритетом. Если у них равны и приоритеты, то берется тот, который оказался ближе к началу очереди.

Видно, что такая схема гарантирует, что любой низкоприоритетный процесс рано или поздно получит управление. Если же нам нужно, чтобы он получал управление рано, то мы должны просто повысить его приоритет.

Кроме того, можно запретить исполнение процессов со статическим приоритетом ниже заданного. Это может уменьшить загрузку процессора и позволить высокоприоритетным процессам разгрести увеличившийся поток внешних событий. Понятно, что такой запрет можно вводить только на небольшое время, чтобы не нарушить честное распределение процессора.

Возможна и более тонкая регулировка – системный вызов, который запрещает увеличивать возраст процесса больше заданного значения. То есть, процесс, стоя в очереди, может достичь этого максимального возраста, после чего он по-прежнему остается в очереди, но его возраст уже не увеличивается. Получающаяся в результате схема распределения времени процессора отчасти похожа на двухслойную организацию VAX/VMS, когда исполнение процессов со статическим приоритетом, превышающим границу, не может быть прервано низкоприоритетным процессом.

Стандарт POSIX практически не накладывает ограничений на организацию распределения приоритетов между программами. Однако наследие ОС UNIX привело к тому, что в него введена команда `nice` – запустить программу с заданным приоритетом. При этом заданный приоритет может быть только ниже приоритета запускающей задачи.

Такая команда неявно предполагает наличие схемы динамического распределения приоритетов, аналогичной вышеописанной. При этом все известные авторам системы, полностью реализующие стандарт POSIX, (Системы семейства Unix и VMS для VAX и Alpha) в действительности реализуют именно такую схему. Отличие между UNIX'ами и VMS в этом плане состоит только в том, что обычный Unix не имеет понятия процесса реального времени.

6.4 Монолитные системы и системы с микроядром

Кроме планировщика типичная ОС включает в себя много других модулей – подсистему ввода/вывода, файловую систему или системы, диспетчер памяти и ряд менее важных. Практически все эти модули работают с разделяемыми ресурсами и не могут быть реентерабельными.

[5]Вообще-то, процесс может повысить или понизить свой приоритет, исполнив соответствующий системный вызов. Но система по собственной инициативе никогда не меняет его.

Те из читателей, кто занимался разработкой вирусов или TSR для MS/DR DOS, должны быть хорошо знакомы с этой проблемой. Вопреки хакерскому фольклору, нереентерабельность ядра DOS связана вовсе не с переустановкой указателя стека при входе в обработчик прерывания 21h, а именно с тем, что ядро работает с разделяемыми данными.

Одно из первых решений этой проблемы состояло в том, чтобы запретить переключение задач во время работы нереентерабельных модулей ОС. При этом все модули ОС собираются в конгломерат, называемый **ядром** (**kernel**). В системах с виртуальной памятью код и данные ядра обычно расположены в едином виртуальном адресном пространстве, которое отличается от адресных пространств пользовательских задач. Ядро можно назвать привилегированной задачей, но оно не является процессом в полном смысле этого слова, потому что планировщик не может отобрать у ядра процессор. В ядре собраны все нереентерабельные модули системы, а часто и многие реентерабельные. Сам планировщик также является частью ядра.

Исполняя системный вызов, пользовательская программа передает управление ядру. При входе в ядро планирование процессов прекращается. Например, это может быть сделано сбросом системного таймера. Ядро исполняет запрос или ставит его в очередь на исполнение и передает управление планировщику. Планировщик переставляет текущий процесс в конец очереди (если он остался активным), программирует таймер, задавая в качестве интервала квант времени, и передает управление следующему активному процессу.

Процесс может потерять управление двумя способами – по истечении кванта времени или при исполнении системного вызова. В первом случае он всегда остается в очереди активных, во втором может остаться, а может и не остаться, в зависимости от сделанного вызова. Запросы ввода/вывода и синхронизации с другими процессами часто приводят к переводу процесса в состояние ожидания.

Ядро может потерять управление только при аппаратном прерывании. Прерывания обрабатываются специальными программами – драйверами устройств. Драйверы являются частью ядра и не имеют права исполнять обычные системные вызовы. Ни один пользовательский процесс не может получить управление, пока какой-то из модулей ядра активен.

В наше время такая архитектура называется **монолитным ядром**. Она является непосредственным потомком однопроцессных или кооперативно многопроцессных ОС, создававшихся в конце 50-х – начале 60-х, и оформилась еще до того, как теоретики разработали удобные методы синхронизации и взаимодействия процессов.

Основным недостатком монолитного ядра является вовсе не то, что это "старая" архитектура. Такая структура вполне приемлема для ОС общего назначения – большинство систем семейства Unix и OS/2 более или менее успешно её используют.

Проблемы возникают, когда мы пытаемся реализовать систему реального времени. Действительно, процессы РВ должны получать управление в течение гарантированного интервала времени. И этот интервал должен быть, по возможности, небольшим. А время работы некоторых модулей ОС может быть довольно долгим – например, время поиска файла в директории. Поэтому, если мы хотим реализовать ОС реального времени с хорошим временем реакции, мы должны присваивать процессам РВ более высокий приоритет, чем у ядра.

Но как такие процессы будут исполнять системные вызовы? Ведь если процесс имеет более высокий приоритет, чем ядро, то он может получит управление во время работы нереентерабельного модуля системы. Как же быть?

Те, кто хорошо знаком с MS/DR DOS, прочитав предыдущий абзац, сразу же произнесут ответ – **DOS Busy flag** (**флаг занятости ДОС**). В нашей ситуации было бы правильнее назвать его **"Kernel Busy"** (**Ядро занято**). Речь идет о флаговой переменной, которая имеет значение `TRUE`, если ядро в данный момент активно, и `FALSE` все остальное время. Пользовательская программа может получить значение флага специальным реентерабельным системным вызовом (в DOS этот вызов не был документирован вплоть до версии 5.0, хотя впервые он появился в DOS 3.0).

Используя этот механизм, программа РВ перед исполнением любого системного вызова должна проверить "Kernel Busy flag". Если флаг равен `FALSE`, то можно исполнять вызов. Если же он равен `TRUE`...

Мы не будем детально обсуждать вопрос о том, что могла бы сделать программа, если `Kernel Busy flag` окажется равен `TRUE`. Например, она могла бы скопировать сегмент данных ядра в свой внутренний буфер, сбросить флаг, исполнить вызов и скопировать данные ядра обратно. Пользователи "нормальных" ОС пришли бы в ужас от такого предложения, хотя это довольно часто используемая техника реализации `TSR`-программ для `DOS`. Более подробное обсуждение вариантов увело бы нас слишком далеко от основной темы.

В первом приближении эта проблема была решена в системах типа RT-11. В этой системе обращения к нереентерабельным модулям, например к подсистеме ввода/вывода, реализованы не как вызовы, а как установка запроса в очередь. То есть, обработка запроса начинается не сразу же после вызова, а, возможно, через некоторое время. Поэтому мы можем поставить запрос в очередь практически в любой момент, в том числе и во время исполнения обработчика запросов.

Мы не можем ставить запрос во время других операций над этой же очередью – но время извлечения/вставки элемента из/в очередь равно времени исполнения трех-четырех команд, а в VAX *эта операция делается всего одной командой. На это время можно просто запретить прерывания.*

Более или менее похожая техника реализована во многих других ОС реального времени и общего назначения, например в OS-9, RSX-11 и VMS. Развитие этой идеи и согласование ее с концепцией гармонически взаимодействующих процессов, привело в 80-е годы к архитектуре, известной как **микроядро** (**microkernel**).

Микроядро состоит из планировщика и базовых средств передачи данных между процессами и синхронизации. В качестве таких средств могут быть предоставлены всего четыре базовые функции:

`port_t create(options, ...);` Создать порт для передачи/приема данных. В транспьютере такой порт называется **линком**.

`int send(port_t to, void * what, size_t how_much);` Передать в порт блок данных заданного размера.

`int receive(port_t from, void * where, size_t how_much);` Получить из порта блок данных заданного размера. Если данных еще нет, программа-получатель будет ожидать их готовности.

`void delete(port_t port);` Удалить ненужный порт.

Набор параметров у функций может изменяться. В первую очередь это касается параметров функций `send/receive`. В нашем примере порт представляет собой неструктурированнный поток данных, позволяющий обмениваться сообщениями произвольного размера. В некоторых реализациях размер сообщения может быть фиксированным. Возможны и более интересные конструкции, например, аналоги механизма **рандеву** языка Ada.

В некоторых системах, например в транспьютере, функция `receive` может обладать большей гибкостью – она может позволять ожидать данных из нескольких портов, анализировать состояние порта без ожидания данных, ожидать не более заданного времени и т.д.

Микроядро получило свое название не только из-за малого количества функций, но и из-за того, что его реализация очень компактна – около 10 Кбайт кода для процессора 80x86; для CISC-процессоров с рациональной архитектурой типа MC680x0 чуть поменьше, для RISC – чуть побольше. В транспьютере полноценное микроядро реализовано микропрограммно, то есть операция `send` и все варианты операции `receive` являются командами процессора.

В таких ОС все остальные модули системы являются отдельными процессами, и взаимодействуют между собой при помощи операций `send` и `receive`. Если пользовательскому процессу нужно открыть файл, он посылает запрос соответствующему системному процессу и ждет ответа. Если системный процесс в это время был активен – ничего страшного, просто придется подождать немного подольше. Такая архитектура снимает все проблемы с реентерабельностью ядра системы, и позволяет процессам реального времени и даже обработчикам прерываний исполнять системные вызовы без ограничений. Последнее, в частности, означает, что теперь можно писать драйверы, обращающиеся к другим драйверам или к ядру системы.

Сами процедуры `send` *и* `receive`*, разумеется, являются нереентерабельными, но время их исполнения очень мало. На это время можно просто запретить прерывания.*

Строго говоря, различие между монолитными и микроядерными ОС не столь велико, как может показаться. Дело в том, что практически все ОС имеют в составе ядра асинхронные подсистемы, в первую очередь – подсистему ввода/вывода (см. раздел 7.3.3). Даже в однопроцессной MS DOS дисковые кэши типа `SMARTDRV.EXE` осуществляют обмен с диском асинхронно.

Основное отличие состоит в том, что в системе на основе микроядра системные и пользовательские процессы одинаково планируются и используют одни и те же системные вызовы для обмена данными и синхронизации. В монолитных же ОС процессы ядра и пользовательские процессы по-разному планируются и, чаще всего, используют для взаимодействия различные средства.

С концептуальной точки зрения разница небольшая. С практической же серьезные отличия есть. Дело в том, что в монолитных системах для обмена данными между процессами ядра нередко используется разделяемая память. Кроме того, процессы ядра часто имеют урезанный контекст, который содержит не все регистры процессора. Чаще всего урезают регистры диспетчера памяти, тем самым уменьшая виртуальное адресное пространство, доступное таким процессам.

Однако реальная ценность всех этих оптимизаций невелика. Самой быстрой ОС для машин на основе процессоров i80x86 является микроядерная система реального времени QNX, намного превосходящая по субъективной скорости, времени переключения процессов и удельным накладным расходам такие монолитные ОС, как OS/2 и Windows NT.

Микроядро оказывает большую услугу в структурировании модулей ОС. Теперь все модули – диспетчер памяти, файловые системы, драйверы устройств и т.д. – должны взаимодействовать друг с другом в выделенных точках и с использованием хорошо определенного интерфейса, в полном соответствии с концепциями структурного и модульного программирования.

Это упрощает разработку ОС "с нуля", но может резко усложнить переработку существующей системы под микроядерную архитектуру. Это сильно зависит от структурированности кода исходной системы.

Так, Unix System Laboratories успешно переделали монолитное ядро UNIX System V R3 в микроядерную систему System V R4. А фирма IBM, пытаясь преобразовать монолитное ядро OS/2 в микроядерную систему Workplace OS (WPOS) для PowerPC, столкнулась с проблемами, существование которых были вынуждены признать даже официальные представители фирмы.

Кроме того, архитектура микроядра резко упрощает разработку ОС для многопроцессорных машин с общей памятью и для распределенных многомашинных систем, не имеющих общей физической памяти. Все, что нам нужно – это создать на каждом процессоре (каждой машине) копию микроядра и научиться передавать сообщения между ними. В случае многомашинной системы сообщения могут передаваться по локальной или даже глобальной сети. При этом пользовательские программы могут вообще не заметить перехода к многомашинной конфигурации – все сообщения передаются теми же командами `send` и `receive`, изменяется только команда создания порта. Именно так реализовано межмашинное взаимодействие в системе QNX.

Еще одна интересная идея, реализация которой резко упрощается в микроядерных системах – это предоставление различным пользовательским программам различных наборов системных вызовов. Впрочем, такая идея потенциально опасна – например тем, что оказывается сложно организовать взаимодействие между программами, исполняемыми в различных подсистемах.

Например, Windows NT предоставляет три различных набора системных вызовов – Unixоподобный интерфейс POSIX-подсистемы, интерфейс, совместимый с MS Windows 3.x, известный как Win16-подсистема или WOW (Windows On Windows), и оригинальный 32-разрядный системный интерфейс, во многом похожий по структуре на Windows 3.x, но более богатый и изощренный, известный как Win32-подсистема. При этом программы в POSIX-подсистеме не могут запускать программы Win16 и Win32 и практически не могут взаимодействовать с ними.

Первой из известных авторам коммерческих ОС, последовательно реализующих архитектуру микроядра, была система QNX. Эта система имеет микроядро размером около 10 Кбайт, реализующее передачу сообщений между процессами как в пределах одной машины, так и между машинами в локальной сети. В качестве сетевого протокола могут использоваться X.25, SNA, TCP/IP или собственный сетевой протокол.

QNX разрабатывался для приложений реального времени, в том числе и для использования во встраиваемых микропроцессорных системах; но, благодаря компактности и фантастической производительности, эта ОС иногда заменяет системы общего назначения, например, на сильно загруженных серверах телеконференций.

Другим примером микроядерной системы реального времени является транспьютер. Аппаратно реализованное микроядро транспьютера содержит планировщик с двумя уровнями приоритета и средства для передачи сообщений по линкам.

Микроядро использует ряд других современных и будущих ОС, например, UNIX System V R4, Workplace OS фирмы IBM, проект HURD и разрабатываемая новая версия BSD UNIX.

Напротив, система Windows NT фирмы MicroSoft, по признанию ее разработчиков, не является полностью микроядерной. К сожалению, авторы не смогли понять на основе общедоступной документации, какова же ее архитектура и чем она отличается от монолитной ОС, с одной стороны, и микроядра – с другой.

Существует несколько общедоступных (**public domain**) реализаций многопроцессорного микроядра, например, разработанное в Университете Беркли микроядро Mach. Системы WPOS и HURD разрабатываются именно на основе этого ядра.

Глава 7

Внешние устройства

Кроме центрального процессора и оперативной памяти, компьютерные системы обычно оснащаются теми или иными внешними устройствами. Прежде всего среди таких устройств следует упомянуть механизмы взаимодействия с пользователем, а также долговременные запоминающие устройства – диски, ленточные накопители и т.д.

С функциональной точки зрения, внешние устройства, подключаемые к современным компьютерам, можно разделить на следующие категории.

- Устройства внешней памяти, которые в свою очередь, можно разделить на два класса:
  - устройства памяти с произвольным доступом, главным образом магнитные диски. К этому же классу относятся дискеты, магнитооптические и оптические диски. (Не все устройства данного типа являются дисками. Например, в 60-е и 70-е годы существовали магнитные барабаны, в которых каждому логическому сектору соответствовала одна дорожка на барабане и одна головка чтения-записи. Такие устройства обеспечивали гораздо более высокую скорость доступа, чем тогдашние магнитные диски, хотя и были намного более дорогими в расчете на килобайт емкости). Удачным универсальным обозначением для этого класса устройств является принятое в документации фирмы IBM сокращение **DASD** (**Direct Access Storage Device** – запоминающее устройство прямого доступа);
  - устройства памяти с последовательным доступом. В основном, это лентопротяжные устройства (стримеры и др.).
- Устройства последовательного ввода/вывода: печатающие устройства, телетайпы, терминалы и т.д.
- Векторные и растровые графические терминалы.
- Позиционные устройства ввода: мыши, планшеты-дигитайзеры, световые перья и т.д.
- Сетевые адаптеры.

- Устройства звукового ввода/вывода.
- Устройства графического ввода/вывода: сканнеры или видеодекодеры (ввод), плоттеры, графические принтеры или видеокодеры (вывод).
- Специализированная контрольно-измерительная аппаратура.

7.1 Обзор существующих внешних устройств

Исторически одним из первых механизмов взаимодействия с пользователем был телетайп. Телетайп представляет собой электрическую пишущую машинку, которая может печатать символы, либо вводимые с клавиатуры, либо передаваемые ей по проводу. Такие машинки использовались для передачи телеграмм в США еще до второй мировой войны, поэтому в 50-е гг. телетайп был очень распространенным аппаратом.

Телетайп принимает и передает по проводу последовательности двоичных значений – битов. В ранних моделях код одного символа содержал только 6 бит. При помощи 6 бит можно представить 64 различных значения, чего с трудом хватает для кодирования управляющих кодов (перевода каретки, конца передачи и т.д.), английских букв верхнего регистра и цифр. Примером такой кодировки является 6-битная телеграфная кодировка RADIX, позволяющая представить символы верхнего регистра, цифры и знак `'$'`, а также знаки препинания, пробел и перевод строки.

Существование 6-битных символов оказало воздействие на архитектуру некоторых машин третьего поколения. Например, в 24-разрядных машинах линии ICL 1900 есть понятие 6-битного байта. Это понятие унаследовал и распространенный в Институте ядерной физики Одренок.

В более современных телетайпах используется 7- или 8-битная кодировка, позволяющая кодировать не только верхний регистр алфавита, но и нижний, а также различные экзотические символы вроде фигурных скобок. Примером такой кодировки является широко известный код ASCII (American Standard Code for Information Interchange – Американский стандартный код информационного обмена). 8-битная кодировка позволяет представлять не только набор символов ASCII, но и символы национальных алфавитов, например, кириллицы.

Независимо от кодировки обычный алфавитно-цифровой терминал, такой, как vt100 или IBM 3270, с точки зрения компьютера ничем не отличается от телетайпа – это тоже устройство, передающее и получающее по последовательной линии двоичные коды. Как правило, такие терминалы помимо алфавитно-цифровых и прочих печатаемых символов умеют обрабатывать специальные командные последовательности: позиционирование курсора, очистка экрана, прокрутка экрана, изменение яркости символов и т.д. В большинстве систем (Unix, VMS) генерация таких командных последовательностей возлагается на прикладную программу, а ОС выполняет только передачу сформированных команд на терминал.

Для подключения терминальных устройств был разработан протокол передачи двоичных данных по последовательной линии RS232 (**Recommended Standard #232**). Этот протокол определяет электрические параметры устройств, выполняющих немодулированнную передачу данных по витой паре проводов или по двум таким парам. Немодулированная передача означает, что битовые значения передаются строго последовательно и биту 1 соответствует высокое напряжение (обычно 5 В.), а биту 0 – низкое (обычно 0).

Такая передача относительно проста, но отличается низкой защищенностью от помех. Стандарт RS232 определяет один из простых способов защиты от помех – передачу вместе с каждой порцией данных так называемого **бита четности**. Аппаратура передатчика устанавливает этот бит так, чтобы общее число единиц в посылке было четным или нечетным, в зависимости от договоренности. Это позволяет обнаруживать ошибки при передаче, но не позволяет их исправлять. Исправление в таком случае возлагается на протокол более высокого уровня. Например, некоторые терминалы могут запросить повтор ошибочного байта. При передаче по протоколу SLIP (Serial Line Internet Protocol) вместе с пакетом данных передается контрольный код **CRC** (**Cyclic Redundancy Code** – циклический избыточный код), который позволяет исправлять единичные, а иногда и множественные ошибки.

Для передачи по длинным линиям, особенно по телефонным каналам, используются специальные устройства – **модемы** (**модулятор/демодулятор**), преобразующие передаваемые значения в звуковой сигнал различного тона. Модемы, как правило, умеют не только делать модулированную передачу, но и дозваниваться до заданного телефонного номера.

В настоящее время порты RS232 используются для подключения широкого спектра различной низкоскоростной аппаратуры. С их помощью подключаются мыши, модемы, различная измерительная аппаратура, даже принтеры. Практически любая современная вычислительная система имеет несколько последовательных портов. Например, в IBM PC под MS DOS последовательные порты видны как устройства `COM` с соответствующим номером. Система UNIX называет те же порты `/dev/tty`, тоже с номером. Современные RS232-порты обеспечивают передачу на скоростях до 115 кБод (килобит в секунду). Существуют также более скоростные стандарты, сверху вниз совместимые с RS232, например, RS422.

С точки зрения компьютера адаптер последовательного порта представляет собой устройство, подключенное к системной шине. Это устройство имеет один или несколько управляющих портов, а также байтовый порт данных. Порты отображены на адреса памяти или находятся в специальном адресном пространстве ввода/вывода в зависимости от архитектуры процессора. Адаптер может генерировать прерывание при получении очередного байта или при завершении передачи.

Некоторые многопортовые платы, например адаптер DL-11 фирмы DEC, могут сбрасывать получаемые данные в заданную область памяти, не прерывая работу центрального процессора, или, наоборот, передавать в порт заданное количество данных из памяти, начиная с заданного адреса. При этом центральный процессор должен только запрограммировать устройство на прием или передачу. Такой режим работы называется **прямым доступом к памяти** (**ПДП**) и часто используется в устройствах, передающих большие объемы данных, особенно в дисковых контроллерах. Словосочетание **ПДП** – это калька с англоязычного термина **DMA** (**Direct Memory Access**).

Для подключения более скоростных устройств, например принтеров или лентопротяжек, используются параллельные порты. В современных системах обычно используется стандарт de-facto, известный как Centronix. В первом приближении такой порт представляет собой восемь последовательных линий, в электрическом отношении очень похожих на RS232, и провода, по которым передаются синхронизационные сигналы. Понятно, что такая линия будет по крайней мере в восемь раз быстрее, чем последовательный порт.

Часто скорость передачи по параллельному порту оказывается недостаточной, например, при печати изображений на лазерном принтере. Для решения этой проблемы существуют специальные высокоскоростные адаптеры, подключаемые непосредственно к системной шине компьютера.

Вообще, устройства, работающие с высокой скоростью, часто имеют собственный контроллер, подключаемый непосредственно к системной шине. Например, в компьютерах третьего поколения и ниже таким образом часто подключались магнитные диски.

Подключение устройства непосредственно к системной шине имеет два серьезных недостатка.

- Такое устройство может подключаться только к компьютерам с определенным типом шины. Это может не казаться недостатком, если мы имеем дело только с монокультурой одинаковых систем, например IBM PC AT, с шиной ISA. Если же у нас появляется несколько других систем, скажем microVAX, Sun, Macintosh или хотя бы x86 с шиной EISA, или PCI, или вообще ЕС-184X с советским стандартом шины, похожим на ISA, но отличающимся от него – сразу начинаются трудности.
- Для подключения устройства необходимо вскрывать компьютер. В некоторых случаях, например, если машина предназначена для работы в промышленных цехах и имеет герметичный корпус, это превращается в целую эпопею. Вообще, системная шина – дело тонкое, и включать в нее сомнительное устройство, разработанное неизвестной третьей фирмой, часто бывает просто страшно. Поэтому многие конторы, занимающиеся гарантийным ремонтом машин, отказываются чинить машины, в которые ставились неизвестные платы.

Для дисковых контроллеров существует еще одна проблема – дисковый контроллер для машины третьего поколения представляет собой сложную систему, сравнимую по сложности с центральным процессором. С одной стороны, контроллер управляет шаговыми двигателями подачи магнитных головок, формирует модулированный сигнал для записи на магнитную поверхность, сочиняет информацию, записываемую в заголовок и конец каждого блока на диске, и т.д. С другой стороны, он буферизует считываемые и записываемые данные, делает их передачу в основную память по ПДП и осуществляет все взаимодействие с центральным процессором.

Поэтому еще в 60 гг. появились различные стандарты де-факто, позволяющие одному дисковому контроллеру работать с различными дисководами. Так, одни и те же дисковые накопители могли использоваться как с мэйнфрэймами фирмы IBM, так и с суперкомпьютерами Cray или CDC, или с VAX-11/780. А контроллер дисковой подсистемы поставлялся вместе с компьютером.

Такая организация дисковой подсистемы использовалась и в ранних моделях IBM PC. Имеются в виду так называемые MFM-винчестеры, у которых большая часть управляющей электроники расположена на плате, подключаемой к системной шине.

Однако электроника прогрессировала очень быстро, и контроллеры дешевели гораздо быстрее, чем механика. Поэтому в 80-е гг. выяснилось, что стоимость контроллера намного ниже стоимости самого дисковода. Оказалось целесообразным перенести весь контроллер на плату, привинченную к корпусу диска, а к системной шине подключать только относительно простой **HBA** (**Host Bus Adapter** – адаптер шины “хозяина”). Наиболее простым примером такой организации являются диски IDE (Integrated Drive Electronic), используемые в большинстве современных дешевых клонов IBM PC/AT-AT386.

Более развитой архитектурой такого типа обладают устройства, выполненные в стандарте SCSI (**Small Computer System Interface** – интерфейс [для] малых компьютерных систем, читается “скази"). SCSI представляет собой высокоскоростную параллельную шину, к которой может подключаться до семи устройств. По этой шине передаются высокоуровневые команды, например – выдать состояние устройства; перевести считывающую головку к заданному сектору[1]; считать блок данных. Система команд достаточно широка, чтобы к этой шине можно было подключать не только магнитные диски, но и лентопротяжки, оптические дисководы и даже устройства, вообще не являющиеся устройствами внешней памяти, например сканеры, или даже сетевые адаптеры Ethernet.

Устройства SCSI бывают двух типов – устанавливаемые внутри корпуса компьютера и внешние. Внешние устройства имеют свой корпус, свой блок питания и даже свой сетевой шнур, поэтому они заметно дороже внутрикорпусных. Внешние устройства очень полезны, если внутри корпуса мало места, как в случае настольных рабочих станций Sun или x86 - low profile. Соответственно есть два типа кабелей SCSI – для внешних устройств (круглый кабель, заключенный в экран и оболочку из толстой пластмассы) и для внутренних – обычный плоский кабель, внешне похожий на кабели для подключения IDE-дисков и флопповодов, но намного шире.

Большинство современных рабочих станций и мощных персональных компьютеров имеют SCSI-адаптеры.

Говоря о стандартах подключения внешних устройств, нельзя не упомянуть о различных стан-

[1] SCSI использует линейную адресацию секторов, т.е. нулевой сектор второй дорожки идет сразу за последним сектором первой и т.д.

дартных шинах для измерительной аппаратуры, например, о стандарте CAMAC. Этот стандарт по своей структуре похож на системную шину компьютера с низкой производительностью. Более того, существует компьютер с такой системной шиной: известный в ИЯФе Одренок. Однако большинство компьютеров подключается к шине CAMAC через более или менее сложный HBA. Существует и ряд других стандартов **магистрально-модульной** аппаратуры – VME, Multibus и т.д.

Следует четко разделять магистрально-модульные стандарты, такие как CAMAC, VME или SCSI, и стандарты системной шины ISA, EISA, MCA, VLB, PCI для IBM PC и ее клонов, NuBus для Macintosh UNIBUS и Q-bus для техники фирмы DEC: PDP-11 и семейства VAX.

Системная шина представляет собой способ непосредственного подключения устройств к центральному процессору. Поэтому такие интерфейсы отличаются:

- высокой скоростью передачи;
- простым доступом со стороны центрального процессора. Как говорилось выше, процессор обычно видит управляющие регистры устройства как несколько байтов или слов в адресном пространстве памяти или специальных адресах ввода/вывода. В первом случае для обращения к устройству можно использовать любые команды работы с памятью, а во втором программист должен пользоваться специальными командами `in` и `out`, как в процессорах фирмы Intel;
- низкой помехоустойчивостью и низкими порогами по напряжению и току;
- Сложностью. Часто для полноценной реализации того или иного интерфейса необходимы специальные заказные микросхемы, как для устройств в стандарте шины NuBus фирмы Apple);
- ориентацией на архитектуру конкретного компьютера или семейства компьютеров. Например, реализовать интерфейс для подключения устройства в стандарте Q-bus к шине IBM PC практически невозможно.

Магистрально-модульные устройства, как правило, подключаются к центральному процессору через HBA. Скорость работы такого интерфейса может быть ненамного ниже, чем у системной шины, особенно в случае VME или SCSI, но доступ к устройствам из процессора осуществляется не напрямую через адресное пространство, а путем программирования HBA. Любопытно, что такое косвенное программирование может в итоге оказаться проще прямого программирования контроллера, особенно если HBA хорошо спроектирован.

Некоторые HBA магистрально-модульных систем, однако, отображают управляющие регистры устройств в адресное пространство процессора. Для VME и даже для CAMAC существуют даже специализированные машины-контроллеры, у которых системная шина совпадает с требуемым интерфейсом. Понятно, что для машины общего назначения такой подход часто может оказаться неприемлемым, например, потому что потребуется выделить окно в несколько мегабайт в физическом адресном пространстве в 16 М (IBM PC AT386/ISA) или просто по стоимости.

Стандарты можно также разделить по принципу: открытый стандарт или фирменный. Например, стандарт NuBus является фирменным стандартом Apple, и получить его описание или лицензию на приобретение соответствующих микросхем не так то просто. Стандарты же SCSI или CAMAC полностью открытые, т.е. любой может получить полную документацию и создать собственное устройство. Если его интерфейс соответствует документации, то оно будет работать совместно с любым "правильным" HBA. Как правило, открытые стандарты создаются в исследовательских центрах, (например, CAMAC был разработан в CERN) или разрабатываются правительственными или межправительственными организациями. Однако иногда и фирмы делают свои внутренние стандарты открытыми, как это произошло со стандартом SCSI.

Открытые стандарты особенно выгодны небольшим фирмам и командам разработчиков, у которых нет денег на проталкивание на рынок "фирменной", ни на что не похожей системы со своим шлейфом периферийных устройств. Например, за счет открытых стандартов VME и SCSI выросла фирма Sun Microsystems, начинавшая с изготовления рабочих станций на процессоре 680x0 с системной шиной VME. Однако, когда они выросли, они тут же выпустили рабочие станции линии SPARCStation с фирменной шиной S-bus.

7.2 Драйверы внешних устройств

Драйвер (**driver**) представляет собой специализированный программный модуль, управляющий внешним устройством. Драйверы обеспечивают единый интерфейс к различным устройствам, тем самым "отвязывая" пользовательские программы и ядро ОС от особенностей аппаратуры.

Слово **driver** происходит от глагола **to drive** (**вести**) и переводится с английского языка как извозчик или шофер: тот, кто ведет транспортное средство.

Нужно отметить, что большинство "настоящих" ОС запрещают пользовательским программам непосредственный доступ к аппаратуре. Это делается для повышения надежности и обеспечения безопасности в многопользовательских системах. В таких системах драйверы являются для прикладных программ единственным способом доступа к внешнему миру.

Еще одна важная функция драйвера – это разделение доступа к устройству в средах с вытесняющей многозадачностью. Допускать одновременный неконтролируемый доступ к устройству для нескольких параллельно исполняющихся процессов просто нельзя, потому что для большинства внешних устройств даже простейшие операции ввода/вывода не являются атомарными.

Например, в большинстве аппаратных реализаций последовательного порта RS232 передача байта состоит из четырех шагов: записи значения в регистр данных, записи команды "передавать" в регистр команды, ожидания прерывания по концу передачи и проверки успешности передачи путем считывания статусного регистра устройства. Нарушение последовательности шагов может приводить к непредсказуемым последствиям – например, перезапись регистра данных после подачи команды может приводить к передаче искаженных данных и т.д.

Нельзя также забывать о неприятностях более высокого уровня – например, смешивании вывода разных процессов на печати или устройстве внешней памяти. Поэтому оказывается необходимо связать с каждым внешним устройством какой-то разграничитель доступа во времени. В современных ОС эта функция возлагается именно на драйвер. Обычно один из модулей драйвера представляет собой процесс-монитор (**fork-процесс** в VAX/VMS, **стратегическую функцию** в Unix, OS/2 и Windows NT), выполняющий асинхронно поступающие запросы на доступ к устройству. Подробнее этот механизм обсуждается в разделах 7.3.3 и 7.3.5.

Чаще всего драйверы являются частью ядра системы, исполняются в высшем кольце защиты и имеют доступ на запись к сегментам данных пользовательских программ, а часто и к данным самого ядра. Чтобы избежать этого, пришлось бы выделять каждый драйвер в отдельное адресное пространство и обеспечивать обмен данными между драйвером, ядром и пользовательской программой посредством статических разделяемых буферов или динамического отображения блоков данных между разными адресными пространствами. Оба решения приводят к значительным накладным расходам, а второе еще и предъявляет чрезвычайно жесткие требования к архитектуре диспетчера памяти (подробнее эта проблема обсуждается в разделе 9.4.2).

Поэтому подавляющее большинство современных ОС предоставляет драйверам практически неконтролируемый доступ к данным пользовательских задач и ядра. Это означает, что ошибка в драйвере может привести к разрушению пользовательских программ и самой ОС.

При определении интерфейса драйвера разработчики ОС должны найти правильный баланс между противоречивыми требованиями:

- стремлением как можно сильнее упростить драйвер, чтобы облегчить его разработку и (косвенно) уменьшить вероятность опасных ошибок;
- желанием предоставить гибкий и интеллектуальный интерфейс к разнообразным устройствам.

7.2.1 Функции драйверов

Понятно, что обеспечить единый интерфейс к разнообразным категориям устройств, перечисленным в начале раздела 7 по меньшей мере сложно. Наиболее радикально подошли к этой проблеме разработчики системы UNIX, разделившие все устройства на два класса: **блочные** (высокоскоростные устройства памяти с произвольным доступом, в первую очередь дисковые устройства) и **последовательные** или **символьные** устройства (всё остальное).

При желании можно обойтись только последовательными устройствами, но скорость работы дисковой подсистемы очень важна для производительности всей машины. UNIX *использует различные методы для работы с разными типами устройств. Механизм доступа к блочным устройствам подробнее обсу-*

ждается в 7.3.4

Для последовательных устройств определен следующий набор операций:

- `int open(char * fname, int flags, mode_t mode)` – процедура инициализации устройства. Например, для лентопротяжек эта процедура может включать в себя перемотку ленты к началу. Функция возвращает целочисленный идентификатор-**"ручку"** (**handle**), часто называемый также **дескриптором файла**, который используется программой при всех последующих обращениях к устройству;

- `int read(int handle, char * where, size_t how_much)` – чтение данных с устройства. Если устройство приспособлено только для вывода (например, принтер), эта функция может быть не определена;

- `int write(int handle, char * what, size_t how_much)` – запись данных на устройство. Если устройство приспособлено только для ввода, (например, перфоленточный ввод или мышь), эта функция также может быть не определена;

- `void close(int handle)` – процедура закрытия (освобождения) устройства;

- `int ioctl(int handle, int cmd, ...)` – процедура задания специальной команды, которая не может быть сведена к операциям чтения и записи. Набор таких команд зависит от устройства. Например, для растровых графических устройств могут быть определены операции установки видеорежима; для последовательных портов RS232 это могут быть команды установки скорости, количества бит, обработки бита четности и т.д., для дисководов – команды форматирования носителя;

- `off_t lseek(int handle, off_t offset, int whence)` Long seek – команда перемещения головки чтения/записи к заданной позиции. Некоторые устройства, например модемы или принтеры, могут не иметь этой функции.

  Слово **long** в названии функции появилось по историческим причинам: в версиях Unix для 16-разрядных машин индекс позиции не мог обозначаться словом, потому что это ограничивало бы логическую длину устройства недопустимо малым значением 65384 байт. Поэтому необходимо было использовать двойное слово, что соответствовало типу `long` языка C.

- `caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int handle, off_t offset)` Memory Map – отображение устройства в адресное пространство процесса. Параметр `prot` задает права доступа к отображенному участку: на чтение, на запись и на исполнение. Отображение может происходить на заданный виртуальный адрес, или же система может выбирать адрес для отображения сама.

> Эта функция отсутствовала в старых версиях системы, но большинство современных систем семейства (BSD 4.4, ряд наследников BSD 4.3, SVR4 и Linux) поддерживают ее.
>
> Речь идет об отображении в память данных, хранящихся на устройстве. Для последовательных устройств ввода/вывода, например, для модемов, эту функцию невозможно реализовать разумным образом. Напротив, для лент и других последовательных устройств памяти, поддерживающих функцию `lseek`, отображение может быть реализовано с использованием аппаратных средств виртуализации памяти и операций `read` и `write`. Смысл в специальной функции отображения появляется для драйверов устройств, использующих большие объемы памяти, отображенной в физическое адресное пространство процессора, например, для растровых видеоадаптеров, некоторых звуковых устройств или страниц **общей памяти** (**backpane memory** – двухпортовой памяти, используемой как высокоскоростной канал обмена данными в многопроцессорных системах).

Речь в данном случае идет не о функциях самого драйвера, а о системных вызовах, доступных прикладной программе. В классическом UNIX и некоторых более поздних системах, например в Linux, эти функции непосредственно отображаются на функции драйвера. Однако вместо целочисленной "ручки"драйвер в этих системах получает два указателя на структуры данных ядра. Например, прототип функции драйвера `write` (цитируется по [14]) выглядит в системе Linux как:

Пример 10

```
static int foo_write(struct inode * inode, struct file * file,
                     char * buf, int count)
```

При открытии файла ядро создает экземпляр `struct file` и `struct inode` в таблицах, размещенных в так называемой **пользовательской области** (**user area** – .) Пользовательская область является атрибутом задачи, но размещена в адресном пространстве ядра и задаче непосредственно не доступна. Передаваемый программе целочисленный идентификатор-"ручка"является индексом структур `file` и `inode` в соответствующих таблицах.

Структура `inode` используется файловой системой UNIX для определения устройства. Эта структура хранит права доступа к устройству, ряд сведений, большая часть из которых имеет смысл только для дисковых файлов, и два числа, идентифицирующих устройство: **major** и **minor**. Права доступа проверяются ядром еще до вызова функций дравера. Major задает номер драйвера в таблицах ядра, а minor – номер устройства, управляемого данным драйвером. Например, последовательные порты `/dev/tty0` и `/dev/tty1` будут управляться одним драйвером последовательных портов и иметь значения minor, соответственно, 0 и 1. Ядро использует значение major для того, чтобы выбрать нужный драйвер.

В большинстве ситуаций драйверу интересно только значение поля minor. Однако существуют хитрые драйверы, использующие также поле major. Например, в большинстве систем семейства Unix лентопротяжные устройства имеют два драйвера. Один из драйверов при открытии перематывает ленту к началу, другой не перематывает. Ничем другим эти драйверы не отличаются. В действительности оба драйвера используют один и тот же код, который определяет текущий режим работы в зависимости от значения поля major.

Структура `file` используется для хранения **дескриптора открытого файла**. Если несколько задач обращается к одному устройству или одна задача открывает устройство несколько раз, создается несколько копий структуры `file`. Наиболее важным, с точки зрения драйвера, полем структуры `file` является положение текущего указателя чтения-записи устройства. Для таких устройств, как последовательные порты, которые не имеют функции `lseek`, это поле смысла не имеет.

Нужно отметить, что многие последовательные драйверы не позволяют открывать устройство несколько раз. При последующих вызовах verb"open"для таких устройств будет возвращаться ошибка EBUSY – “Устройство занято”. Например, для лентопротяжек такое поведение представляется вполне разумным. У драйвера, разрешающего создание нескольких дескрипторов файла для одного устройства, все функции должны быть реентерабельными. В классическом Unix и в Linux обеспечение такой реентерабельности возлагается на разработчика драйвера.

Подробнее интерфейс драйвера в Linux обсуждается в документе [14].

Как уже говорилось, для последовательных драйверов в системах семейства Unix системные вызовы отображаются в функции драйвера, хотя во многих случаях вместо прямого вызова драйвера используются более сложные механизмы, обсуждаемые в разделах 7.3.3 и 7.3.5. Чаще всего это делается для того, чтобы обеспечить иллюзию реентерабельности функций драйвера и, таким образом, обеспечить возможность одновременного использования драйвера несколькими процессами.

Понятно, что систему команд лазерного принтера, плоттера или даже просто терминала с позиционированием курсора, вроде vt100, нельзя свести к операциям `read` и `write`, использование функции `lseek` выглядит несколько искусственно, а соответствующих команд `ioctl` получилось бы неприемлемо много. В системах семейства UNIX эта проблема решается очень просто: система берет на себя только передачу данных между компьютером и устройством, а за формирование вывода (возможно, это будут не только данные, но и команды) и, напротив, интерпретацию введенных данных отвечает пользовательская программа.

Отчасти этот подход оправдан тем, что к одному и тому же последовательному порту RS232 можно подключить огромное количество разнообразных устройств:

- классическое устройство ввода/вывода (терминал, принтер),
- сетевой адаптер (модем или нуль-модемный кабель),
- координатное устройство (мышь или дигитайзер),
- устройство звукового ввода/вывода, (голосовой модем),
- устройство графического ввода/вывода (Чем факс-модем не графическое устройство?),
- контрольно-измерительную аппаратуру

и вообще, практически любое достаточно медленное устройство.

Понятно, что поступив таким образом, мы переносим проблему поддержки различных устройств из системы в пользовательскую программу, что не очень хорошо. Это было осознано еще в 70-е гг. при разработке экранного редактора vi, который должен был работать с большим количеством различных видеотерминалов, использовавших различные несовместимые системы команд.

Тогда проблема была решена путем создания системной базы данных по системам команд разных терминалов. Эта база данных представляет собой текстовый файл с именем `/etc/termcap`. Причины, по которым этот файл называется именно так, точно неизвестны. Легенда гласит, что имя происходит от сокращения слов **terminal capabilities** (**возможности терминала**). Файл состоит из абзацев. Заголовок абзаца представляет собой имя терминала, а текст состоит из фраз вида `Name=value`, разделенных символом `':'`. При этом `Name` представляет собой символическое имя того или иного свойства, а `value` – его значение. Это может быть последовательность символов, формирующих соответствующую команду или генерируемых терминалом при нажатии соответствующей клавиши. Кроме того, это может быть именно значение, например, ширина экрана терминала, измеренная в символах.

Для работы с терминальной базой данных было создано несколько библиотек подпрограмм – низкоуровневая библиотека termcap и высокоуровневый пакет curses.

Для терминалов описанный подход оказался если и не идеальным, то, во всяком случае, приемлемым. Но, например, для графических устройств он не подошел – системы команд различных устройств оказались слишком непохожими и не сводимыми к единой системе "свойств". Первым приближением к решению этой проблемы стало создание специализированных программ-**фильтров**. При использовании фильтров пользовательская программа генерирует графический вывод в виде последовательности команд некоторого языка. Фильтр принимает эти команды, синтезирует на их основе изображение, и выводит его на графическое устройство. Вся поддержка различных устройств вывода возлагается на фильтр, пользовательская программа должна лишь знать его входной язык.

Самым удачным вариантом языка графического вывода в наше время считается PostScript – язык управления интеллектуальными принтерами, разработанный фирмой Adobe. PostScript предоставляет богатый набор графических примитивов, но основное его преимущество состоит в том, что это полнофункциональный язык программирования с условными операторами, циклами и подпрограммами. Первоначально этот язык использовался только для управления дорогими моделями лазерных принтеров, но потом появились интерпретаторы этого языка, способные выводить PostScript на более простые устройства. Наиболее известной программой этого типа является GhostScript – программа, реализованная в рамках проекта GNU и доступная как freeware. GhostScript поставляется в виде исходных текстов на языке C, способен работать во многих операционных системах: практически во всех ОС семейства Unix, OS/2, MS/DR DOS и т.д. и поддерживает практически все популярные модели графических устройств, принтеров, плоттеров и прочего оборудования.

Аналогичный подход используется в оконной системе X Window. В этой системе весь вывод на терминал осуществляется специальной программой-сервером. На сервер возлагается задача поддержки различных графических устройств. В большинстве реализаций сервер исполняется как обычная пользовательская программа, осуществляя доступ к устройству при помощи функций `ioctl` "установить видеорежим" и "отобразить видеопамять в адресное пространство процесса".

В обоих случаях "драйвер" оказывается разбит на две части: собственно драйвер, исполняющийся в режиме ядра, который занимается только обменом данными с устройством, и программу, интерпретирующую полученные данные и/или формирующую команды для устройства. Эта программа может быть довольно сложной, но ошибка в ней не будет фатальной для системы, так как она исполняется в пользовательском кольце доступа.

Напротив, в операционных системах OS/2 и Windows NT существует несколько типов драйверов с различными наборами функций. Так, в OS/2 используются драйверы физических устройств следующих типов:

- простые драйверы последовательных устройств ввода/вывода, аналогичные драйверам символьных устройств в Unix;
- драйверы **запоминающих устройств прямого доступа**, аналогичные драйверам блочных устройств в Unix;
- драйверы видеоадаптеров, используемые графической оконной системой Presentation Manager (PM);
- драйверы позиционных устройств ввода (мышей и др.), также используемые PM;
- драйверы принтеров и других устройств вывода твердой копии;
- драйверы звуковых устройств, используемые подсистемой “мультимедиа” MMOS/2;
- драйверы сетевых адаптеров стандарта NDIS, используемые сетевым программным обеспечением фирм IBM и MicroSoft;
- драйверы сетевых адаптеров стандарта ODI, используемые программным обеспечением фирмы Novell;
- **DMD** (**Device Manager Driver** – драйвер-менеджер класса устройств).
- различного рода “фильтры”, например, ODINSUP.SYS – преобразователь ODI-интерфейса в NDIS.

На функциях DMD следует остановиться подробнее. Рассмотрим достаточно типичную конфигурацию, содержащую HBA магистрально-модульного стандарта SCSI, к которому подключены пять устройств: жесткий диск, привод CD-ROM, магнитооптический диск, лентопротяжка и сканер. При этом каждое из устройств обладает достаточно серьезной спецификой, так что их сложно свести к общему набору функций.

Жесткий и магнитооптический диски наиболее схожи между собой, так как и то, и другое является запоминающим устройством большой емкости с произвольным доступом. Однако жесткий диск является неудаляемым устройством, а магнитооптический носитель можно извлечь из привода, не выключая компьютера. Это накладывает определенные требования на стратегию кэширования соответствующего устройства и требует от драйвера умения понимать и обрабатывать аппаратный сигнал о смене устройства.

CD-ROM, в свою очередь, нельзя рассматривать как удаляемый диск, доступный только для записи: ведь практически все CD-ROM приводы, кроме функции считывания данных, еще имеют функцию проигрывания музыкальных компакт-дисков.

Лентопротяжка и сканер попросту не являются устройствами памяти прямого доступа, а сканер вообще нельзя рассматривать как устройство памяти.

Когда OS/2 управляет описанной аппаратной конфигурацией, оказываются задействованы пять DMD:

`OS2DASD.DMD` управляет классом запоминающих устройств прямого доступа (**DASD** – Direct Access Storage Device), и предоставляет стандартные функции для доступа к дискам.

`OPTICAL.DMD` обеспечивает управление устройствами прямого доступа с удаляемыми носителями. Основная его задача – обработка аппаратного сигнала смены носителя и оповещение других модулей системы (дискового кэша, файловой системы) об этой смене.

`OS2CDROM.DMD` обеспечивает специфические для приводов CD-ROM функции, например проигрывание аудизаписей.

Каждый из этих DMD не работает непосредственно с аппаратурой, а транслирует запросы пользовательских программ и других модулей ядра (в первую очередь, менеджеров файловых систем) в запросы к драйверу нижнего уровня. Такой многослойный подход позволяет вынести общую для класса устройств логику в DMD и не заниматься повторной реализацией этой логики в каждом новом драйвере.

В случае SCSI ситуация дополнительно усложняется тем, что все запросы к устройствам должны быть оформлены в виде команд SCSI и пройти через HBA, доступ к которому нужно синхронизовать, не допуская попыток одновременно передать в HBA две команды. Именно эти функции, то есть:

- трансляцию абстрактных запросов к абстрактным дискам, приводам CD-ROM и т.д. в последовательности команд SCSI,
- обеспечение последовательной передачи этих команд через HBA
- диспетчеризацию пришедших в ответ на команды данных или кодов ошибки

исполняет четвертый DMD: `OS2SCSI.DMD`. Этот DMD передает запросы на отправку команд SCSI **ADD** (**Adapter Device Driver** – драйверу устройства-адаптера), то есть собственно драйверу HBA. От ADD требуется только умение передавать команды в шину SCSI и производить первичный анализ пришедших на команды ответов: какой из ранее переданных команд соответствует ответ, чем завершилась операция – успехом или ошибкой, пришли ли в ответ данные и если пришли, то сколько именно и куда их положить и т.д.

Пятый DMD – `OS2ASPI.DMD` – обеспечивает сервис **ASPI** (**Advanced SCSI Programming Interface** – продвинутый интерфейс для программирования SCSI). Этот сервис дает возможность прикладным программам и другим драйверам формировать произвольные команды SCSI и таким образом осуществлять доступ к устройствам, которые не являются дисками. Сервисом ASPI пользуются драйверы лентопротяжки и сканера.

Видно, что разработчики Unix и OS/2 совершенно по-разному подошли к решению проблемы, поставленной в начале данного раздела. Впрочем, нужно отметить, что большинство современных систем семейства Unix сделали не один шаг на пути к аналогичной многослойной структуре драйвера.

Например, в Linux и системах линии BSD Unix за управление устройствами SCSI отвечает специализированный модуль ядра, по функциям вполне аналогичный вышеперечисленным пяти DMD. Он изображает из себя группу драйверов блочных и последовательных устройств (дисков, приводов CD-ROM, лентопротяжек и т.д.) и транслирует запросы ко всем этим устройствам в вызовы достаточно простого драйвера HBA, который аналогичен драйверу адаптера в OS/2 и непосредственно недоступен для прикладных программ.

В этих же системах все драйверы терминальных устройств должны уметь обрабатывать достаточно обширный набор `ioctl` и выполнять ряд важных функций по управлению заданиями. Для реализации соответствующей логики ядро предоставляет таким драйверам сервисную библиотеку, которую тоже можно рассматривать как аналог DMD.

В Unix System V Release 3 драйверы последовательных устройств реализуются с использованием модуля STREAMS. С каждым драйвером последовательного устройства связан абстрактный поток данных, на который можно повесить практически неограниченное количество промежуточных фильтров пред- и постобработки данных и модулей обработки команд `ioctl`.

Однако такой многослойно-модульный подход не охватывает всей подсистемы ввода/вывода, что порой приводит к неприятным последствиям. Например, системы семейства Unix традиционно имеют проблемы с удаляемыми носителями в устройствах памяти произвольного доступа. Проблема состоит в том, что ядро Unix вообще не имеет представления о том, что устройство может быть внезапно заменено. При смене устройства оператор должен явным образом сообщить об этом системе, используя команды `umount` и `mount`. Если, к примеру, заменить смонтированную дискету, не сообщив об этом системе, то можно ожидать совершенно неожиданных результатов, начиная от чтения содержимого удаленной дискеты и кончая порчей данных на новой дискете и даже разрушением системы.

Впрочем, обработка смены носителя является гораздо более сложной проблемой, чем просто написание эквивалента `OPTICAL.DMD`, который детектирует сигнал смены носителя. Ведь этот сигнал должен быть передан модулям управления дисковым кэшем и файловой системой, которые, в свою очередь, должны разумно обработать его: как минимум, дисковый кэш должен объявить все связанные с диском буферы неактуальными, а менеджер файловой системы должен сбросить все свои внутренние структуры данных, связанные с удаленным диском, и объяснить всем пользовательским программам, работавшим с этим диском, что их данные пропали. Другие аспекты работы с удаляемыми носителями обсуждаются в разделе 8.4.

7.3 Вызов функций драйвера

Выше был описаны варианты наборов функций драйвера устройства, но практически не обсуждался механизм взаимодействия драйвера с остальными модулями системы и пользовательскими программами.

Следует провести различие между системными вызовами и функциями ядра, доступными для драйверов. Наборы системных вызовов и драйверных сервисов совершенно независимы друг от друга. Как правило, системные вызовы недоступны для драйверов, а драйверные сервисы – для пользовательских программ.

Системный вызов включает в себя переключение контекста между пользовательской программой и ядром. В системах с виртуальной памятью во время такого переключения процессор переходит из “пользовательского” режима, в котором запрещены или ограничены доступ к регистрам диспетчера памяти, операции ввода/вывода и ряд других действий, в “системный”, в котором все ограничения снимаются. Обычно системные вызовы реализуются с использованием специальных команд процессора, чаще всего – команды программного прерывания.

Напротив, драйвер исполняется в “системном” режиме процессора и, как правило, в контексте ядра, поэтому для вызова сервисов ядра драйверу не надо делать никаких переключений контекста. Практически всегда такие вызовы реализуются обычными командами вызова подпрограммы.

Еще одно важное различие состоит в том, что системные вызовы практически всегда являются реентерабельными – ядро либо обеспечивает подлинную реентерабельность, либо обеспечивает иллюзию реентерабельности благодаря тому, что исполняется с более высоким приоритетом, чем все пользовательские программы. Напротив, доступные драйверам сервисы ядра делятся на две группы – те сервисы, которые можно вызывать из обработчиков прерываний и те, которые нельзя. Сервисы, доступные для обрабочиков прерываний, должны удовлетворять двум требованиям: они должны быть реентерабельными и завершаться за гарантированное время. Например, выделение памяти может потребовать сборки мусора или даже поиска жертвы для удаления в адресных пространствах пользовательских задач. Кроме того, выделение памяти требует работы с разделяемым ресурсом (пулом памяти ядра) и его достаточно сложно реализовать реентерабельным. Аналогично, копирование данных между пользовательским и системным адресными пространствами может привести к возникновению страничного отказа, время обработки которого может быть непредсказуемо большим.

Ниже, для краткости, мы будем называть доступные для обрабочиков прерываний сервисы реентерабельными, хотя для них важна не только реентерабельность, но и завершение в течении фиксированного времени. К реентерабельные сервисам относятся примитивы синхронизации (и часто только они) – установка/очистка семафоров, активизация или деактивация пользовательских и системных процессов, манипуляции над очередями запросов и т.д. Сервисы, занимающиеся распределением ресурсов, например функции выделения памяти, чаще всего являются нереентерабельными.

Мы не будем подробно обсуждать методы передачи данных между пользовательским и системным адресными пространствами при обработке системного вызова, поскольку они сильно зависят от архитектуры диспетчера памяти. Различные процессоры предлагают для этого различные по изощренности методы, описание которых отняло бы много времени, но не было бы поучительным.

Например, в VAX пользовательские адреса целиком отображаются в системное адресное пространство. При этом пользовательская программа может использовать только первые два гигабайта из четырех, которые можно адресовать 32-битовым адресом. Вторые два гигабайта зарезервированы для системы. Исполняя системный вызов, ядро отображает в свое адресное пространство всю память программы, сделавшей вызов, и, таким образом, получает прямой доступ ко всем пользовательским данным. Подробнее изучить архитектуру виртуальной памяти системы VAX можно по работам [10, 11].

Обработку запроса можно разделить на три фазы: предобработки, собственно исполнения запроса и постобработки. Пользовательская программа запрашивает операцию, исполняя соответствующий системный вызов. В ОС семейства Unix это будет системный вызов `read(int file, void * buffer, size_t size)`.

Предобработка выполняется модулем системы, который обычно исполняется с приоритетом пользовательского процесса и нередко в контексте этого процесса, но имеет привилегии ядра.

Фаза предобработки включает:

- проверку допустимости параметров. Например, пользователь должен иметь право выполнять запрошенную операцию над данным устройством, адрес буфера должен быть допустимым адресом пользовательского адресного пространства и т.д.;
- возможно, копирование или отображение данных из пользовательского адресного пространства в системное.
- возможно, преобразование выводимых данных. Например, в системах семейства Unix при выводе на терминал система может заменять символ горизонтальной табуляции на соответствующее число пробелов (если терминал не поддерживает горизонтальную табуляцию) и преобразовывать символ перевода строки. Дело в том, что внутри системы в качестве разделителя строк используется символ новой строки `'\n'` (ASCII NL), а различные модели терминалов и принтеров могут использовать также `'\r'` (ASCII RET, возврат каретки) или последовательности `'\r''\n'` или `'\n''\r'`;
- возможно, обращение к процедурам драйвера. Эти процедуры могут блокировать код и данные драйвера в физической памяти и выделять буфера для ПДП. Эти операции реализуются нереентерабельными сервисами ядра и не всегда могут быть выполнены драйвером во время обработки запроса.
- Передача запроса драйверу. Некоторые системы реализуют передачу запроса как простой вызов соответствующей функции драйвера, но чаще используются более сложные асинхронные механизмы, которые будут обсуждаться ниже.

Выполнив запрос, драйвер активизирует программу постобработки, которая анализирует результат операции, предпринимает те или иные действия по восстановлению в случае неуспеха, копирует или отображает полученные данные в пользовательское адресное пространство и оповещает программу о завершении запроса.

Некоторые системы на этой фазе также делают преобразование введенных данных. В качестве примера можно вновь привести системы семейства Unix, которые при вводе с терминала выполняют трансляцию символа перевода строки и ряд других операций по редактированию, например, стирание последнего введенного символа по запросу пользователя. Разбиение потока терминальных данных на строки в этих системах также происходит на фазе постобработки.

В той или иной форме эти три фазы обработки запроса ввода/вывода присутствуют во всех многопроцессных системах.

Сейчас нас интересует гораздо более простой, на первый взгляд, вопрос: каким образом процедура предобработки обращается к драйверу? И каким образом драйвер потом передает результат исполнения запроса процедуре постобработки? Ответ на эти вопросы далеко не так прост.

7.3.1 Синхронный ввод/вывод в однозадачных системах

Самым простым механизмом вызова функций драйвера был бы косвенный вызов соответствующих процедур, составляющих тело драйвера, подобно тому, как это делается в MS DOS.

Например, программа, желающая прочитать строку с клавиатуры, вызывает функцию `read` драйвера консоли. Функция `read` анализирует состояние клавиатуры; если была нажата клавиша, запоминает ее код в буфере; при нажатии клавиши "конец ввода" или заполнении буфера функция возвращает управление программе. Такой метод очень прост в реализации, но, как показывает более внимательный взгляд на проблему, совершенно неадекватен, даже для однопрограммных однопользовательских систем.

Справедливости ради следует отметить, что даже MS DOS использует описанный метод для обмена с блочными устройствами (да и то если не загружен дисковый кэш), но клавиатуру обрабатывает более разумным методом с использованием **прерываний**.

Начнем с того, что пользователь может нажимать клавиши в любой момент, а не только когда программа попросила его что-то ввести. Например, многие пользователи командных интерпретаторов любят режим **type ahead** (дословно переводится как **печать вперед**), когда система исполняет одну команду, а пользователь набирает следующую, не дожидаясь приглашения. Понятно, что реализовать такой режим путем опроса клавиатуры невозможно.

Дочитав до этого места, средний досовский хакер должен воскликнуть: "Но какой же [подставьте ваше любимое ругательство] будет опрашивать клавиатуру? Ведь для этого и придуманы прерывания!"

Действительно, прерывания помогают реализовать режим type ahead и решить много других проблем. Например, если клавиатура генерирует прерывание при каждом нажатии на клавишу, мы можем реализовать драйвер клавиатуры следующим образом:

Пример 11

```
// Пример драйвера клавиатуры.
// (C) Дмитрий Иртегов, 1995.
```

```
char buffer[1024];
int last_char, first_char;
int count; // Счетчик символов в буфере

// Эта функция считывает код нажатой клавиши
// из физического порта клавиатуры.
extern char get_char();

interrupt keyboard_handler() {
  // Считать код нажатой клавиши
  // и поместить его в кольцевой буфер.
  if (last_char == first_char) return;
  buffer[last_char] = get_char();
  last_char = (last_char+1) & 1023;
  count++;
}

void init_driver() {
  set_interrupt_handler(KEYBOARD, keyboard_handler);
  last_char = 0;
  first_char = 1023;
  count = 0;
}

int read(char * where, int how_much) {
  // Копирует содержимое буфера куда скажут.
  // Возвращает количество скопированных символов
  // и сбрасывает счетчик
  int i = how_much;
  int tmp;

  disable_interrupts();

  if (i < count) i = count;
  // Мы не можем считать больше символов, чем есть в буфере
  tmp = i;
  for (; i; i--) {
    *where = buffer[first_char];
    first_char = (first_char+1) & 1023;
    count--;
  }
```

```
  enable_interrupts();
  return tmp;
}

int write(char * what, int how_much) {
  return -1;
  // Не умеем записывать данные в клавиатуру.
}
```

Программа 11 максимально упрощена. Например, функция чтения возвращает все, что накопилось в буфере к моменту вызова, тогда как “настоящие” драйверы терминала обычно осуществляют ввод по строкам, и т.д.

Тем не менее программа 11 иллюстрирует основную идею программ такого рода:

- накапливать вводимые данные в буфере по мере их поступления,
- отдавать содержимое буфера прикладной программе, когда она попросит

На примере этой же программы мы можем увидеть деление драйвера на две части – “синхронную” программу и асинхронный обработчик прерывания. Такое деление в той или иной форме существует практически во всех системах, использующих прерывания или аналогичные прерываниям механизмы.

Казалось бы, теперь все хорошо. Но при более внимательном рассмотрении мы обнаруживаем вторую проблему.

Наша процедура чтения возвращает все, что было в буфере на момент вызова, и по существу ничего не делает, если буфер был пуст. Но, как уже говорилось, “настоящие” драйверы терминала осуществляют ввод по строкам. Как минимум, если буфер пуст, мы должны были бы остановиться и подождать, пока там что-нибудь не появится. Возможно, нам следует предусмотреть какое-либо средство, позволяющее синхронной части драйвера ожидать, пока обработчик прерывания не просигнализирует нам о вводе нужного числа символов. Например, мы можем проверять счетчика накопленных в буфере символов и переводить процессор в состояние ожидания, если этот счетчик равен нулю.

Если читатель внимательно читал всю книгу, на этом месте он должен сказать: “Но мы это уже проходили!”. Действительно, подобная ситуация рассматривалась в разделе 4.3.1, и там было показано, что проверка флага или другого условия, устанавливаемого обработчиком прерывания, с блокировкой при невыполнении условия, создает неприятные проблемы.

В однопроцессной системе мы можем позволить себе роскошь циклически опрашивать значение счетчика байтов, накопленных в буфере. Это снимает проблему, с которой мы столкнулись в примере 5, но само по себе превращается в проблему, когда мы переходим в многопроцессную среду.

Такие проблемы, например, возникают при исполнении программ для MS DOS в DOS-эмуляторе OS/2 или под управлением DesqView. Обе эти системы вынуждены использовать нетривиальные алгоритмы для выявления программ, циклически опрашивающих клавиатуру или последовательный порт, потому что такие программы сильно и плохо влияют на поведение всей системы. Для призвания к порядку программ, не выявляемых такими алгоритмами, существует специальная утилита с характерным названием tame – "приручить", использующая еще более изощренные методы. Обсуждение этих методов увело бы нас далеко от основной темы.

Даже если мы будем использовать для синхронизации что-то типа семафора, останется очень серьезная трудность: функция `read` вызывается из ядра системы, поэтому, заблокировав эту функцию, мы заблокируем ядро, что совершенно недопустимо в многопроцессной системе.

Еще хуже ситуация с запоминающими устройствами типа лент или дисководов. При чтении данных драйвер такого устройства должен:

- если речь идет о дискете, то включить мотор дисковода и дождаться, пока диск разгонится до рабочей скорости.
- дать устройству команду на перемещение считывающей головки.
- дождаться прерывания по концу операции перемещения.
- запрограммировать ПДП и инициировать операцию чтения.
- дождаться прерывания, сигнализирующего о конце операции чтения.

Лишь после этого можно будет передать данные программе. В обоих промежутках между инициацией запроса и окончанием операции драйвер должен отдавать управление системе, иначе обращения к дисководу будут надолго блокировать все остальные процессы в системе.

Например, Windows 3.x в `enhanced` режиме предоставляет вытесняющую многозадачность для **VDM** (**Virtual DOS Machine** – Виртуальная машина [для] DOS), однако сама Windows 3.x использует DOS для обращения к дискам и дискетам. Ядро однозадачной DOS не умеет отдавать управление другим процессам во время исполнения запросов ввода/вывода. В результате во время обращения к диску все остальные задачи оказываются заблокированы. У современных PC время исполнения операций над жестким диском измеряется десятыми долями секунды, поэтому фоновые обращения к жесткому диску почти не приводят к нарушениям работы остальных программ. Однако скорость работы гибких дисков осталась достаточно низкой, поэтому работа с ними в фоновом режиме блокирует систему на очень заметные промежутки времени.

Эффектная и убедительная демонстрация этой проблемы очень проста: достаточно запустить в фоновом режиме форматирование дискеты или просто команду `COPY C:\TMP\*.* A:`, если в директории `C:\TMP` достаточно много данных. При этом работать с системой будет практически невозможно: во время обращений к дискете даже мышиный курсор не будет отслеживать движений мыши, будут теряться нажатия на клавиши и т.д.

Windows 95 использует несколько методов обхода DOS при обращениях к диску, поэтому пользователи этой системы не всегда сталкиваются с описанной проблемой. Однако при использовании блочных драйверов реального режима система по прежнему использует DOS в качестве подсистемы ввода/вывода и работа с дискетами в фоновых задачах также нарушает работу задач первого плана.

Сказанное означает, что переход от однозадачной или кооперативно многозадачной системы к вытесняющей многозадачности может потребовать не только изменения планировщика, но и радикальной переделки всей подсистемы ввода/вывода, в том числе и самих драйверов.

Переделка драйверов означает, что все независимые изготовители оборудования также должны будут обновить свои драйверы. Организация такого обновления оказывается сложной, неблагодарной и часто попросту невыполнимой задачей – например, потому, что изготовитель оборудования уже не существует как организация или отказался от поддержки данного устройства. Поэтому интерфейс драйвера часто оказывается наиболее консервативной частью ОС.

В качестве примера такого консерватизма можно привести подсистему ввода/вывода OS/2. Совместный проект фирм IBM и Microsoft, OS/2 1.x разрабатывалась как операционная система для семейства персональных компьютеров Personal System/2. Младшие модели семейства были основаны на 16-разрядном процессоре 80286, поэтому вся ОС была полностью 16-битной.

Позднее разработчики фирмы IBM реализовали 32-битную OS/2 2.0, но для совместимости со старыми драйверами им пришлось сохранить 16-битную подсистему ввода/вывода. Все точки входа драйверов должны находиться в 16-битных (`'USE16'`) сегментах кода; драйверам передаются только 16-разрядные `'far'`) указатели и т.д. По утверждению фирмы IBM, они рассматривали возможность реализации также и 32-битных драйверов, но их измерения не показали значительного повышения производительности при переходе к 32-битной модели.

Так или иначе, OS/2 2.x и последующие версии системы по-прежнему используют 16-битные драйверы последовательных, блочных, координатных и сетевых устройств. Ряд ключевых модулей ядра в этих системах по прежнему использует 16-битный код. Благодаря этому сохраняется возможность использовать драйверы, разработанные еще в конце 80-х и рассчитанные на OS/2 1.x. Эта возможность оказывается особенно полезна при работе со старым оборудованием.

Напротив, разработчики фирмы Microsoft отказались от совместимости с 16-битными драйверами OS/2 1.x в создававшейся ими 32-битной версии OS/2, называвшейся OS/2 New Technology. Фольклор утверждает, что именно это техническое решение оказалось причиной разрыва партнерских отношений между Microsoft и IBM, в результате которого OS/2 NT вышла на рынок под названием Windows NT 3.1.

Трудно сказать, было ли это решение оправданным. За чисто 32-битное ядро пришлось расплачиваться потерей совместимости со старыми драйверами и резким сужением набора поддерживаемых внешних устройств, а наблюдаемых технических преимуществ чистая 32-битность не дала. Во всяком случае, по производительности и потребляемым ресурсам Windows NT значительно уступает 32-разрядным версиям OS/2.

Фактически, совместимость со старыми драйверами часто оказывается по важности сопоставима с совместимостью со старыми приложениями. Отказ от такой совместимости на практике означает "брошенное" периферийное оборудование и, как следствие, "брошенных" пользователей, которые оказываются вынуждены либо отказываться от установки новой системы, либо заменять оборудование. Оба варианта, естественно, не улучшают отношения пользователей к поставщику ОС, поэтому многие поставщики просто не могут позволить себе переделку подсистемы ввода/вывода.

Именно из-за этого, например, фирма Apple до сих пор не может реализовать вытесняющую многозадачность в MacOS. По этой же причине фирма Microsoft так долго держалась за кооперативную многозадачность в различных вериях MS Windows и даже в Windows 95 им не удалось полностью преодолеть наследие однозадачной DOS.

Однако нужно отметить, что ряд однозадачных ОС, например RT-11SJ фирмы DEC, использует реентерабельную подсистему ввода/вывода, пригодную для реализации вытесняющей многозадачности. Такое проектирование "на вырост" представляется очень удачной технической политикой, так как упрощает разработчикам и пользователям переход к более мощным версиям системы.

7.3.2 Синхронный ввод/вывод в многозадачных системах

В предыдущем разделе была сформулирована одна из проблем, возникающих при организации ввода/вывода в многозадачных системах – обращения к устройствам ввода/вывода, особенно к низкосокоростным устройствам, не должны без необходимости блокировать исполнение других процессов. Другая проблема состоит в том, что драйвер устройства работает с разделяемым ресурсом и поэтому является нереентерабельным. Ядро или сам драйвер должны предоставить какие-то средства, защищающие код драйвера от повторных вызовов. Рассмотрим некоторые подходы к решению этих проблем.

В "классических" версиях Unix и некоторых других системах этого семейства, например в Linux, драйвер последовательного устройства исполняется в рамках того процесса, который издал запрос, хотя и с привилегиями ядра. Ожидая реакции устройства, драйвер переводит процесс в состояние ожидания и вызывает функцию ядра `schedule()`. Эта функция аналогична обсуждавшейся в п. 6.1 функции переключения процессов: она передает управление первому активному процессу, стоящему в очереди с наивысшим приоритетом. Она возвращает управление только тогда, когда до процесса вновь дойдет очередь. Когда устройство генерирует прерывание, оповещающее о завершении операции, обработчик прерывания выводит процесс из состояния ожидания. Для перевода процесса в состояние ожидания и обратно используются реентерабельные сервисы ядра.

Ниже приводится скелет функции `write()` драйвера последовательного устройства в системе Linux. Текст цитируется по документу [14].

Пример 12

```
static int foo_write(struct inode * inode, struct file * file,
                     char * buf, int count)
{
    unsigned int minor = MINOR(inode->i_rdev);
    unsigned long copy_size;
    unsigned long total_bytes_written = 0;
    unsigned long bytes_written;
    struct foo_struct *foo = &foo_table[minor];

    do {
        copy_size = (count <= FOO_BUFFER_SIZE ?
                                 count : FOO_BUFFER_SIZE);
        memcpy_fromfs(foo->foo_buffer, buf, copy_size);

        while (copy_size) {
        /* initiate interrupts */

            if (some_error_has_occured)  {
               /* handle error condition */
            }

            current->timeout = jiffies + FOO_INTERRUPT_TIMEOUT;
         /* set timeout in case an interrupt has been missed */
            interruptible_sleep_on(&foo->foo_wait_queue);
            bytes_written = foo->bytes_xfered;
            foo->bytes_written = 0;
            if (current->signal & ~current->blocked) {
                if (total_bytes_written + bytes_written)
                    return total_bytes_written + bytes_written;
                else
                  return -EINTR; /* nothing was written, system
                             call was interrupted, try again */
            }
        }

        total_bytes_written += bytes_written;
        buf += bytes_written;
        count -= bytes_written;

    } while (count > 0);

    return total_bytes_written;
```

```
}

static void foo_interrupt(int irq)
{
    struct foo_struct *foo = &foo_table[foo_irq[irq]];

 /* Here, do whatever actions ought to be taken on an interrupt.
    Look at a flag in foo_table to know whether you ought to be
    reading or writing. */

 /* Increment foo->bytes_xfered by however many characters were
    read or written */

    if (buffer too full/empty)
        wake_up_interruptible(&foo->foo_wait_queue);
}
```

Эта программа сложнее, чем приведенная в примере 11. Например, предполагается, что драйвер 12 способен обслуживать несколько устройств. Номер устройства, к которому сейчас осуществляется обращение, определяется значением переменной `minor`, которая, в свою очередь, определяется на основании содержания идентификационной записи устройства – его **инода** (**i-node**). Указатель на инод передается первым параметром.

Тем не менее, структуры программ 11 и 12 очень близки. Основное отличие состоит в том, что в 11 мы обошли стороной проблему освобождения процессора на время обработки запроса, пример же 12 показывает, как эта проблема может быть решена, если ядро системы предоставляет соответствующие сервисы.

Перечислим по порядку все сервисы ядра, к которым обращается драйвер.

`memcpy_fromfs(char * dst, const char * src, size_t size)` используется для передачи данных из пользовательского адресного пространства в системное. При этом `src` – это адрес в системном пространстве, а `dst` – в пользовательском. Устройство принимает по `copy_size` символов за раз и генерирует прерывание, когда оказывается готово принять следующий блок данных. Знатоки архитектуры `x86` могут догадаться, что селектор пользовательского адресного пространства передается драйверу в регистре `FS`.

Драйвер копирует данные из пользовательского сегмента в свой внутренний буфер. Это необходимо потому, что функция `memcpy_fromfs` *может приводить к возникновению страничных отказов, обработка которых может занимать непредсказуемое время. Поэтому* `memcpy_fromfs` *нельзя вызывать из обработчика прерывания.*

Использование внутреннего буфера приводит к увеличению ядра системы и накладным расходам на копирование. Альтернативой такому копированию могло бы служить динамическое отображение данных из пользовательского адресного пространства в системное. При этом данные нужно не только отображать, но и блокировать в физической памяти, чтобы обработчик прерывания мог без опасения работать с ними. Разработчики оригинальной системы Unix, а вслед за ними и Линус Торвальдс (разработчик Linux) сочли, что такое отображение чрезмерно усложнило бы механизм управления виртуальной памятью.

`void interruptible_sleep_on(struct wait_queue ** p)` устанавливает процесс в заданную очередь ожидания и вызывает `schedule()`. Соответственно, эта функция возвращает управление только когда процесс вернется в активное состояние. Обратная ей функция
`void wake_up_interruptible(struct wait_queue ** p)` переводит первый процесс из очереди в активное состояние и устанавливает его в конец активной очереди соответствующего приоритета.

Обратите внимание, что кроме инициализации устройства драйвер перед засыпанием еще устанавливает "будильник" – таймер, который должен разбудить процесс через заданный интервал времени. Это необходимо на случай, если произойдет аппаратная ошибка и устройство не сгенерирует прерывания. Если бы такой будильник не устанавливался, драйвер в случае ошибки мог бы заснуть навсегда, заблокировав при этом пользовательский процесс.

Еще одна особенность синхронных драйверов в ОС семейства Unix состоит в том, что все операции ввода/вывода, а также все остальные операции, переводящие процесс в состояние ожидания, могут быть прерваны **сигналом**[2]. При этом системный вызов возвращает код ошибки EINTR, говорящий о том, что вызов был прерван и, возможно, операцию следует повторить. Например, пользовательский процесс может использовать сигнал SIGALARM для того, чтобы установить свой собственный будильник, сигнализирующий, что операция над устройством исполняется подозрительно долго.

Если драйвер не установит своего будильника и не станет отрабатывать сигналы, посланные процессу, может возникнуть очень неприятная ситуация. Дело в том, что в Unix все сигналы, в том числе и сигнал безусловного убийства SIGKILL, обрабатываются процедурой постобработки системного вызова. Если драйвер не передает управления процедуре постобработки, то и сигнал, соответственно, оказывается необработанным, поэтому процесс остается висеть. Других средств, кроме посылки сигнала, для уничтожения процесса в системах семейства Unix не предусмотрено. Поэтому процесс, зависший внутри обращения к драйверу, оказывается невозможно прекратить ни изнутри, ни извне. Один из авторов столкнулся с этим при эксплуатации многопроцессорной версии системы SCO Open Desktop 4.0.

Система была снабжена лентопротяжным устройством, подключаемым к внешней SCSI-шине. Из-за аппаратных проблем это устройство иногда "зависало", прекращая отвечать на запросы системы. Драйвер лентопротяжки иногда правильно отрабатывал это состояние как аппаратную ошибку, а иногда тоже впадал в ступор, не пробуждаясь ни по собственному будильнику, ни по сигналам, посланным другими процессами (По

[2] Понятие сигнала кратко обсуждалось в 4.3.1.

имеющимся у автора сведениям, эта проблема специфична именно для многопроцессорной версии системы. По видимому это означает, что ошибка допущена не в драйвере, а в коде сервисных функций). В результате процесс, обращавшийся в это время к ленте, также намертво зависал и от него оказывалось невозможно избавиться.

Из-за наличия неубиваемого процесса оказывалось невозможно выполнить нормальное закрытие системы; в частности, оказывалось невозможно размонтировать файловые системы, где зависший процесс имел открытые файлы. Выполнение холодной перезагрузки системы с неразмонтированными файловыми томами приводило к неприятным последствиям для этих томов. Одна из аварий, к которым это привело, подробно описывается в разделе 8.4.2.

Неустановка будильника и неумение обрабатывать сигналы являются грубыми ошибками, которые возможны не только в синхронной модели драйвера, но и в большинстве асинхронных моделей, которые будут рассматриваться ниже.

Синхронная модель драйвера очень проста в реализации, но имеет существенный недостаток – приведенный в примере 12 драйвер нереентерабелен. Предполагается, что такой драйвер позволяет работать с устройством только одному процессу, а на обращения со стороны остальных процессов отвечает "устройство занято". Такое поведение было бы разумным для лентопротяжек или печатающих устройств, но неудобно для терминалов и совершенно недопустимо для дисков. Поэтому даже в полностью монолитных системах семейства Unix, таких как Linux и FreeBSD, драйверы блочных устройств используют асинхронную модель (см. 7.3.4), а синхронные драйверы терминалов гораздо сложнее примера 12.

7.3.3 Асинхронный ввод/вывод

Иной подход используется в микроядерных системах: в таких ОС "синхронная" часть драйвера выделяется в отдельный процесс, взаимодействующий с остальной системой через операции `send/receive`.

Ниже приведен пример такого драйвера, написанный на псевдокоде. Псевдокод представляет собой язык C, расширенный аналогом оператора `select` языка Ada. Этот оператор позволяет ожидать данные одновременно из нескольких портов; при приходе данных из какого-либо порта исполняется соответствующий код. Синтаксис оператора аналогичен оператору `switch()` языка C.

Предполагается, что микроядро поддерживает следующие операции:

`send(Port port, char * data, int data_size)`: послать данные в порт.

`receive(Port port, char * data, int data_size)`: получить данные из порта. Если `data == 0`, данные игнорируются.

`when(Port port, char * data, int data_size)`: аналогично `receive`, но используется в операторе `select` при ожидании данных из нескольких портов.

Пример 13

```
// Пример драйвера клавиатуры в микроядерной системе.
```

```
// (C) Дмитрий Иртегов 1995

// Порт для обмена данными с ядром ОС.
extern Port OS_port;

// Порт для синхронизации с обработчиком прерываний
static Port internal_port;

char buffer[1024];
int last_char, first_char;
int count; // Счетчик символов в буфере

// Эта функция считывает код нажатой клавиши
// из физического порта клавиатуры.
extern char get_char();

interrupt keyboard_handler() {
  // Считать код нажатой клавиши
  // и отдать его ‘‘синхронному" процессу
  char ch;
  ch = get_char();
  send(internal_port, &ch, 1);
}

void driver_body() {
  char ch;
  int i;

  while(1) {
    select(internal_port, OS_port) {
      when (internal_port, &ch, 1):
      // Получить символ от обработчика прерывания
      // и поместить его в кольцевой буфер.
      // Если буфер полон, издать звуковой сигнал.
        if (last_char != first_char) {
          buffer[last_char] = ch;
          last_char = (last_char+1) & 1023;
          count++;
        } else beep();
      continue;

      when (OS_port,
```

```
        &request_header,
        sizeof(request_header)):

    switch(request_header.rq_type) {
case RQ_READ:
  // Сформировать заголовок.
  request_header.rq_result-code = READ_OK.
  if (count == 0) {
          // Если буфер пуст, считать хотя бы один символ.
    receive(internal_port, &ch, 1);
          buffer[last_char] = ch;
          last_char = (last_char+1) & 1023;
          count++;
  }

        if (request_header.data_size > count)
    request_header.data_size = count;
  // Отправить заголовок в порт.
  send(OS_port,
      &request_header,
      sizeof(request_header));

  i = request_header.data_size;
  // Послать данные из буфера в порт.
  for (; i; i--) {
    send(OS_port, &buffer[first_char], 1);
    first_char = (first_char+1) & 1023;
    count--;
  }

break;

case RQ_SHUTDOWN: // Завершить работу драйвера:
  return;

case RQ_WRITE:
default:
  // Не умеем выполнять другие запросы:
  request_header.rq_result_code =
            COMMAND_NOT_SUPPORTED;
  receive(OS_port, 0, request_header.rq_data_size);
  send(OS_port,
```

```
        &request_header,
        sizeof(request_header));
  break;
} // end case
      continue;
    } // end select
  } // end while
}

void init_driver() {
  set_interrupt_handler(KEYBOARD, keyboard_handler);
  create_port(internal_port);

  last_char = 0;
  first_char = 1023;
  count = 0;

  process_start(driver_body);
}
```

Приведенный пример также считывает только данные, уже находившиеся в буфере на момент запроса. Впрочем, сейчас уже относительно легко реализовать чтение по строкам: начав обработку запроса чтения, мы можем остановиться и подождать новых символов, используя операцию `receive(internal_port, ...)`, пока не встретим конец строки. Или же мы можем возложить разбиение потока данных на строки на программу постобработки, как это сделано в системах семейства Unix.

Здесь возникает интересный вопрос: обязана ли пользовательская программа ожидать окончания операции? Вообще говоря, не обязана, но этот вопрос подробнее будет обсуждаться в п. 7.3.5.

Предполагается, что микроядро обменивается с драйвером сообщениями, состоящими из заголовка стандартного формата и, возможно, блока данных переменной длины. Заголовок содержит код запроса: чтение, запись и т.д., количество сопутствующих данных и поле, в котором драйвер возвращает код результата операции.

В нашем примере приводится только код самого драйвера. Читателю предлагается самостоятельно представить себе код программ пред- и постобработки.

Любопытно, что в транспьютере подобный драйвер был бы еще проще: мы избавились бы от обработчика прерывания. Вместо этого мы просто считывали бы данные из физического линка операцией `get`.

Основанная на микроядре асинхронная модель драйвера решает проблему реентерабельности. Основным недостатком данной модели являются относительно большие накладные расходы – притивы `send/receive` нашего микроядра осуществляют копирование данных; кроме того, эти примитивы сопровождаются бо́льшими накладными расходами, чем прямые вызовы в синхронной модели. Однако, как показывает практика QNX и VxWorks, в действительности эти расходы невелики (например, в монолитных системах семейства Unix тоже происходит копирование данных в системные буфера) и оптимально реализованное микроядеро не уступает по производительности монолитным системам или даже превосходит их.

7.3.4 Асинхронный ввод/вывод в системах с монолитным ядром

Ряд систем с монолитным ядром также реализуют асинхронный ввод/вывод, используя вместо прямого вызова функций драйвера установку запроса в очередь. Пожалуй, наиболее поучительна в этом смысле структура драйвера в операционной системе VAX/VMS.

С точки зрения планировщика драйвер представляет собой процесс с укороченным контекстом, так называемый **fork-процесс**. Укорочение заключается в том, что драйвер может работать только с одним банком виртуальной памяти из трех; таким образом, при переключении контекста задействуется меньше регистров диспетчера памяти. Fork-процесс имеет более высокий приоритет, чем все пользовательские процессы, и может быть вытеснен только более приоритетным fork-процессом. Вместо обычного дескриптора процесса (`PCB` – Process Control Block) используется `UCB` – Unit Control Block, блок управления устройством.

Драйвер получает запросы на ввод/вывод через очередь запросов. Элемент очереди называется `IRP` (**Input[-Output] Request Packet** – пакет запроса ввода/вывода). Обработав первый запрос в очереди, драйвер начинает обработку следующего. Операции над очередью запросов выполняются специальными командами процессора VAX и являются атомарными. Если очередь пуста, fork-процесс завершается. При появлении новых запросов система вновь создаст fork-процесс.

`IRP` содержит: код операции (чтение, запись или код `SPFUN`[3]); адрес блока данных, которые должны быть записаны, или буфера, куда данные необходимо поместить; информацию, используемую при постобработке, такую как идентификатор процесса, запросившего операцию.

В зависимости от кода операции драйвер запускает соответствующую подпрограмму. В VAX/VMS адрес подпрограммы выбирается из таблицы `FDT`: Function Definition Table. Подпрограмма инициирует операцию и приостанавливает fork-процесс, тем самым возвращая управление системе. Затем, когда происходит прерывание, его обработчик возобновляет исполнение fork-процесса. Как и в примере 12, приостановка и возобновление исполнения fork-процесса выполняются соответствующими сервисами ядра.

В некоторых случаях обработчику прерывания приходится решать более сложную задачу: определять, какой из fork-процессов необходимо активизировать. Например, терминальный мультиплексор обслуживает несколько (некоторые модели – до 64) последовательных каналов, но имеет только одно прерывание. С каждым из каналов ассоциирован свой `UCB` и соответственно свой fork-процесс. Обработчик прерывания должен определить, какой из каналов вызвал событие, и активизировать процесс, связанный с этим каналом.

---

[3]Специальная функция, подобная `ioctl` в системах семейства Unix.

Окончив обработку запроса, fork-процесс драйвера вызывает AST, сигнализируя программе постобработки об окончании запроса.

Более подробно организация подсистемы ввода/вывода в VAX/VMS описана в работах [11] и [15].

Аналогичным образом устроены драйверы в OS/2. Даже элемент очереди запросов там называется IRP. Точнее, модель драйвера в OS/2 похожа на наше описание работы драйвера в VAX/VMS, но структура самого драйвера отличается. Драйвер в VAX/VMS состоит из набора таблиц фиксированной структуры, задающих точки входа в подпрограммы обработки, и самих этих подпрограмм. Напротив, драйвер в OS/2 представляет собой загрузочный модуль с единственным обязательным элементом: заголовком драйвера, который задает точки входа инициализационной программы драйвера и программы fork-процесса, которая называется **стратегической функцией устройства** (**device strategy function**).

Термин **device strategy function** унаследован из систем семейства Unix. Хотя во многих системах этого семейства используется полностью синхронная модель работы с последовательными устройствами, блочные устройства с самого начала использовали асинхронную модель, основанную на очереди запросов. Запросы обрабатываются стратегической функцией, которая представляет собой кооперативный процесс. Так же, как fork-процесс в VAX/VMS, стратегическая функция завершается при опустошении очереди запросов. При появлении новых запросов ядро перезапускает ее. Буферизация запросов и формирование очереди осуществляется специальным модулем системы, который называется **дисковым кэшем**. Принцип работы дискового кэша будет обсуждаться в п. 7.3.6.

В Unix System V существует модуль STREAMS, который осуществляет буферизацию и "асинхронизацию" запросов к последовательным устройствам. Драйверы, рассчитанные на работу с этим модулем, также предоставляют стратегическую функцию вместо непосредственно вызываемых функций `read/write`. Unix System V Release 4 имеет микроядерную архитектуру, поэтому в ней все драйверы являются асинхронными.

Модель драйвера, основанную на форк-процессах или стратегических функциях можно считать переходной между монолитным ядром и микроядром, а операции над очередью запросов можно считать реализацией примитивов `send/receive`. Поэтому можно говорить о непрерывном спектре более или менее монолитных (или более или менее микроядерных) архитектур ядра, с полностью микроядерными системами (напр., QNX) на одном конце спектра и полностью монолитными – на другом. При этом даже монолитный "классический" Unix будет иметь некоторые микроядерные черты; полностью монолитной можно считать, разве что MS DOS, да и то при отсутствии асинхронного дискового кэша.

7.3.5 Асинхронная модель ввода/вывода с точки зрения приложений

В п. 7.3.3, обсуждая асинхронную модель драйвера, мы задались вопросом: должна ли прикладная программа, сформировав запрос на ввод/вывод, дожидаться его завершения? Ведь система, приняв запрос, передает его асинхронному драйверу, который инициирует операцию на внешнем устройстве и освобождает процессор. Сама система не ожидает завершения запроса. Так должна ли программа ожидать его?

Если было запрошено чтение данных, то ответ, на первый взгляд, очевиден: должна. Ведь если данные запрошены, значит они сейчас будут нужны программе.

Однако программа может выделить буфер для данных, запросить чтение, потом некоторое время заниматься чем-то полезным, но не относящимся к запросу, и лишь в точке, когда данные действительно будут нужны, спросить систему: а готовы ли данные? Если готовы, то можно продолжать работу. Если нет, то придется ждать.

Во многих приложениях, особенно интерактивных, такое асинхронное чтение оказывается единственно приемлемым вариантом, поскольку оно позволяет задаче одновременно осуществлять обмен с несколькими источниками данных и таким образом повысить пропускную способность и/или улучшить время реакции на запрос пользователя.

Напротив, при записи, казалось бы, нет необходимости дожидаться физического завершения операции. При этом мы получаем режим, известный как **отложенная запись** (**lazy write** – "ленивая" запись, если переводить дословно). Однако такой режим создает две специфические проблемы.

Во-первых, программа должна знать, когда ей можно использовать буфер с данными для других целей.

Если система копирует записываемые данные из пользовательского адресного пространства в системное, то эта же проблема возникает внутри ядра; внутри ядра проблема решается использованием многобуферной схемы, и все относительно просто. Однако копирование приводит к дополнительным затратам времени и требует выделения памяти под буфера. Наиболее остро эта проблема встает при работе с дисковыми и сетевыми устройствами, с которыми система обменивается большими объемами данных. Проблема управления дисковыми буферами подробнее обсуждается в 7.3.6.

В большинстве современных вычислительных систем общего назначения накладные расходы, обусловленные буферизацией запросов, относительно невелики или по крайней мере считаются приемлемыми. Но в системах реального времени и/или встраиваемых контроллерах, где время и объем оперативной памяти жестко ограничены, эти расходы оказываются серьезным фактором.

Если же вместо системных буферов используется отображение данных в системное адресное пространство (системы с открытой памятью можно считать вырожденным случаем такого отображения), то ситуация усложняется. Программа должна иметь возможность узнать о физическом окончании записи, потому что только после этого буфер действительно свободен. Фактически, программа должна самостоятельно реализовать многобуферную схему или искать другие выходы.

Во-вторых, программа должна дождаться окончания операции, чтобы узнать, успешно ли она закончилась.

Часть ошибок, например, попытку записи на устройство, физически не способное выполнить такую операцию, можно отловить еще во время предобработки, однако аппаратные проблемы могут быть обнаружены только на фазе исполнения запроса.

Многие системы, реализующие отложенную запись, при обнаружении аппаратной ошибки просто устанавливают флаг ошибки в блоке управления устройством. Программа предобработки, обнаружив этот флаг, отказывается исполнять следующий запрос. Таким образом, прикладная программа считает ошибочно завершившийся запрос успешно выполнившимся и обнаруживает ошибку лишь при одной из следующих попыток записи, что не совсем правильно.

Иногда эту проблему можно игнорировать. Например, если программа встречает одну ошибку при записи, то все исполнение программы считается неуспешным и на этом заканчивается. Однако во многих случаях, например в задачах управления промышленным или исследовательским оборудованием, программе необходимо знать результат завершения операции, поэтому простая отложенная запись оказывается совершенно неприемлемой.

Так или иначе ОС, реализующая асинхронное исполнение запросов ввода/вывода, должна иметь средства сообщить пользовательской программе о физическом окончании операции и результате этой операции.

Например, в системах RSX-11 и VAX/VMS фирмы DEC для синхронизации используется **флаг локального события** (**local event flag**). Как говорилось в п. 4.3.2, флаг события в этих системах представляют собой аналог двоичных семафоров Дийкстры, но с ним также может быть ассоциирована процедура `AST`.

Системный вызов ввода/вывода в этих ОС называется `QIO` (Queue Input/Output [Request] – установить в очередь запрос ввода/вывода) и имеет две формы: асинхронную `QIO` и синхронную `QIOW` (Queue Input/Output and Wait – установить запрос и ждать [завершения]). С точки зрения подсистемы ввода/вывода эти вызовы ничем не отличаются, просто при запросе `QIO` ожидание конца запроса выполняется пользовательской программой "вручную", а при `QIOW` выделение флага события и ожидание его установки делается системными процедурами пред- и постобработки.

В ряде систем реального времени, например, в OS-9 и RT-11, используются аналогичные механизмы.

Напротив, большинство современных ОС общего назначения не связываются с асинхронными вызовами и предоставляют прикладной программе чисто синхронный интерфейс, тем самым вынуждая ее ожидать конца операции.

Возможно, это объясняется идейным влиянием ОС Unix. Набор операций ввода/вывода, реализованных в этой ОС, стал общепризнанным стандартом де-факто и основой для нескольких официальных стандартов. Например, набор операций ввода/вывода в MS DOS является прямой копией Unix; кроме того, эти операции входят в стандарт ANSI на системные библиотеки языка C и стандарт POSIX.

Некоторые системы, например Unix System V R4, разрешают программисту выбирать между отложенной записью по принципу **Fire And Forget** (**выстрелил и забыл**) и полностью синхронной, выбирая между этими режимами вызовом `fcntl` с соответствующим кодом операции.

Разработчики приложений для OS/2 и Win32 (Windows NT и Windows 95) часто сталкиваются с необходимостью асинхронного исполнения запросов ввода/вывода. Разработчикам предлагается самостоятельно имитировать асинхронный обмен, создавая для каждого асинхронно исполняемого запроса свой процесс (**нить** (**thread**) в принятой в этих системах терминологии). Обычно нить, исполняющая запрос, тем или иным способом сообщает основной нити о завершении операции. Для этого чаще всего используются штатные средства межнитевого взаимодействия – семафоры и др.

Грамотное использование нитей позволяет создавать интерактивные приложения с очень высоким субъективным временем реакции, но за это приходится платить усложнением логики программы.

Любопытно, что “нитевая” оптимизация неинтерактивных приложений, ориентированных на интенсивный ввод/вывод (серверов транзакций и др.) не приводит к столь впечатляющим результатам: тесты показывают, что приложения такого типа для систем семейства Unix оказываются быстрее или, во всяком случае, не медленнее, чем эквивалентные приложения для OS/2 или NT, несмотря на то, что большинство систем семейства Unix не имеют ничего даже отдаленно похожего на нити или асинхронный ввод/вывод.

Как OS/2, так и Win32 не предоставляют системных вызовов для асинхронного ввода/вывода, что несколько удивительно, так как обмен данными с драйвером OS/2 и Windows NT происходит асинхронно, путем установки запроса в очередь.

В Windows NT существует средство организации асинхронного ввода/вывода – так называемые **порты завершения [операции]** (**completion ports**). Однако это средство не поддерживается в Windows 95, возможно из-за синхронной организации ввода/вывода в этой системе, поэтому большинство разработчиков избегают использования портов завершения.

Синхронная модель ввода/вывода проста в реализации и использовании и, как показал опыт систем семейства Unix и его идейных наследников[4], вполне адекватна большинству приложений общего назначения. Однако, как уже было показано, она не очень удобна (а иногда и просто непригодна) для задач реального времени.

7.3.6 Дисковый кэш

Функции и принципы работы дискового кэша существенно отличаются от общих алгоритмов кэширования, обсуждавшихся в п. 3.6. Дело в том, что характер обращения к файлам обычно существенно отличается от обращений к областям кода и данных задачи. Например, компилятор C и макропроцессор

[4] MS DOS, как уже говорилось, использует прямую копию модели Unix, а OS/2 и Windows NT во многих отношениях являются наследниками MS DOS, в том числе и по используемой модели ввода/вывода.

TEX рассматривают входные и выходные файлы как потоки данных. Входные файлы прочитываются строго последовательно и полностью, от начала до конца. Аналогично выходные файлы полностью перезаписываются, и перезапись тоже происходит строго последовательно. Попытка выделить аналог рабочей области при таком характере обращений обречена на провал независимо от алгоритма, разве что рабочей областью будут считаться все входные и выходные файлы.

Тем не менее кэширование или, точнее, буферизация данных при работе с диском имеет смысл и во многих случаях может приводить к значительному повышению производительности системы. Если отсортировать механизмы повышения производительности в порядке их важности, мы получим следующий список:

1. Размещение в памяти структур файловой системы – каталогов, `FAT` или таблицы инодов[5] и т.д. Это основной источник повышения производительности при использовании дисковых кэшей под MS/DR DOS.

2. Отложенная запись. Само по себе откладывание записи не повышает скорости обмена с диском, но позволяет более равномерно распределить загрузку канала.

3. Группировка запросов на запись. Система имеет пул буферов отложенной записи, который и называется дисковым кэшем. При поступлении запроса на запись, система выделяет буфер из этого пула и ставит его в очередь к драйверу. Если за время нахождения буфера в очереди в то же место на диске будет произведена еще одна запись, система может дописать данные в имеющийся буфер вместо установки в очередь второго запроса. Это значительно повышает скорость, если запись происходит массивами, не кратными размеру физического блока на диске.

4. Собственно кэширование. После того как драйвер выполнил запрос, буфер не сразу используется повторно, поэтому какое-то время он содержит копию записанных или прочитанных данных. Если за это время произойдет обращение на чтение соответствующей области диска, система может отдать содержимое буфера вместо физического чтения.

5. Опережающее считывание. При последовательном обращении к данным чтение из какого-либо блока значительно повышает вероятность того, что следующий блок также будет считан. Теоретически опережающее чтение должно иметь тот же эффект, что и отложенная запись, т.е. обеспечивать более равномерную загрузку дискового канала и его работу параллельно с центральным процессором. На практике, однако, часто оказывается, что считанный с опережением блок оказывается никому не нужен, поэтому эффективность такого чтения заметно ниже, чем у отложенной записи.

6. Сортировка запросов по номеру блока на диске. По идее, такая сортировка должна приводить к уменьшению времени позиционирования головок чтения/записи. Кроме того, если очередь за-

[5]Эти понятия подробнее обсуждаются в гл. 8.

просов будет отсортирована, это облегчит работу алгоритмам кэширования, которые производят поиск буферов по номеру блока. Во всяком случае вреда от сортировки нет, а польза теоретически возможна, поэтому многие системы выполняют такую сортировку.

В работе [25] подробно описываются различные подходы к сортировке запросов. Авторы не имеют данных, которые подтверждали бы повышение производительности за счет сортировки на современных дисковых устройствах.

Кэширование значительно повышает производительность дисковой подсистемы, но создает ряд проблем, причем некоторые из них довольно неприятного свойства.

Первая из проблем – та же, что и у отложенной записи вообще. При использовании отложенной записи программа не знает, успешно ли завершилась физическая запись. Однако многие современные файловые системы поддерживают так называемый **hotfixing** (**горячую починку**) – механизм, обеспечивающий динамическую замену “плохих” логических блоков на “хорошие”, что в значительной мере компенсирует эту проблему.

Вторая проблема гораздо серьезнее и тоже свойственна всем механизмам отложенной записи: если в промежутке между запросом и физической записью произойдет сбой всей системы, то данные будут потеряны. Например, пользователь сохраняет отредактированный файл и, не дождавшись окончания физической записи, выключает питание – содержимое файла оказывается потеряно или повреждено. Другая ситуация, до боли знакомая всем пользователям DOS/Windows 3.x/Windows 95: пользователь сохраняет файл, и в это время система зависает – результат тот же. Аналогичного результата можно достичь, не вовремя достав дискету или другой удаляемый носитель из дисковода.

Очень забавно наблюдать как пользователь, хотя бы раз имевший неприятный опыт общения с дисковым кэшем `SMARTDRV`, копирует данные с чужого компьютера на дискету. Перед тем, как извлечь ее из дисковода, он оглядывается на хозяина машины и с опаской спрашивает: “У тебя там никаких кэшей нет?”. Авторам доводилось наблюдать такое поведение у нескольких десятков людей.

Если откладывается запись не только пользовательских данных, но и модифицированных структур файловой системы, ситуация еще хуже: системный сбой может привести не только к потере данных, находившихся в кэше, но и к разрушению файловой системы, т.е., в худшем случае к потере всех данных на диске.

Методы обеспечения целостности данных при системном сбое подробнее обсуждаются в п. 8.4. Находившиеся в кэше данные при фатальном сбое гибнут всегда, но существуют способы избежать повреждения системных структур данных на диске без отказа от использования отложенной записи.

Третья проблема, связанная с дисковым кэшем – это выделение памяти под него. Уменьшение кэша приводит к снижению производительности дисковой подсистемы, увеличение же кэша отнимает память у пользовательских программ. В системах с виртуальной памятью это может привести к увеличению дисковой активности за счет увеличения объема подкачки, что ведет к снижению как дисковой, так и общей производительности системы. Перед администратором системы встает нетривиальная задача: найти точку оптимума. Положение этой точки зависит от:

- объема физической памяти;
- скорости канала обмена с диском, а также от того, работает этот канал с использованием центрального процессора, как IDE-контроллеры у IBM PC или же по ПДП;
- скорости центрального процессора;
- суммы рабочих наборов программ, исполняющихся в системе;
- интенсивности и характера обращений к диску.

При этом зависимость количества страничных отказов от объема памяти, доступной приложениям, имеет существенно нелинейный вид. Это же утверждение справедливо для связи между размером дискового кэша и соответствующей экономией обращений к диску. Таким образом, задача подбора оптимального размера кэша является задачей нелинейной оптимизации. Самым неприятным является то, что ключевой исходный параметр – *характер* обращений к диску – является не количественным, а качественным; точнее сказать, его можно измерить лишь при помощи очень большого числа независимых количественных параметров.

Во многих ситуациях невозможно теоретически оценить положение оптимальной точки, и единственным способом оказывается эксперимент: прогон типичной для данной машины смеси заданий при различных объемах кэша. При этом нужно иметь возможность различать дисковую активность, связанную с обращениями к файлам и со страничным обменом. Большинство современных ОС предоставляют для этой цели различниые инструменты системного мониторинга. Чаще, однако, объем кэша выставляется на глаз, а к дополнительной настройке прибегают только если производительность оказывается слишком низкой.

Возникает вполне естественное желание возложить подбор размера кэша на саму систему, т.е. менять размер кэша динамически в зависимости от рабочей нагрузки. Кроме упрощения работы администратора, такое решение имеет еще одно большое преимущество: система начинает “автомагически” подстраиваться под изменения нагрузки.

Но далеко не все так просто. Если объем памяти в системе превосходит потребности прикладных программ, то динамический дисковый кэш может формироваться по очень простому “остаточному” принципу – все, что не пригодилось приложениям, отдается под кэш. Однако оперативная память до сих пор относительно доро́га и представляет собой дефицитный ресурс, поэтому наибольший практический интерес представляет ситуация, когда памяти не хватает даже приложениям, не говоря уже о кэше. Тем не менее кэш обычно нужен.

Разумной политикой была бы подстройка кэша в зависимости от количества страничных отказов: если число отказов становится слишком большим, система уменьшает кэш; если же число отказов мало́, а идут интенсивные обращения к диску, система увеличивает кэш. Получается саморегулирующаяся система с отрицательной обратной связью. Однако, если вдуматься, то видно, что вместо одной произвольной переменной (объема статического кэша) мы вынуждены ввести как минимум три:

- количество страничных отказов, которое считается слишком большим;
- количество отказов, которое считается достаточно малым;
- величину, на которую следует увеличить или уменьшить кэш в этих случаях.

Оптимальные значения этих переменных зависят практически от тех же самых параметров, что и объем статического кэша, но подбор значений экспериментальным путем оказывается значительно сложнее, потому что вместо одномерной нелинейной оптимизации мы вынуждены заниматься трехмерной нелинейной оптимизацией, что несоизмеримо сложнее.

Кроме того, читатель, знакомый с теорией управления, должен знать, что неудачный подбор параметров у системы с отрицательной обратной связью может приводить к колебательному процессу вместо саморегуляции. В дискуссиях USENET news приводились примеры развития таких колебаний в динамическом кэше системы Windows NT при компиляции большого проекта в условиях недостатка памяти.

Вполне возможно, что низкая производительность Windows NT на системах с "небольшим" (8 - 16 Мб) количеством памяти объясняется вовсе не низким качеством реализации и даже не секретным сговором между фирмой MicroSoft и производителями оперативной памяти, а просто плохо сбалансированным динамическим кэшем.

7.3.7 Спулинг

Гигабайт тебе в спул.

Популярное ругательство

Термин **спулинг** (**spooling**) не имеет общепринятого русского аналога. В соответствии с программистским фольклором, слово это происходит от аббревиатуры **Simultaneous Peripherial Operation Off Line**. Эту фразу трудно дословно перевести на русский язык; имеется в виду метод работы с внешними устройствами вывода (реже – ввода) в многозадачной ОС или многомашинной среде, при которой задачам создается иллюзия одновременного доступа к устройству. При этом, однако, задачи не получают к устройству прямого доступа, а работают в режиме **off-line** (без прямого подключения). Выводимые данные накапливаются системой, а затем выводятся на устройство так, чтобы вывод различных задач не смешивался.

Видно, что этот метод работы отчасти напоминает простую отложенную запись, но основная задача здесь не только и не столько повышение производительности, сколько разделение доступа к медленному внешнему устройству. Чаще всего спулинг используется для работы с печатающими устройствами, а для промежуточного хранения данных используется диск.

Многие почтовые системы используют механизм, аналогичный спулингу: если получатель не готов принять письмо или линия связи с получателем занята либо вообще разорвана, предназначенное к отправке письмо помещается в очередь. Затем, когда соединение будет установлено, письмо отправляется.

Классический спулинг реализован в ОС семейства Unix. В этих ОС вывод задания на печать осуществляется командой `lpr`. Эта команда копирует предназначенные для печати данные в каталог `/usr/spool/lp`, возможно, пропуская их при этом через программу-фильтр. Каждая порция данных помещается в отдельный файл. Имена файлов генерируются так, что имя каждого вновь созданного файла было “больше” предыдущего при сравнении ASCII-кодов. За счет этого файлы образуют очередь.

Системный процесс-**демон** (**daemon**) `lpd` (или `lpshed` в Unix SVR4) периодически просматривает каталог. Если там что-то появилось, а печатающее устройство свободно, демон копирует появившийся файл на устройство. По окончании копирования он удаляет файл, тем или иным способом уведомляет пользователя об окончании операции (в системах семейства Unix чаще всего используется электронная почта) и вновь просматривает каталог. Если там по-прежнему что-то есть, демон выбирает первый по порядку запрос и также копирует его на устройство.

Тот же механизм используется почтовой системой Unix – программой `sendmail`, только вместо каталога `/usr/spool/lp` используется `/usr/spool/mail`.

Этот механизм очень прост, но имеет один специфический недостаток: демон не может непосредственно ожидать появления файлов в каталоге, как можно было бы ожидать установки семафора или другого флага синхронизации. Если бы демон непрерывно сканировал каталог, это создавало бы слишком большую и бесполезную нагрузку на систему. Поэтому демон пробуждается через фиксированные интервалы времени; если за это время ничего в очереди не появилось, демон засыпает вновь. Такой подход также очень прост, но увеличивает время прохождения запросов: запрос начинает исполняться не сразу же после установки, а лишь после того, как демон в очередной раз проснется.

В OS/2 и Windows NT спулинг организован отчасти похожим образом с той разницей, что установка запроса в очередь может происходить не только командой `PRINT`, но и простым копированием данных на псевдоустройство `LPT[1-9]`. В отличие от систем семейства Unix как программа `PRINT`, так и псевдоустройства портов активизируют процесс спулера непосредственно при установке запроса. Графические драйверы печатающих устройств в этих системах также используют спул вместо прямого обращения к физическому порту.

Novell Netware предоставляет специальный механизм для организации спулинга – очереди запросов. Для хранения данных в этих очередях также используется диск, но прикладные программы вместо просмотра каталога могут пользоваться системными функциями `GetNextMessage` и `PutMessage`. Вызов `GetNextMessage` блокируется, если очередь пуста; таким образом, нет необходимости ожидать пробуждения демона или специальным образом активизировать его – демон сам пробуждается при появлении запроса. Любопытно, что почтовая система Mercury Mail для Novell Netware может использовать для промежуточного хранения почты как очередь запросов, так и выделенный каталог в зависимости от конфигурации.

Глава 8

Файловые системы

Одним из первых внешних устройств после клавиатуры и телевизора, которые перечисляются в любом руководстве по персональным компьютерам для начинающих, является магнитный диск. Вообще говоря, вместо магнитного диска в наше время может использоваться и какая-то другая энергонезависимая память, например, EEPROM или файловый сервер, но наличие такой памяти является очень важным.

Ведь вы же не будете набирать вашу программу каждый раз при новом включении компьютера. Правда, на PDP-11 такое еще было возможным; в НГУ даже ходят легенды о людях, которые набирали с консоли в пультовом режиме тетрис, и играли в него, пока сетевой файловый сервер висел. А если программа – это пакет типа `CorelDraw!`, исходный текст которого занимает десятки тысяч строк, а бинарный загрузочный модуль – полтора мегабайта?

Понятно также, что недостаточно иметь возможность просто запомнить программу и данные. Ведь вы можете работать с несколькими программами, или над несколькими проектами одновременно. Ясно, что записывать на бумажке, в какое место вашей энергонезависимой памяти вы что-то сохранили, по меньшей мере неудобно. Поэтому естественно желание создать специализированную программу, которая будет как-то структурировать сохраненные данные.

Именно эту работу по структурированию пользовательских данных берет на себя модуль ОС, называемый **файловым менеджером**. Дисковые операционные системы (ДОС) состоят по сути только из файлового менеджера и загрузчика бинарных модулей.

Понятие файла вводится в самом начале любого курса компьютерного ликбеза, но мало в каком курсе дается внятное определение этого понятия. Слово файл (file) дословно переводится с английского как папка или подшивка, но такой перевод почти не добавляет ясности. Одно из наилучших определений, известных авторам, звучит так:

Файл – это совокупность данных, доступ к которой осуществляется по ее имени.

Файл, таким образом, противопоставляется другим объектам, доступ к которым осуществляется по их адресу, например, записям внутри файла или блокам на диске.

ОС семейства Unix трактуют понятие файла более широко – там файлом называется любой объект, имеющий имя в файловой системе. Однако файлы, не являющиеся совокупностями данных (каталоги, внешние устройства, псевдоустройства, именованные программные каналы, семафоры Xenix), часто называют не простыми файлами, а "специальными".

Понятно, что магнитный диск или другое устройство памяти (кроме, пожалуй, файлового сервера) чаще всего организует доступ к данным не по их именам, а все-таки по адресам, например, по номеру сектора, дорожки и поверхности диска. Поэтому, если система хочет предоставлять доступ по именам, она должна хранить таблицу преобразования имен в адреса – **директорию** (**directory**), или, как чаще говорят по-русски, **каталог**.

В каталоге хранится имя файла и другая информация о файле, такая, как его размер и местоположение на диске. Как правило, хранят также дату создания файла, дату его последней модификации, а в многопользовательских системах – идентификатор хозяина этого файла и права доступа к нему для других пользователей. Во многих файловых системах эта информация хранится не в самом каталоге, а в специальной структуре данных – **иноде**, **метафайле** и т.д. В этом случае запись в каталоге содержит только имя и указатель на управляющую структуру файла.

Большинство современных операционных систем позволяет делать вложенные каталоги – файлы, которые сами являются каталогами. В таких системах файл задается полным или **путевы́м** именем (**path name**), состоящим из цепочки имен вложенных каталогов и имени файла в последней из них.

Совокупность каталогов, а также системных структур данных, отслеживающих размещение файлов на диске и свободное дисковое пространство, называется **файловой системой** (**ФС**). Иногда на диске размещается только одна файловая система. Современные ОС часто позволяют размещать на одном физическом диске несколько файловых систем, выделяя каждой из них фиксированную часть диска. Такие части диска называются **разделами** (**partition**) или **срезами** (**slice**).

8.1 Файлы с точки зрения пользователя

8.1.1 Формат имен файлов

В различных ФС допустимое имя файла может иметь различную длину, и в нем могут использоваться различные наборы символов. Так, в RT-11 и RSX-11 имена файлов состоят из символов кодировки RADIX-50 и имеют длину 9 символов: 6 символов – собственно имя, а 3 – расширение. При этом имя имеет вид `"XXXXXX.XXX"`, но символ `'.'` не является частью имени – это просто знак препинания. Предполагается, что расширение должно соответствовать типу данных, хранящихся в файле: `.SAV` будет именем абсолютного загружаемого модуля, `.FOR` – программы на Фортране, `.CRH` – "файлом информации о системном крахе", как было написано в одном переводе руководства (попросту говоря, это посмертная выдача ОС, по которой можно попытаться понять причину аварии).

В CP/M и ее потомках MS DOS-DR DOS, а также в VMS имена файлов хранятся в 8-битной ASCII-кодировке, но почему-то разрешено использование только букв верхнего регистра, цифр и некоторых печатаемых символов. При этом в системах линии CP/M имя файла имеет 8 символов + 3 символа расширения, а в VMS как имя, так и расширение могут содержать более 32 символов.

Любопытно, что MS/DR DOS при поиске в каталоге переводят в верхний регистр имя, заданное пользователем, но оставляют как есть имя, считанное из каталога. Строго говоря, это ошибка: если мы создадим имя файла, содержащее буквы нижнего регистра, то ни одна программа не сможет открыть или переименовать такой файл.

Одному из авторов довелось столкнуться с такой проблемой при попытке прочитать дискету, записанную ОС TC (Экспериментальная UNIXоподобная ОС для Паскаль-машины N9000). Проблему удалось решить только при помощи шестнадцатиричного дискового редактора прямым редактированием имен в каталогах. Возможно, существует и более элегантное решение, но авторам не удалось его найти. Использовать конструкцию `*.*` бесполезно, потому что, в действительности, операции над файлами, заданными таким образом, состоят из двух операций: `FindFirst/FindNext`, возвращающей [следующее] имя файла, соответствующее шаблону, и `Open`. `FindFirst/FindNext` возвращает недопустимое имя файла и `Open` не может использовать его. Программа `CHKDSK` не возражает против имен файлов в нижнем регистре. Строго говоря, это тоже ошибка. Все остальные способы так или иначе сводятся к прямой (в обход ДОС) модификации ФС.

Кроме того, любопытных эффектов можно достичь, попытавшись создать файл с именем, содержащим русские буквы.

Наибольшим либерализмом в смысле имен отличаются ОС семейства Unix, в которых имя файла может состоять из *любых* символов кодировки ASCII, кроме символов `'\000'` и `'/'`, например, из восьми символов перевода каретки. При этом `'\000'` является ограничителем имени, а `'/'` – разделителем между именем каталога и именем файла. Никакого разделения на имя и расширение нет, и хотя имена файлов с программой на языке C заканчиваются `".c"`, а объектных модулей – `".o"`, точка здесь является частью имени. Вы можете создать файл с именем `"gcc-2.5.8.tar.gz"`. В UNIX SVR3 длина имени файла ограничена 14 символами, а в BSD UNIX, Linux и SVR4 – только длиной блока на диске, т.е. 512 байтами или более. При этом нулевой символ считается концом имени в каталоге.

Возможность использовать в именах неалфавитные символы типа перевода каретки или `ASCII EOT` кажется опасным излишеством. На самом деле:

1. Это не излишество а, скорее, упрощение – из процедур, работающих с именами, удалена проверка символа на “допустимость”.
2. Оно не столь уж опасно: такой файл всегда можно переименовать. В некоторых случаях процесс набора имени файла в командной строке превращается в нетривиальное упражнение, потому что shell (командный процессор) рассматривает многие неалфавитные символы как команды. Но надо отметить, что, правильно используя кавычки и символ `'\'`, пользователь может передать команде аргумент, содержащий *любые* символы ASCII, кроме `\000"`.

В последнее время среди ОС стало модным поддерживать длинные имена файлов. Отчасти это, возможно, связано с тем, что производители ПО для персональных компьютеров осознали, что системы семейства Unix являются потенциально опасными конкурентами, а длинные имена файлов традиционно считаются одним из преимуществ этого семейства.

Например, OS/2, использующая файловую систему HPFS (High Performance File System – высокопроизводительная файловая система[1]), поддерживает имена файлов длиной до 256 символов, содержащие печатаемые символы и пробелы. Точка считается частью имени, как и в UNIX, и можно создавать имена, содержащие несколько точек.

При этом нужно отметить, что система при поиске файла приводит к одному регистру все алфавитные символы в имени. С одной стороны, это означает дополнительное удобство для пользователя – при наборе имени не нужно заботиться о регистре букв, с другой – пользователь не может создать в одном каталоге файлы `"text.txt"` и `"Text.txt"`. Из-за этого, например, нельзя использовать принятое в UNIX соглашение о том, что файл на языке C имеет расширение `".c"`, а на языке C++ – `".C"`.

Поддержка длинных имен файлов реализована и в наследнике линии CP/M-MS DOS, операционной системе Windows 95, ранее известной под кодовым именем Chicago.

Некоторые ОС, например, RSX-11 и VMS, поддерживают также номер версии файла. В каталоге может существовать несколько версий файла с одним именем; если номер версии при открытии файла не задается, то открывается последняя версия.

Версии файла очень удобны при разработке любых объектов, от программ или печатных плат до книг: если вам не понравились изменения, внесенные вами в последнюю версию, вы всегда можете откатиться назад.

По ряду исторических причин большинство современных ОС, к сожалению, не поддерживают такую возможность, но подробное обсуждение этих причин увело бы нас далеко от основной темы.

8.1.2 Операции над файлами

Большинство современных ОС рассматривают файл как неструктурированную последовательность байт переменной длины. В стандарте POSIX над файлом определены следующие операции:

- `int open(char * fname, int flags, mode_t mode)` Эта операция "открывает" файл, устанавливая соединение между программой и файлом. При этом программа получает **дескриптор файла** – целое число, идентифицирующее данное соединение. Фактически это индекс в системной таблице открытых файлов для данной задачи. Все остальные операции используют этот индекс для ссылки на файл.

  Параметр `char * fname` задает имя файла. `int flags` – это битовая маска, определяющая режим открытия файла. Файл может быть открыт только на чтение, только на запись и на чтение и запись; кроме того, можно открывать существующий файл, а можно пытаться создать новый

[1] HPFS *действительно* имеет более высокую производительность, чем файловая система MS DOS, называемая FAT; в п. 8.3.1 будет сказано, за счет чего.

файл нулевой длины. Необязательный третий параметр `mode` используется только при создании файла и задает атрибуты этого файла.

- `off_t lseek(int handle, off_t offset, int whence)` Эта операция перемещает указатель чтения/записи в файле. Параметр `offset` задает количество байт, на которое нужно сместить указатель, а параметр `whence` – откуда отсчитывать смещение. Предполагается, что смещение можно отсчитывать от начала файла (`SEEK_SET`), от его конца (`SEEK_END`) и от текущего положения указателя (`SEEK_CUR`). Операция возвращает положение указателя, отсчитываемое от начала файла. Таким образом, вызов `lseek(handle, 0, SEEK_CUR)` возвратит текущее положение указателя, не передвигая его.

- `int read(int handle, char * where, size_t how_much)` Операция чтения из файла. Указатель `where` задает буфер, куда нужно поместить прочитанные данные; третий параметр указывает, сколько данных надо считать. Система считывает требуемое число байт из файла, начиная с указателя чтения/записи в этом файле, и перемещает указатель к концу считанной последовательности. Если файл кончился раньше, считывается столько данных, сколько оставалось до его конца. Операция возвращает количество считанных байт. Если файл открывался только для записи, вызов `read` возвратит ошибку.

- `int write(int handle, char * what, size_t how_much)` Операция записи в файл. Указатель `what` задает начало буфера данных; третий параметр указывает, сколько данных надо записать. Система записывает требуемое число байт в файл, начиная с указателя чтения/записи в этом файле, заменяя хранившиеся на в этом месте данные, и перемещает указатель к концу записанного блока. Если файл кончился раньше, его длина увеличивается. Операция возвращает количество записанных байт.

  Если файл открывался только для чтения, вызов `write` возвратит ошибку.

- `int ioctl(int handle, int cmd, ...);`

- `int fcntl(int handle, int cmd, ...)` Дополнительные операции над файлом. Первоначально, по-видимому, предполагалось, что `ioctl` – это операции над самим файлом, а `fcntl` – это операции над дескриптором открытого файла, но потом историческое развитие несколько перемешало функции этих системных вызовов. Стандарт POSIX определяет некоторые операции как над дескриптором, например дублирование (в результате этой операции мы получаем два дескриптора, связанных с одним и тем же файлом), так и над самим файлом, например, операцию `truncate` – обрезать файл до заданной длины. В большинстве версий Unix операцию `truncate` можно использовать и для вырезания данных из середины файла. При считывании данных из такой вырезанной области считываются нули, а сама эта область не занимает физического места на диске.

  Важной операцией является блокировка участков файла. Стандарт POSIX предлагает для этой цели библиотечную функцию, но в системах семейства Unix эта функция реализована через вызов `fcntl`.

Большинство реализаций стандарта POSIX предлагает и свои дополнительные операции. Так, в Unix SVR4 этими операциями можно устанавливать синхронную или отложенную запись[2] и т.д.

- `caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int handle, off_t offset)` Отображение участка файла в виртуальное адресное пространство процесса. Параметр `prot` задает права доступа к отображенному участку: на чтение, запись и исполнение. Отображение может происходить на заданный виртуальный адрес, или же система может выбирать адрес для отображения сама.

Еще две операции выполняются уже не над файлом, а над его именем: это операции переименования и удаления файла. В некоторых системах, например в системах семейства Unix, файл может иметь несколько имен, и существует только системный вызов для удаления имени. Файл удаляется при удалении последнего имени.

Видно, что набор операций над файлом в этом стандарте очень похож на набор операций над внешним устройством. И то и другое рассматривается как неструктурированный поток байт. Для полноты картины следует сказать, что основное средство межпроцессной коммуникации в системах семейства Unix (**труба**) также представляет собой неструктурированный поток данных. Идея о том, что большинство актов передачи данных может быть сведено к байтовому потоку, довольно стара, но Unix был одной из первых систем, где эта идея была приближена к логическому завершению.

Примерно та же модель работы с файлами принята в CP/M, а набор файловых системных вызовов MS DOS фактически скопирован с вызовов Unix v7. В свою очередь, OS/2 и Windows NT унаследовали принципы работы с файлами непосредственно от MS DOS.

Напротив, в системах, не имеющих Unix в родословной, может использоваться несколько иная трактовка понятия файла. Чаще всего файл трактуется как набор записей. Обычно система поддерживает записи как постоянной длины, так и переменной. Например, текстовый файл интерпретируется как файл с записями переменной длины, а каждой строке текста соответствует одна запись. Такова модель работы с файлами в VMS и в ОС линии OS/360-MVS фирмы IBM.

Практика систем с неструктурированными файлами показала, что хотя структурированные файлы часто бывают удобны для программиста, необязательно встраивать поддержку записей в ядро системы. Это вполне можно сделать и на уровне библиотек. К тому же структурированные файлы сами по себе не решают серьезной проблемы, полностью осознанной лишь в 80 гг. при разработке новых моделей взаимодействия человека с компьютером.

[2]Подробнее понятие отложенной записи обсуждается в п. 7.3.5.

8.1.3 Тип файла

Легко понять, что структурированные файлы предоставляют системе и программисту информацию о структуре хранящихся данных, но не дают никаких сведений о смысле и форме представления этих данных.

Например, с точки зрения системы исходный текст программы на языке C и документ в формате LaTeX совершенно идентичны: и то и другое представляет собой текстовый файл (или, если угодно, файл с записями переменной длины). Однако, если мы попытаемся подать наш документ на вход C-компилятора, мы получим множество синтаксических ошибок и никакого полезного результата.

Этот пример показывает, что во многих случаях оказывается желательно связать с файлом - неважно, структурированный ли это файл или байтовый поток - какую-то метаинформацию: в каком формате хранятся данные, какие операции над ними допустимы, а иногда и сведения о том, кому и зачем эти данные нужны.

По-видимому, наиболее общим решением этой проблемы был бы объектно-ориентированный подход, в котором файл данных рассматривается как объект, а допустимые операции - как методы этого объекта. Ни в одной из известных авторам ОС эта идея в полной мере не реализована, но пользовательские интерфейсы многих современных ОС предоставляют возможность ассоциировать определенные действия с файлами различных типов.

Так, например Explorer - пользовательская оболочка Windows 95 и Windows NT 4.0 - позволяет связать ту или иную программу с файлами, имеющими определенное расширение, например, программу MS Word с файлами, имеющими расширение `.DOC`. Когда пользователь нажимает левой кнопкой мыши на иконке, представляющей такой файл, то автоматически запускается MS Word. Эти же ассоциации доступны и из командной строки - можно напечатать `start Доклад.DOC` и опять-таки запустится MS Word.

Такое связывание очень простó в реализации и реализуется не только в Explorer, но и в простых текстовых оболочках вроде Norton Commander. От ОС при этом требуется только дать возможность каким-то образом различать типы файлов.

Первые попытки ассоциировать с файлом признак типа были сделаны еще в 60 гг. При этом идентификатор типа добавлялся к имени файла в виде короткой, но мнемонической последовательности символов – **расширения** (**extension**). В большинстве современных ОС расширение отделяется от имени символом `'.'`, но проследить истоки этой традиции авторам не удалось. При этом, например, файлы на языке C будут иметь расширение `".c"`, на C++ – `".C"`, а документы в формате LaTeX – `".tex"`.

В ОС семейства Unix имя файла может содержать несколько символов `'.'`, и, таким образом, файл может иметь несколько каскадированных расширений. Например, файл `"main.C"` – это программа на языке C++; `"main.C.gz"` – это программа на языке C++, упакованная архиватором GNU Zip с целью сэкономить место; `"main.C.gz.crypt"` – это программа, которую упаковали и потом зашифровали, чтобы никто посторонний не смог ее прочитать; наконец, "main.C.gz.crypt.uue" – это упакованная и зашифрованная программа, преобразованная в последовательность печатаемых символов кода ASCII, например, для пересылки по электронной почте.

В принципе расширения являются вполне приемлемым и во многих отношениях даже очень удобным способом идентификации типа файла. Одно из удобств состоит в том, что для использования этого метода не нужно никаких или почти никаких усилий со стороны ОС: просто программы договариваются интерпретировать имя файла определенным образом.

Например, стандартный **драйвер компилятора** в системах семейства Unix – программа cc – определяет тип файла именно по расширению. Командная строка

Пример 14

```
cc main.C c-code.c asm-code.s obj-code.o\
    library.a -o program
```

будет интерпретироваться следующим образом:

main.C: текст на языке C++. Его нужно пропустить через препроцессор и откомпилировать компилятором C++, а затем передать то, что получится, редактору связей. Большинство компиляторов в Unix генерируют код на ассемблере, то есть вывод компилятора еще нужно пропустить через ассемблер.

c-code.c: текст на языке C. Он обрабатывается, так же, как и C++-программа, только вместо компилятора C++ используется обычный компилятор C.

asm-code.s: программа на языке ассемблера. Её нужно обработать ассемблером и получить объектный модуль.

obj-code.o: объектный модуль, который непосредственно можно передавать редактору связей.

library.a: объектная библиотека, которую нужно использовать для разрешения внешних ссылок наравне со стандартными библиотеками.

Многие ОС, разработанные в 70-е гг., такие как RT-11, RSX-11, VAX/VMS, CP/M, навязывают программисту разделение имени на собственно имя и расширение, интерпретируя точку в имени файла как знак препинания. В таких системах имя может содержать только одну точку и соответственно иметь только одно расширение. Напротив, в ОС нового поколения – OS/2, Windows NT и даже в Windows 95 – реализована поддержка имен файлов свободного формата, которые могут иметь несколько каскадированных расширений, как и в Unix.

Однако никакие средства операционной системы не могут навязать прикладным программам правил выбора расширения для файлов данных. Это приводит к неприятным коллизиям. Например, почти все текстовые процессоры от Лексикон до Word 7.0 включительно используют расширение файла .doc (сокращение от **document**), хотя форматы файлов у различных процессоров и даже у разных версий одного процессора сильно различаются.

Другая проблема связана с исполняемыми загрузочными модулями. Обычно система использует определенное расширение для исполняемых файлов. Так, VMS, MS/DR DOS, OS/2, MS Windows и Windows NT используют расширение .exe: сокращение от **executable** (**исполняемый**). Однако по мере развития системы формат загрузочного модуля может изменяться. Так, например, OS/2 v3.0 Warp поддерживает по крайней мере пять различных форматов загрузочных модулей.

1. 16-разрядные сегментированные загрузочные модули: формат OS/2 1.x.
2. 32-разрядные загрузочные модули, использующие **"плоскую"** (**flat**) модель памяти: формат OS/2 2.x.
3. 32-разрядные модули нового формата, использующие упаковку кода и данных: формат OS/2 3.x.
4. DOS-овские .exe и .com модули.
5. Загрузочные модули MS Windows.

Для исполнения последних двух типов программ OS/2 создает задачу, работающую в режиме совместимости с 8086. Эта задача запускает копию ядра DOS и, если это необходимо, копию MS Windows, которые уже выполняют загрузку программы. Загрузочные модули всех трех "родных" форматов загружаются системой непосредственно.

Так или иначе загрузчик должен уметь правильно распознавать все форматы. При этом он не может использовать расширение файла: файлы всех перечисленных форматов имеют расширение .EXE.

Похожая ситуация имеет место в системах семейства Unix, где бинарные загрузочные модули и командные файлы вообще не имеют расширения. При этом большинство современных версий системы также поддерживает несколько различных исторически сложившихся форматов загрузочного модуля.

Разработчики Unix столкнулись с этой проблемой еще в 70 гг. В качестве решения они предложили использовать **магические числа** (**magic number**) – соглашение о том, что файлы определенного формата содержат в начале определенный байт или последовательность байтов. Первоначально это были численные коды; файл /etc/magic содержал коды, соответствующие известным типам файлов. Позднее в качестве магических чисел стали использоваться длинные текстовые строки. Так, например, изображения в формате Compuserve GIF 87a должны начинаться с символов GIF87a.

Легко понять, что магические числа ничуть не лучше расширений, а во многих отношениях даже хуже. Например, пользователь, просмотрев содержимое каталога, не может сразу узнать типы содержащихся в ней файлов. Еще хуже ситуация, когда расширение файла не соответствует его реальному типу. Это будет вводить в заблуждение не только пользователя, но и некоторые программы, полагающиеся при определении формата на расширение вместо магического числа.

С длинными мнемоническими текстовыми строками может быть связана еще одна забавная проблема, которая может иметь неприятные последствия. Например, текстовый файл следующего содержания:

Пример 15

```
GIF87a -- это очень плохой формат
хранения изображений.
```

будет воспринят некоторыми программами как изображение в формате Compuserve GIF 87a, каковым он, безусловно, не является.

Пытаясь как-то решить проблему идентификации типа файла, разработчики Macintoch отказались как от расширений, так и от магических чисел. В MacOS каждый файл состоит из двух частей или **ветвей** (**forks**): ветви данных (**data fork**) и ветви ресурсов (**resource fork**). Кроме идентификатора типа файла, ветвь ресурсов хранит информацию о:

- иконке, связанной с этим файлом;
- расположении иконки в открытой **папке** (**folder**);
- программе, которую нужно запустить при “открытии” этого файла.

Еще дальше в этом же направлении пошли разработчики системы OS/2. В этой системе с каждым файлом связан набор **расширенных атрибутов** (**extended attributes**). Атрибуты имеют вид `ИМЯ:Значение`. При этом значение может быть как текстовой строкой, так и блоком двоичных данных произвольного формата и размера. Некоторые расширенные атрибуты используются оболочкой Workplace Shell (`WPS`) как эквивалент ветви ресурсов в MacOS: для идентификации типа файла, связанной с ним иконы и размещения этой иконы в открытом окне. Тип файла идентифицируется текстовой строкой. Например, программа на языке C идентифицируется строкой `C code`. Это резко уменьшает вероятность конфликта имен типов – ситуацию, довольно часто возникающую при использовании расширений или магических чисел.

Другие атрибуты могут использоваться для других целей. Например, LAN Server – файловый сервер фирмы IBM – использует расширенные атрибуты для хранения информации о собственнике файла и правах доступа к нему. Некоторые текстовые редакторы используют расширенные атрибуты для хранения положения курсора при завершении последней сессии редактирования, так что пользователь всегда попадает в то место, где он остановился в прошлый раз.

Кроме того, расширенные атрибуты могут использоваться и для хранения сведений о назначении файла. В OS/2 существуют предопределенные расширенные атрибуты с именами `Subject` (тема), `Comment` (комментарии) и `Key phrases` (ключевые фразы), которые могут, например, использоваться для поисков документов, относящихся к заданной теме. К сожалению, такой поиск возможен только если создатель документа позаботился присвоить этим атрибутам правильные значения...

8.2 Монтирование файловых систем

Прежде чем ОС сможет использовать файловую систему, она должна выполнить над этой системой операцию, называемую **монтированием** (**mount**). В общем случае операция монтирования включает:

- проверку типа монтируемой ФС;
- проверку целостности ФС;
- считывание системных структур данных и инициализация соответствующего модуля файлового менеджера (драйвера файловой системы);
- в некоторых случаях – модификацию ФС с тем, чтобы указать, что она уже смонтирована – устанавливается так называемый **флаг загрязнения** (**dirty flag**). Это может быть необходимо, если ФС неустойчива к сбоям[3] В этом случае при ее размонтировании необходимо выполнить специальные операции.

  Если система пытается подмонтировать такую ФС и видит, что флаг загрязнения установлен, это означает, что при последнем использовании ее не размонтировали надлежащим образом. В этом случае перед монтированием необходимо запустить программу починки ФС, которая выявит все проблемы, возникшие при таком неправильном размонтировании. Чаще всего проблем нет, но они *могут* возникнуть;
- включение новой файловой системы в общее пространство имен. В различных системах это делается различными способами.

Если мы монтируем ФС, размещенную на удаленной машине (**файловом сервере**), то шаги 1 и 2 заменяются на установление соединения этим файловым сервером.

Многие пользователи MS/DR DOS никогда не сталкивались с понятием монтирования. Дело в том, что эта система (как и многие другие ДОС, например RT-11) выполняет упрощенную процедуру монтирования при каждом обращении к файлу. Упрощения состоят в пропуске шагов 1 и 2 и отсутствии шага 4 (ФС MS/DR DOS устойчива к сбоям).

При работе с файловыми серверами Novell Netware монтирование серверных файловых систем производится командой `MAP`.

Обычно имя файла в подмонтированной файловой системе имеет вид `ИМЯ_ФС:\имена\каталогов\имя.файла` При этом вместо разделителей `':'` и `'\'` могут использоваться другие символы.

В RT-11, RSX-11 и VMS в качестве имени файловой системы используется имя физического устройства, на котором размещена ФС. Если используется DECNet, перед именем устройства можно

[3] Устойчивые и неустойчивые к сбоям ФС обсуждаются в п. 8.4. Здесь мы отметим только, что большинство современных высокопроизводительных ФС относятся к категории неустойчивых.

поместить имя узла сети, на котором это устройство находится. Полное имя файла в VMS выглядит так: `DUA0:[USERS.FAT_BROTHER.WORK]test.exe`. При этом `DUA0` означает дисковое устройство 0, присоединенное к дисковому контроллеру `A`: `Disk Unit A [device #] 0`, `USERS.FAT_BROTHER.WORK]` означает каталог `WORK` в каталоге `FAT_BROTHER` в каталоге `USERS`.

В CP/M, MS/DR DOS и OS/2 имена файловых систем обозначаются буквами латинского алфавита, а сами файловые системы часто почему-то называются "драйвами". При некотором желании можно использовать в качестве имен ФС также символы `'['` и `']'`. Устройства `A:` и `B:` – это всегда дисководы гибких дисков[4]; устройство `C:` – обычно первый жесткий диск, или первый раздел на первом жестком диске.

Авторов всегда интересовал вопрос: что будет делать пользователь, когда у него кончатся доступные буквы алфавита? При использовании только локальных дисков такая ситуация кажется маловероятной, но при подключении к нескольким файловым серверам количество используемых файловых систем резко возрастает...

В OS/2, Windows NT и Windows for Workgroups эта проблема решена использованием так называемых UNC-имен, задающих имя файла в виде `\\NODE\SHARE\PATH\FILE.NAM`, где `NODE` – это имя сетевого узла, `SHARE` – имя разделяемого ресурса на этом узле (это может быть не только разделяемый каталог, но и принтер, а в OS/2 также и модем), а `PATH\FILE.NAM` – путь к файлу относительно разделяемого каталога. К сожалению, далеко не все старые программы понимают такие имена. Например, даже стандартный командный процессор системы Windows NT не может исполнить команду `cd \\NODE\SHARE\DIR`.

ДОС, как правило, помещают в пространство имен все доступные блочные устройства, не выполняя полной процедуры монтирования. Если какое-то из этих устройств не содержит ФС известного типа, то система будет возмущаться при обращениях к такому устройству, но не удалит его из списка доступных ФС. А иногда даже не будет возмущаться – попробуйте поставить в дисковод машины под управлением MS/DR DOS дискету, не содержащую файловой системы (например, созданную программой `tar`) и сказать `DIR A:`. Скорее всего, вы увидите несколько экранов мусора, но ни одного сообщения об ошибке!

В операционных системах семейства Unix смонтированные ФС выглядят как каталоги единого дерева[5], начинающегося с корневого каталога, выделенной первой ФС, называемой *корневой* (*root*). Администратор системы может подмонтировать новую ФС к любому каталогу, находящемуся на любом уровне дерева. Такой каталог после этого называют "точкой монтировки", но это выражение отражает только текущее состояние каталога. После того как мы размонтируем ФС, мы сможем использовать этот каталог как обычный, и наоборот, мы можем сделать точкой монтировки *любой* каталог.

Такой подход имеет неочевидное, на первый взгляд, но серьезное преимущество перед раздельными пространствами имен для разных физических файловых систем. Преимущество состоит в том, что пространство имен оказывается "отвязано" от физического размещения файлов. Поэтому администратор может поддерживать неизменную структуру дерева каталогов, перемещая при этом отдельные ветви по дискам ради более

[4] Любопытная деталь – 3.5' диск совсем не гнется. Тем не менее, он, почему-то, называется гибким.

[5] Строго говоря, структура каталогов в UNIX не обязана являться деревом; но об этом ниже

эффективного использования дискового пространства или даже просто ради удобства администрирования.

По традиции все Unix-системы имеют примерно одинаковую структуру дерева каталогов: системные утилиты находятся в каталоге `/bin`, системные библиотеки – в каталоге `/lib`, конфигурационные файлы - в каталоге `/etc` и т.д. Например, база данных об именах пользователей всегда находится в файле `/etc/passwd`.

8.3 Структуры файловых систем

8.3.1 Простые файловые системы

Наиболее простой файловой системой можно считать структуру, создаваемую архиватором системы UNIX – программой `tar` (Tape ARchive – архив на [магнитной] ленте). Этот архиватор просто пишет файлы один за другим, помещая в начале каждого файла заголовок с его именем и длиной. Аналогичную структуру имеют файлы, создаваемые архиваторами типа `arj`; в отличие от них, `tar` не упаковывает файлы.

Если вы хотите найти какой-то определенный файл, вы должны прочитать первый заголовок; если это не тот файл, то отмотать ленту до его конца, прочитать новый заголовок и т.д. Если мы часто обращаемся к отдельным файлам, то это не очень удобно, особенно если учесть, что цикл перемотка - считывание - перемотка у большинства лентопротяжных устройств происходит намного медленнее, чем простая перемотка.

Изменение же длины файла в середине архива или его стирание вообще превращается в целую эпопею. Поэтому `tar` используется для того, чтобы собрать файлы с диска в некую единую сущность, например, для передачи по сети или для резервного копирования, а для работы файлы обычно распаковываются на диск или другое устройство с произвольным доступом.

Чтобы не заниматься при каждом новом поиске просмотром всего устройства, удобнее всего разместить каталог в определенном месте, например в начале ленты. Наиболее простую, из известных авторам, файловую систему такого типа имеет ОС RT-11[6] В этой ФС, как и во всех, обсуждаемых далее, место на диске или ленте выделяется по блокам. Размер блока, как правило, совпадает с аппаратным размером сектора (512 байт у большинства дисковых устройств), однако многие ФС могут использовать логические блоки, состоящие из нескольких секторов (так называемые кластеры).

Использование блоков и кластеров вместо адресации с точностью до байта обусловлено двумя причинами.

Во первых, у большинства устройств произвольного доступа доступ произволен лишь с точно-

[6]Это единственная известная авторам файловая система, которая с одинаковым успехом применяется как на лентах, так и на устройствах с произвольным доступом. Все более сложные ФС, обсуждаемые далее, используются только на устройствах произвольного доступа. В наше время наиболее распространенный тип запоминающего устройства произвольного доступа – это магнитный или оптический (реже - магнитооптический) диск, поэтому далее в тексте мы будем называть такие устройства просто дисками – для краткости, хотя такое сокращение и не вполне корректно.

стью до сектора, то есть нельзя произвольно считать или записать любой байт – нужно считывать или записывать весь сектор целиком. Именно поэтому в системах семейства Unix такие устройства называются **блочными** (**block-oriented**).

Во-вторых, использование крупных адресуемых единиц позволяет резко увеличить адресуемое пространство. Так, используя 16-битный указатель, с точностью до байта можно адресовать всего 64к, но если в качестве единицы адресации взять 512-байтовый блок, то объем адресуемых данных сможет достигать 32 мегабайт; если же использовать кластер размером 32к, то можно работать с данными объемом до 2 Гб. Аналогично, 32-битовый указатель позволяет адресовать с точностью до байта 4Гб данных, то есть уже далеко не каждый современный жесткий диск; зато если перейти к адресации по блокам, то адресное пространство вырастет до 256Гб, что уже вполне приемлемо практически для всех современных запоминающих устройств.

Таким образом, адресация по блокам и кластерам позволяет использовать в системных структурах данных короткие указатели, что приводит к уменьшению объема этих структур и к снижению накладных расходов. Под накладными расходами в данном случае подразумевается не только освобождение дискового пространства, но и ускорение доступа: структуры меньшего размера быстрее считываются, в них быстрее производится поиск и т.д. Однако увеличение объема кластера имеет оборотную сторону – оно приводит к так называемой внутренней фрагментации.

Например, мы записываем в файл структуру данных объемом 624 байта, Файл этот предполагается хранить на устройстве с 512-байтовым блоком. В один блок наш файл не влезает, поэтому нужно выделять два блока. Первый блок можно занять целиком, зато во втором блоке будет занято всего 112 байт, а оставшиеся 400 байт не могут быть отданы другому файлу и оказываются потеряны. Это и называется внутренней фрагментацией.

Величина потерь зависит от среднего размера файла на диске. Грубая оценка состоит в том, что на каждый файл в среднем теряется полкластера, т.е. отношение занятого пространства к потерянному будет $\frac{1}{2}\frac{Ns_c}{N\bar{l}}$, где N – количество файлов на диске, s_c – размер кластера, а $\bar{l}$ – средний размер файла. Иными словами, потери будут равны $\frac{s_c}{2\bar{l}}$, т.е. будут линейно расти с увеличением размера кластера.

Когда средний размер файла оказывается сравним с размером кластера, наша формула теряет точность, но все равно дает хорошую оценку порядка величины потерь. Так, если $s_c = \bar{l}$, наша формула дает 50 % потерь, что вполне согласуется со здравым смыслом: если файл чуть короче кластера, теряется только это "чуть"; зато если файл чуть длиннее, то для него отводится два кластера и второй кластер теряется почти весь. Точная величина потерь сейчас уже определяется количеством файлов, которые "чуть" длиннее среднего. Количество таких файлов вовсе не равно $\frac{1}{2}N$, а зависит от распределения файлов по длине, но мы предпочитаем оставить вывод точной формулы любопытному читателю.

Ряд современных файловых систем использует механизм, по английски называемый **block suballocation**, то есть размещение частей блоков. В этих ФС кластеры имеют большой размер, но есть возможность разделить кластер на несколько блоков меньшего размера и записать в эти блоки "хвосты" от нескольких разных файлов. Это, безусловно, усложняет ФС, но позволяет одновременно использовать преимущества, свойственные и большим, и маленьким блокам. Поэтому ряд распространенных ФС, например файловая система Novell Netware 4.1 и FFS (известная также как UFS и Berkley FS), используемая во многих системах семейства Unix, используют этот механизм.

Но вернемся к простым файловым системам. В RT-11 каждому файлу выделяется непрерывная область на диске. Благодаря этому в каталоге достаточно хранить адрес первого блока файла и его длину, также измеренную в блоках. В RT-11 поступили еще проще: порядок записей в каталоге совпадает с порядком файлов на диске, и началом файла считается окончание предыдущего файла. Свободным участкам диска тоже соответствует запись в каталоге. При создании файла система ищет первый свободный участок подходящего размера.

Фактически эта структура отличается от формата `tar` только тем, что каталог вынесен в начало диска, и существует понятие свободного участка внутри области данных. Но эта простая организация имеет очень серьезные недостатки.

1. При создании файла программа должна указать его длину. Часто это бывает затруднительно, например, при создании объектного файла во время компиляции программы на языке высокого уровня.

   Особенно неудобно оказывается увеличивать в размере уже созданный файл. Точнее, это просто невозможно: вместо удлинения старого файла приходится создавать файл новой длины и копировать содержимое старого файла в него.

2. При хаотическом создании и удалении файлов возникает проблема фрагментации свободного пространства. Для ее решения существует специальная программа `SQUEESE` (сжать [диск]), которая переписывает файлы так, чтобы объединить все свободные фрагменты. Эта программа требует много времени, особенно для больших дисковых томов, и потенциально опасна: если при ее исполнении произойдет сбой системы (а с машинами третьего поколения такое случалось по нескольку раз в день), то значительная часть данных будет необратимо разрушена.

Для того, чтобы решить обе эти проблемы, необходимо позволить файлам занимать несмежные области диска. Наиболее простым решением было бы хранить в конце каждого блока файла указатель на следующий, то есть превратить файл в связанный список блоков. При этом, естественно, в каждом блоке хранилось бы не 512 байт данных, а на 2 - 4 байта меньше. При чисто последовательном доступе к файлу такое решение было бы вполне приемлемым, но при попытках реализовать произвольный доступ возникнут неудобства. Возможно, поэтому, а может и по каким-то исторически сложившимся причинам такое решение не используется ни в одной из известных авторам ОС.

Однако, отчасти похожее решение было реализовано в MS DOS и DR DOS. Эти системы создают на диске таблицу, называемую **FAT** (**File Allocation Table** – таблица размещения файлов). В этой таблице каждому блоку, предназначенному для хранения данных, соответствует 12-битовое значение. Если блок свободен, то значение будет нулевым. Если же блок принадлежит файлу, то значение будет равно **адресу следующего блока этого файла**. Если это последний блок в файле, то значение будет `0xFFF`. Существует также специальный код для обозначения **плохого** (**bad**) блока, не читаемого из-за дефекта физического носителя. В каталоге хранится номер первого блока и длина файла, измеряемая в байтах.

Емкость диска при использовании 12-битной `FAT` оказывается ограничена 4096-ю блоками (2Мб), что приемлемо для дискет, но совершенно не годится для жестких дисков и других устройств большой емкости. На таких устройствах DOS использует `FAT` с 16-битовыми элементами. На совсем уж больших (более 32 мегабайт) дисках DOS выделяет пространство не блоками, а **кластерами** из нескольких блоков. Эта файловая система так и называется – FAT.

Такая файловая система очень проста и имеет одно серьезное достоинство: врожденную **устойчивость к сбоям** (**fault tolerance**), но об этом ниже.

В то же время она имеет и ряд серьезных недостатков.

Первый недостаток состоит в том, что при каждой операции над файлами система должна обращаться к `FAT`. Это приводит к частым перемещениям головок дисковода и в результате к резкому снижению производительности. Действительно, исполнение программы `arj` на одной и той же машине под MS DOS и под DOS-эмулятором систем Unix или OS/2 различается по скорости почти в 1.5 раза. Особенно это заметно при архивировании больших каталогов.

Использование дисковых кэшей, а особенно помещение `FAT` в оперативную память, существенно ускоряет работу, хотя обычно `FAT` кэшируется только на чтение ради устойчивости к сбоям.

Но здесь мы сталкиваемся со специфической проблемой: чем больше диск, тем больше у него `FAT`, соответственно, тем больше нужно памяти: у тома Novell Netware 3.12размером 1.115 Гбайт с размером кластера 4 кбайта размер `FAT` достигает мегабайта[7]. При монтировании такого тома Netware занимает под `FAT` и кэш каталогов около 6 Мбайт памяти.

Для сравнения, Netware 4 использует block suballocation, поэтому там можно относительно безнаказанно увеличивать объем кластера и нет необходимости делать кластер таким маленьким. Для дисков такого объема Netware 4 устанавливает размер кластера 16 килобайт, что приводит к уменьшению всех структур данных в 4 раза.

Понятно, что для MS DOS, которая умеет адресовать всего 1 Мбайт (1088 кбайт, если разрешен `HMA`), хранить такой `FAT` в памяти целиком просто невозможно.

Поэтому разработчики фирмы MicroSoft пошли другим путем: они ограничили разрядность элемента `FAT` 16 битами. При этом размер таблицы не может превышать 128 кбайт, что вполне терпимо.

[7]Легко понять, что Netware использует `FAT` с 32-разрядными элементами. При 16-разрядном элементе `FAT` дисковый том такого объема с таким размером кластера просто невозможен.

Зато вся файловая система может быть разбита только на 64 К блоков. В старых версиях MS DOS это приводило к невозможности создавать файловые системы размером более 32 Мбайт. В версиях старше 3.11 появилась возможность объединять блоки в кластеры. Например, на дисках размером от 32 Мбайт до 64 Ммбайт кластер будет состоять из 2 блоков и иметь размер 1 кбайт. На дисках размером 128-265 Мбайт кластер будет уже размером 4 кбайта, и т.д.

Windows 95 использует защищенный режим процессора x86, поэтому адресное пространство там намного больше одного мегабайта. Разработчкик фирмы Microsoft решили воспользоваться этим и реализовали версию FAT с 32-битовым элементом таблицы – так называемый FAT32. Однако дисковый кэш Windows 95 не стремился удержать весь FAT в памяти; вместо этого FAT кэшировался на общих основаниях, наравне с пользовательскими данными. Поскольку работа с файлами большого (больше одного кластера) объема требует прослеживания цепочки элементов FAT, а соответствующие блоки таблицы могут не попадать в кэш, то производительность резко падает. В сопроводительном файле Microsoft признает, что производительность FAT32 на операциях последовательного чтения и записи может быть в полтора раза ниже, чем у кэшированного FAT16.

Реализация FAT32 распространялась в виде патча к Windows 95, но новая ФС не вызвала энтузиазма у пользователей. Авторам не известно ни одного случая практического применения этой ФС, да и сама фирма Microsoft не очень активно занимается ее продвижением.

В качестве итога можно сказать, что при использовании FAT на больших дисках мы вынуждены делать выбор между низкой производительностью, чудовищными по современным меркам требованиями к оперативной памяти или большим размером кластера, который приводит к большим потерям из-за внутренней фрагментации.

Например, у одного из авторов на локальном диске емкостью 210 Мбайт (кластер 4 кбайта) терялось около 10 Мбайт, т.е. около 5 % емкости диска. Средний размер файла был около 20 кбайт в основном за счет пакета emTeX, который содержит несколько сотен коротких файлов. По приведенной в начале этого пункта формуле можно оценить, что при том же среднем размере файла на диске емкостью 1 Гбайт терялось бы 40 % пространства... .

Нужно отметить, впрочем, что более характерный средний размер файла на современных персоналках ближе к 64 - 128 кбайтам, поэтому потери в большинстве случаев не так катастрофически велики. На гигабайтном разделе будет, таким образом, теряться “всего лишь” около 10 %. Действительно, размер кластера равен 16 кбайтам, и потери по нашей формуле будут $16K/2*64K = 1/8 = 0.125 \approx 0.1$. Но на диске объемом 4 Гбайта мы опять получаем около 50 % потерь!

При современных темпах развития электроники и падения цен на нее четырехгигабайтные диски для персоналок станут в ближайшие несколько лет реальностью. А если учесть скорость роста размеров коммерческих прикладных программ (программы из пакета MicroSoft Office занимают по нескольку десятков мегабайт каждая), то такие диски быстро станут остро необходимыми.

В свете всего этого сложно оправдать решение фирмы MicroSoft использовать в новой ОС

Windows 95 старую файловую систему с небольшими изменениями для поддержки длинных имен файлов. Ведь для эффективного управления большими объемами данных необходимо что-то более сложное, чем FAT.

8.3.2 "Сложные" файловые системы

Структуры "сложных" файловых систем отличаются большим разнообразием, однако можно выделить несколько общих идей.

Обычно файловая система начинается с заголовка, или, как это называется в системах семейства Unix, **суперблока** (**superblock**). Суперблок хранит информацию о размерах дискового тома, отведенного под ФС, указатели на начала системных структур данных и другую информацию, зависящую от типа ФС. Например, для статических структур может храниться их размер. Часто суперблок содержит также **магическое число** – идентификатор типа файловой системы. Аналог суперблока существует даже в FAT – это так называемая **загрузочная запись** (**boot record**).

Практически все современные ФС разделяют список свободных блоков и структуры, отслеживающие размещение файлов. Чаще всего вместо списка свободных блоков используется битовая карта, в которой каждому блоку соответствует один бит: занят/свободен. В свою очередь, список блоков для каждого файла обычно связан с управляющей структурой файла.

На первый взгляд, такую структуру кажется естественным хранить в каталоге, но их обычно выносят в отдельные структуры, часто собранные в специальные таблицы описателей файлов – таблицу **инодов**, **метафайл** и т.д. Такое решение уменьшает объем каталога и, соответственно, ускоряет поиск файла по имени. К тому же многие ФС сортируют записи в каталоге по имени файла, также с целью ускорения поиска. Понятно, что сортировка записей меньшего размера происходит быстрее.

В файловой системе HPFS, используемой в OS/2 и Windows NT, каждая запись в каталоге содержит имя файла и указатель на **fnode** (**файловую запись**). Каталоги в этой ФС организованы в виде B-деревьев и отсортированы по именам файлов.

Файловая запись занимает один блок на диске и содержит список так называемых **extents** (**"расширений"**)[8]. Каждый экстент описывает непрерывную последовательность дисковых блоков, выделенных файлу. Он задает начальный блок такой последовательности и ее длину. Вместо списка свободных блоков используется битовая карта диска, в которой каждому блоку соответствует один бит: занят/свободен.

Нужно отметить, что идея экстентов далеко не нова: аналогичная структура используется в некоторых версиях Unix с начала 80-х годов, а истоки этой идеи теряются в глубине 60-х.

Файловая запись размешается перед началом первого экстента файла, хотя это и не обязательно. Она занимает один блок (512 байт) и может содержать до десяти описателей экстентов. Кроме того, она содержит информацию о времени создания файла, его имени и **расширенных атрибутах**[9]. Если для списка экстентов

[8] У этого термина нет приемлемого русского аналога. В переводах документации по OS/360 (ОС ЕС) так и писали: **экстент**.

[9] Подробнее о расширенных атрибутах см. в разделе 8.1.3.

или расширенных атрибутов места не хватает, то для них также выделяются экстенты. В этом случае экстенты размещаются в виде B-дерева для ускорения поиска. Максимальное количество экстентов в файле не ограничено ничем, кроме размера диска.

Пользовательская программа может указать размер файла при его создании. В этом случае система сразу попытается выделить место под файл так, чтобы он занимал как можно меньше экстентов. Если программа не сообщила размера файла, используется значение по умолчанию. Фактически, HPFS размещает место под файл по принципу **worst fit** (**наименее подходящего**), начиная выделение с наибольшего непрерывного участка свободного пространства. В результате фрагментированными оказываются только файлы, длина которых увеличивалась во много приемов или же те, которые создавались при почти заполненном диске. При нормальной работе файл редко занимает больше 3 - 4 экстентов.

Ещё одно любопытное последствие стратегии **worst fit** заключается в том, что пространство, освобожденное стертым файлом, обычно используется не сразу. Отмечались случаи, когда файл на активно используемом диске удавалось восстановить через несколько дней после стирания.

Экстенты открытых файлов и карта свободных блоков во время работы размещаются в ОЗУ, поэтому производительность такой ФС в большинстве ситуаций намного (в 1.5 - 2 раза и более) выше, чем у FAT без кэша, при вполне приемлемых требованиях к памяти и размере кластера 512 байт.

Понятно также, что структура HPFS упрощает произвольный доступ к файлу: вместо прослеживания цепочки блоков нам нужно проследить цепочку экстентов, которая гораздо короче.

Более подробное описание структуры HPFS можно найти в статье [16]. Среди пользователей OS/2 считается целесообразным форматировать все разделы емкостью более 128M под HPFS, поскольку при этом выигрывается как скорость, так и эффективность использования дискового пространства; кроме того, исчезает необходимость в дефрагментации и появляется возможность создавать файлы и каталоги с длинными именами, не укладывающимися в модель `8.3`, принятую в MS DOS.

Но за эти преимущества приходится платить неустойчивостью к сбоям[10]. В отличие от ДОС, спонтанные развалы системы с последующим зависанием в OS/2 случаются относительно редко, даже при запуске программ, заведомо содержащих ошибки (как при разработке или тестировании прикладного программного обеспечения). С другой стороны, на практике “неустойчивость” приводит лишь к тому, что после аварийной перезагрузки автоматически запускается программа починки ФС, что увеличивает время перезагрузки в несколько раз; реальный же риск потерять данные при сбое не выше, а как показывает практика, даже существенно ниже, чем при использовании FAT, поэтому игра явно стоит свеч.

Наиболее интересна структура файловых систем в ОС семейства Unix. В этих ФС каталог не содержит почти никакой информации о файле. Там хранится только имя файла и номер его **инода** (**i-node** – по-видимому, сокращение от index node: индексная запись). Иноды всех файлов в данной ФС собраны в таблицу, которая так и называется: таблица инодов. В ранних версиях Unix таблица инодов занимала фиксированное пространство в начале устройства; в современных файловых системах эта таблица разбита на участки, распределенные по диску.

[10]Проблема устойчивости к системным сбоям обсуждается в разделе 8.4.

Например, в файловой системе BSD Unix FFS (Fast File System – быстрая файловая система), которая в Unix SVR4 называется просто UFS (Unix File System), диск разбит на группы цилиндров. Каждая группа цилиндров содержит копию суперблока, битовую карту свободных блоков для данного участка и таблицу инодов для файлов, расположенных в пределах этого участка. Такая распределенная структура имеет два преимущества:

- Ускорение доступа к системным структурам данных. Когда системные данные расположены вблизи от блоков пользовательских данных, уменьшается расстояние, на которое перемещаются головки дисковода.
- Повышенная устойчивость к сбоям носителя. При повреждении участка поверхности диска теряется только небольшая часть системных данных. Даже потеря суперблока не приводит к потере структуры файловой системы.

Инод хранит информацию о самом файле и его размещении на диске. Информационная часть инода может быть получена системным вызовом `int stat(const char * fname, struct stat * buf);`. Формат структуры `stat` описан во многих руководствах по языку C, ОС UNIX и стандарту POSIX, например в работе [12]. Эта структура содержит:

тип файла. Исследуя это поле, можно понять, является данный объект файлом данных или специальным файлом. В этом же поле закодированы права доступа к файлу;

идентификаторы хозяина файла и группы, к которой хозяин принадлежит;

времена:

- создания файла,
- последней модификации файла,
- последнего доступа к файлу;

длину файла. Для специальных файлов это поле часто имеет другой смысл;

идентификатор файловой системы, в которой расположен файл;

количество связей файла. Это поле заслуживает отдельного обсуждения.

Структура, описывающая физическое размещение файла по диску, недоступна пользовательским программам. Собственно, формат этой структуры может быть не очень элегантен. Например, в файловой системе Veritas это список экстентов, похожий на HPFS; в файловых системах s5, Xenix и FFS это массив из 13 чисел, задающих номера физических блоков файла. Если файл в s5 содержит более десяти блоков (т.е. его длина больше 5 кбайт), то предпоследние три указателя обозначают не блоки данных, а так называемые **косвенные блоки** (**indirection blocks**), в которых хранятся указатели на следующие блоки данных и, возможно, на следующие косвенные блоки.

Наиболее интересная особенность Unix-овых ФС состоит не в этом. Внимательный читатель, возможно, заметил, что инод не содержит имени файла. С другой стороны, он содержит счетчик

связей (**links**) – ссылок на этот файл из каталогов. Т.е. на один и тот же инод можно ссылаться из различных каталогов или из одного каталога под различными именами. Иными словами,

один и тот же файл в этих ФС может иметь несколько различных имен.

Именно это и имелось в виду, когда говорилось, что структура каталогов в ОС UNIX не обязана являться деревом.

Это свойство предоставляет неоценимые возможности для организации иерархии каталогов, но имеет и некоторые оборотные стороны.

1. Создание нескольких связей для каталога потенциально опасно – оно может привести к возникновению кольца, в котором каталог является своим собственным подкаталогом. Отслеживать такую ситуацию сложно, поэтому разработчики ОС UNIX запретили создавать дополнительные имена для каталогов.

2. Удаление файла превращается в проблему: чтобы удалить файл, нужно проследить все его связи. Поэтому UNIX не имеет средств для удаления файла, а имеет только системный вызов `unlink` – удалить связь. Когда у файла не остается связей, он действительно удаляется.

   Этот подход вполне разумен, но также имеет неожиданную оборотную сторону: поскольку теперь стирание файла – это операция не над файлом, а над каталогом, то **для удаления файла не нужно иметь никаких прав доступа к нему. Достаточно иметь право записи в каталог**. Одно из самых неприятных последствий состоит в том, что нельзя защитить индивидуальный файл от стирания: либо вы закрываете на запись весь каталог и не можете ни удалить в нем любой другой файл, ни создать новый; либо вы открываете его на запись и тогда можете стереть ваш драгоценный файл по ошибке.

3. Такие связи (называемые еще *жесткими* связями), как легко понять, могут создаваться только в пределах одной файловой системы, поскольку каждая ФС имеет собственную таблицу инодов, и соответственно собственную их нумерацию.

Последнее обстоятельство резко уменьшает полезность жестких связей для организации иерархии каталогов. Эта проблема была осознана еще в 70-е гг., и программисты из группы BSD придумали интересное новое понятие – символическую связь.

Символическая связь представляет собой специальный файл. Вместо блоков данных инод такого файла содержит текстовую строку – *имя того файла, с которым создана связь*. Это может быть файл на другой файловой системе, в том числе и на такой, которая сама по себе не поддерживает ни жестких, ни символических связей, например, FAT, HPFS или файловом сервере Novell Netware. Такого файла может и вообще не существовать, например потому, что его уже удалили, или потому, что файловая система, в которой он находится, не смонтирована, или просто потому, что имя было задано неправильно. Тогда попытки открыть символическую связь будут завершаться неуспехом с кодом ошибки "файла не существует".

Единственным недостатком символических связей является их относительно низкая **дуракоустойчивость** (**fool-tolerance**): глупый пользователь может не распознать ситуации, когда символическая связь указывает в никуда. Зато они обеспечивают полную свободу в размещении и именовании файлов.

Пример из жизни: На двух Unix-системах с именами `orasrv` и `Indy` установлен один и тот же программный продукт: редакторная система GNU Emacs. Бинарные загрузочные модули для этих систем различаются, но большая часть Emacs – это файлы на языке Emacs Lisp, которые с одинаковым успехом могут использоваться обеими системами. Поэтому у администратора системы возникает желание использовать одну копию lisp-файлов.

При этом Emacs установлен таким образом, что он ищет все свои lisp-файлы в каталоге `/usr/local/lib/emacs/19.27/lisp`. Изменение этого каталога потребовало бы частичной переустановки продукта. Но его не надо менять! Мы просто подмонтируем на машине `orasrv` каталог `/` машины `Indy` как `/fs/Indy`, и исполняем следующие команды:

Пример 16

```
cd /usr/local/lib/emacs/19.27 # перейти в заданный каталог
rm -Rf lisp       # стереть каталог lisp со всем содержимым
ln -s /fs/Indy/usr/local/lib/emacs/19.27/lisp lisp
# создать символическую связь с именем lisp с соответствующим
# каталогом на машине Indy
```

Вся операция вместе с ее планированием занимает около двух минут. Освобождается около 9 Мбайт дискового пространства. Emacs ничего не замечает, и работает без всяких проблем.

Через неделю у администратора возникает идея, что для удобства администрирования неплохо было бы монтировать с `orasrv` не всю файловую систему `Indy`, а только ее каталог `/home`. За чем дело стало: переносим на `Indy` каталог `lisp` в каталог `/home/emacs/19.27/lisp`; создаем в старом каталоге символическую связь с новой; редактируем файл `/etc/mnttab` на машине `orasrv` так, чтобы с `Indy` монтировалась только `/home`; меняем символическую связь каталога `lisp` на машине `orasrv`. Заставляем `orasrv` подмонтировать диски в соответствии с новым `/etc/mnttab`. Готово.

Операция занимает чуть больше двух минут, потому что нужно переносить девять мегабайт с одной файловой системы на другую. Emacs опять ничего не замечает и снова работает без всяких проблем. Администратор счастлив. Все довольны.

Сравните это описание с впечатлениями пользователя, который пытается переместить с одного ДОСовского драйва на другой пакет CorelDRAW! или любую другую программу, которая при установке записывает в свою конфигурацию полное имя каталога, в который ее ставили...

Символические связи уникальны для систем семейства Unix. Напротив, эквиваленты жестких связей существуют в других ОС. Фактически жесткие связи можно создавать в любой ФС, где каталоги содержат ссылки на централизованную базу данных вместо самого дескриптора файла.

Например, в файловой системе VAX/VMS данные о размещении файлов на диске хранятся в специальном **индексном** (**index**) файле; каталоги же хранят только индексы записей в этом файле. Основное отличие этой структуры от принятой в Unix состоит в том, что вместо статической таблицы или набора таблиц используется динамическая таблица, пространство под которую выделяется тем же методом, что и для пользовательских файлов. Этот же подход реализован в файловой системе NTFS, используемой в Windows NT, но там индексный файл называется **метафайлом** (**metafile**). Несколько более подробное описание структуры NTFS приводится в статье [17].

VAX/VMS и Windows NT позволяют создавать дополнительные имена для файлов, хотя в VMS утилиты починки ФС выдают предупреждение, обнаружив такое дополнительное имя. Все имена файла в этих ФС обязаны находиться в одной файловой системе. Кроме того, операция удаления файла в VMS ведет себя не так, как в Unix: применение операции удаления к любому из имен приводит к удалению самого файла, даже если существовали и другие имена.

8.4 Устойчивость ФС к сбоям

8.4.1 Устойчивость к сбоям питания

Свойство **устойчивости к сбоям питания** (**power-fault tolerance**) является одной из важных характеристик файловой системы. Строго говоря, имеется в виду устойчивость не только к сбоям питания, но и к любой ситуации, при которой работа с ФС прекращается без выполнения операции размонтирования. Поэтому правильнее было бы говорить об устойчивости к любым сбоям системы, а не только питания.

С другой стороны, если говорить просто об "устойчивости к сбоям", возникает неприятная двусмысленность. Под устойчивостью к сбоям можно понимать устойчивость к сбоям всей системы, например, к тем же сбоям питания, а можно понимать устойчивость к дефектам физического носителя, которые могут привести к потере части данных на диске. Вторая проблема также довольно сложна, но она обсуждается в разделе 8.4.4, а сейчас мы будем говорить только о первой проблеме, называя ее просто "устойчивостью" для краткости.

На самом деле, неожиданное прекращение работы с ФС может произойти не только при сбое питания, но и при:

- извлечении носителя из дисковода (подробнее об этом см. ниже);
- нажатии кнопки RESET нетерпеливым пользователем;
- фатальном аппаратном сбое;
- аппаратном сбое, который сам по себе не был фатальным, но система не смогла правильно восстановиться, что привело к ее развалу;

- разрушении системы из-за чисто программных проблем[11].

В системах класса ДОС последняя причина возникает очень часто, поэтому в таких системах можно использовать только устойчивые к сбоям ФС.

В системах класса ОС "спонтанные" зависания происходят намного реже. Система, зависающая по непонятным или неустранимым причинам раз в несколько дней считается малопригодной для серьезного использования, а делающая это раз в месяц – подозрительно ненадежной. Поэтому в таких системах можно позволить себе роскошь использовать неустойчивые к сбоям ФС. Тем более, что такие системы, как правило, обладают более высокой производительностью, чем устойчивые ФС.

К тому же перед выключением системы, интегрированной в сеть, необходимо уведомить всех клиентов и все сервера о разъединении. Только чисто клиентская машина может быть выключена из сети без проблем. Поэтому на сетевых серверах в любом случае необходима процедура **закрытия системы** (**shutdown**). Так почему бы не возложить на эту процедуру еще и функцию размонтирования ФС?

Напротив, в системах реального времени, которые по роду работы могут часто и неожиданно перезапускаться, например, при отладке драйверов или программ, осуществляющих доступ к аппаратуре, стараются использовать устойчивые к сбоям ФС.

Хотя первая из перечисленных выше причин – извлечение носителя из дисковода – не является "сбоем" даже в самом широком смысле этого слова, с точки зрения ФС это мало чем отличается от сбоя. Поэтому на удаляемых носителях, таких, как дискеты, можно использовать только устойчивые ФС.

Интересный альтернативный подход используется в компьютерах Macintosh и некоторых рабочих станциях. У этих машин дисковод не имеет кнопки для извлечения дискеты. Выталкивание дискеты осуществляется программно, подачей соответствующей команды дисководу. Перед подачей такой команды ОС может выполнить нормальное размонтирование ФС на удаляемом диске.

В узком смысле слова "устойчивость" означает лишь то, что такая ФС после аварийной перезагрузки необязательно нуждается в починке. Такие ФС обеспечивают целостность собственных структур данных в случае сбоя, но, вообще говоря, не гарантируют целостности пользовательских данных в файлах.

Нужно отметить, что даже если ФС считается в этом смысле устойчивой, некоторые сбои для нее могут быть опасны. Например, если запустить команду `DISKOPT` *на "устойчивой" файловой системе* FAT *и в подходящий момент нажать RESET, то значительная часть данных на диске может быть навсегда потеряна.*

---

[11]На практике часто сложно провести границы между тремя последними типами сбоев. В качестве примера можно привести проблему, описанную в разделе 7.3.2

Напротив, можно говорить об устойчивости в том смысле, что в ФС после сбоя гарантирована целостность пользовательских данных. Хотя после простого анализа можно убедиться, что такую гарантию нельзя обеспечить на уровне ФС; обеспечение такой целостности накладывает серьезные ограничения и на программы, работающие с данными, а иногда оказывается просто невозможным.

Характерный и очень простой пример: архиватор InfoZip работает над созданием архива. Программа сформировала заголовок файла, упаковала и записала на диск около 50 % данных, и в этот момент произошел сбой. В zip-архивах каталог находится в конце архивного файла и записывается туда после завершения упаковки всех данных. Обрезанный в случайном месте zip-файл не содержит каталога и поэтому, безусловно, является безнадежно испорченным.

Поэтому при серьезном обсуждении проблемы устойчивости к сбоям говорят не о гарантии целостности пользовательских данных, а об уменьшении вероятности их порчи.

Поддержание целостности структур ФС обычно гораздо важнее, чем целостность недописанных в момент сбоя пользовательских данных. Дело в том, что если при сбое оказывается испорчен создававшийся файл, это достаточно неприятно; если же окажется испорчена файловая система, в худшем случае это может привести к потере *всех* данных на диске, то есть к катастрофе. Поэтому обычно обеспечению целостности ФС при сбоях уделяется гораздо больше внимания.

Упоминавшаяся выше "врожденная" устойчивость к сбоям файловой системы FAT объясняется тем, что в этой ФС удаление блока из списка свободных и выделение его файлу производится одним действием – модификацией элемента FAT. Поэтому если во время этой процедуры произойдет сбой или дискета будет вынута из дисковода, то ничего страшного не случится: просто получится файл, которому выделено на один блок больше, чем его длина, записанная в каталоге. При стирании этого файла все его блоки будут помечены как свободные, поэтому вреда практически нет.

Нужно отметить, что при использовании отложенной записи FAT *и родственные ФС теряют это преимущество. При этом отложенная запись* FAT *является единственным способом добиться хоть сколько-нибудь приемлемой производительности от ФС с 32-битовым* FAT*. Поэтому, например, хотя* Novell NetWare *и использует ФС, основанную на 32-битной* FAT*, после аварийной перезагрузки эта система вынуждена запускать программу аварийной починки дисковых томов. Аналогичным образом ведет себя и 32-битная версия ДОСовской ФС* FAT32*.*

Если же система хранит отдельно список или карту свободных блоков, а отдельно списки блоков, выделенных каждому файлу, как это делают HPFS или ФС систем семейства Unix, то при прерывании операции выделения места в неподходящий момент могут либо теряться блоки (если мы сначала удаляем блок из списка свободных), либо получаться блоки, которые одновременно считаются и свободными, и занятыми (если мы сначала выделяем блок файлу).

Первая ситуация достаточно неприятна, вторая же просто недопустима: первый же файл, созданный после перевызова системы, будет "перекрещиваться" с испорченным. Поэтому все ОС, использующие файловые системы такого типа (системы семейства Unix, OS/2, Windows NT и т.д.), после аварийной перезагрузки первым делом проверяют свои ФС соответствующей программой починки.

Задача обеспечения целостности файловых систем при сбоях усложнияется тем, что дисковые подсистемы практически всех современных ОС активно используют отложенную запись, в том числе и при работе с системными структурами данных. Отложенная запись, особенно в сочетании с сортировкой запросов по номеру блока на диске, может приводить к тому, что изменения инода или ф-нода файла могут все-таки записываться на диск раньше, чем изменения списка свободных блоков, что может приводить к возникновению “скрещенных"файлов.

8.4.2 Восстановление ФС после сбоя

Чаще всего суперблок неустойчивых ФС содержит флаг **dirty** (**“грязный”**), сигнализирующий о том, что ФС, возможно, нуждается в починке. Этот флаг сбрасывается при нормальном размонтировании ФС и устанавливается при ее монтировании или при первой модификации после монтирования. Таким образом, если ОС погибла, не успев размонтировать свои дисковые тома, после перезагрузки на этих томах dirty-флаг будет установлен, что и будет сигнализировать о необходимости починки.

Починка состоит в том, что система прослеживает пространство, выделенное всем файлам. При этом должны выполняться следующие требования:

1. Каждая запись в каталоге должна иметь правильный формат и содержать осмысленные данные. Например, если запись помечена как свободная, она не должна ссылаться на данные, помеченные как принадлежащие файлу, или на инод. Не во всех ФС можно обнаружить ошибки такого типа.

2. Каждый блок или кластер диска должен принадлежать не более, чем одному файлу. Блоки, принадлежащие одновременно двум или более файлам, являются очень серьезной ошибкой. На практике это означает, что данные во всех этих файлах (в лучшем случае – во всех, кроме того, запись в который была последней) безнадежно испорчены.

   Чаще всего программа починки в этой ситуации требует вмешательства пользователя, с тем чтобы решить, какие из файлов следует удалить или обрезать по месту пересечения. Иногда для каждого из файлов создается копия “общего” блока, но и в этом случае пользователю все равно нужно определить, какие из файлов испорчены.

   Практически все ФС при выделении блока сначала удаляют его из списка свободных и лишь потом отдают файлу, поэтому при “обычных” сбоях перекрещивание файлов возникнуть может только как следствие отложенной записи в сочетании с сортировкой запросов. Возникновение таких ошибок обычно сигнализирует либо об ошибке в самом файловом менеджере, либо об аппаратных сбоях на диске, либо о том, что структуры ФС были модифицированы в обход файлового менеджера. Например, в ДОС это может быть признаком вирусной активности.

3. Каждому файлу должно быть выделено пространство, соответствующее его длине. Если файлу выделено больше блоков, чем требуется, лишние блоки помечаются как свободные. Если меньше, файл укорачивается. Возможно, в укороченных файлах часть данных оказывается потеряна.

4. Все блоки, не принадлежащие файлам, должны быть помечены, как свободные. Соответствующий тип ошибок – потерянные блоки – наиболее частый результат системных сбоев как в файловой системе FAT, так и в более сложных файловых системах. Сами по себе ошибки этого типа относительно безобидны.

   Обычно программа починки не помечает потерянные блоки как свободные, а выделяет из них непрерывные цепочки и создает из этих цепочек файлы. Например, в OS/2 программа починки пытается найти в потерянных блоках файловые записи, а потом создает ссылки на найденные таким образом файлы в каталоге `\FOUND.XXX`. В DOS эти файлы помещаются в корневой каталог ФС под именами `FILEXXXX.CHK` (вместо `XXXX` подставляется номер). Предполагается, что пользователь просматривает все такие файлы и определяет, не содержит ли какой-то из них ценной информации, например конца насильственно укороченного файла.

5. В системах семейства Unix существует несколько специфических ошибок, связанных с инодами:

   - инод, внутренний счетчик ссылок которого не соответствует реальному количеству ссылок из каталогов. Эта проблема может возникать при системном сбое в момент удаления существовавшей связи или создания новой. Она решается коррекцией внутреннего счетчика инода. После этого можно обнаружить следующие две ошибки;
   - инод, не имеющий ни одной ссылки, но не помеченный как свободный – **сирота** (**orphan**). Ссылка на такой инод создается в каталоге `lost+found`;
   - инод с ненулевым количеством ссылок из каталогов, но помеченный свободным. Чаще всего это свидетельствует о порче самого каталога. Обычно ссылки на такой инод удаляются.

В некоторых особенно тяжелых случаях, программа починки оказывается не в состоянии справиться с происшедшей аварией и администратору системы приходится браться за дисковый редактор.

В процессе эксплуатации системы SCO Open Desktop 4.0, у одного из авторов неоднократно возникала необходимость выполнять холодную перезагрузку без нормального размонтирования файловых систем. Одна из таких перезагрузок привела к катастрофе.

Дисковая подсистема машины состояла из кэширующего контроллера дискового массива. Контроллер активно использовал отложенную запись, что, по-видимому и послужило причиной катастрофы. Во время планового резервного копирования драйвер лентопротяжки впал в ступор и заблокировал процесс копирования[12]. Из-за наличия зависшего процесса оказалось невозможно размонтировать файловую систему и машина была перезагружена нажатием кнопки `RESET` без выполнения нормального закрытия, в том числе и без выполнения

[12] Механизм возникновения этой аварии подробнее обсуждался в 7.3.2.

операции закрытия драйвера дискового массива, которая должна была сбросить на диски содержимое буферов контроллера.

После перезагрузки система автоматически запустила программу починки файловой системы `fsck` (File System ChecK), которая выдала радостное сообщение: `DUP in inode 2`. Для незнакомых с системными утилитами Unix необходимо сказать, что DUP означает ошибку перекрещивания файлов, а инод 2 – это инод корневого каталога ФС. Таким образом, корневая директория дискового тома объемом около 2 гигабайт оказалась испорчена. При этом подавляющее большинство каталогов и файлов были не затронуты катаклизмом, но оказались недоступны. Программа починки не могла перенести ссылки на соответствующие иноды в каталог `lost+found`, так как ссылка на него также идет из корневого каталога.

Ситуация представлялась безвыходной. Катастрофа усугублялась тем, что произошла она *во время* резервного копирования, а последняя “хорошая"копия была сделана неделю назад. Бо́льшую часть пользовательских данных можно было бы восстановить, но среди потерянных файлов оказался журнальный файл сервера СУБД ORACLE (сама база данных находилась в другом разделе диска). Пришлось заняться восстановлением ФС с использованием дискового редактора, мотивируя это тем, что “терять всё равно уже нечего". Собственно, в восстановлении ФС участвовало два человека – автор и штатный администратор системы. Автор ни в коем случае не хочет создать у читателя впечатление, что план восстановления был разработан лично им – это был плод совместных усилий, проб и ошибок.

Редактирование системных данных “сложных"ФС с использованием простого шестнадцатиричного дискового редактора является крайне неблагодарным занятием. Есть основания утверждать, что это вообще невозможно. Во всяком случае, ни одному из авторов не доводилось слышать об успехе такого предприятия. К счастью, системы семейства Unix предоставляют для редактирования ФС специальную программу `fsdb` (**File System DeBugger** – отладчик файловой системы). По пользовательскому интерфейсу эта программа напоминает программу `DEBUG.COM`, поставляемую с MS DOS; основным ее преимуществом является то, что она “знакома"с основными понятиями файловой системы. Так, например, `fsdb` позволяет просмотреть содержимое 10-го логического блока файла с инодом 23456, выделить физический блок 567345 файлу с инодом 2 или пометить инод 1245 как свободный.

Первая попытка восстановления состояла в том, что мы удалили тот инод, с которым перекрещивался корневой каталог, смонтировали том командой “безусловного"монтирования (которая позволяла монтировать поврежденные тома), создали командой `mkdir` каталог `lost+found` и вновь запустили `fsck`. Попытка завершилась крахом. Беда была в том, что, как оказалось, корневой каталог пересекался также и со списком свободных блоков, то есть создание каталога, а потом его расширение командой `fsck` снова приводило к порче корневого каталога и задача сводилась к предыдущей.

Таким образом, нам необходимо было либо починить вручную список свободных блоков, либо найти способ создать директорию `lost+found` без обращений к этому списку. Дополнительная сложность состояла в том, что с каталогом `lost+found` не связано фиксированного инода, а определить инод старого `lost+found` не представлялось возможным.

Мы решили не связываться с починкой списка свободны блоков. Вместо этого мы просмотрели листинг последней “правильной"резервной копии, нашли там ненужный пустой каталог и присоединили его к корневому под названием `lost+found`. После этого нам оставалось лишь уповать на то, что вновь создаваемые `fsck` ссылки на файлы не приведут к необходимости удлинить наш `lost+found`. К счастью, этого не произошло: все потомки корневого каталога благополучно получили имена в `lost+found`. По существу, ФС пришла в пригодное для чтения состояние, оставалось лишь правильно определить имена найденных каталогов. Это также оказалось относительно несложной задачей: большая часть каталогов на томе была домашними каталогами пользователей и их имена можно было восстановить на основании того, кому эти каталоги принадлежали. Для остальных каталогов имя достаточно легко определялось после сопоставления их содержимого с листингом резервной копии.

Во многих современных ОС реализованы устойчивые к сбоям файловые системы: jfs в AIX(версия Unix, поставляемая IBM), Merlin), Veritas в UnixWare, NTFS в Windows NT. Практически все такие ФС основаны на механизме, который по-английски называется **intention logging** (**регистрация намерений**).

8.4.3 Файловые системы с регистрацией намерений

Термин, вынесенный в заголовок этого подпункта, является дословной калькой (возможно, не очень удачной) англоязычного термина **intention logging**. *В русском языке, к сожалению, еще нет общепринятого термина для этого понятия.*

Идея журналов регистрации намерений пришла из систем управления базами данных. В СУБД часто возникает задача внесения согласованных изменений в несколько разных структур данных. Например, банковская система переводит миллион долларов с одного счета на другой. СУБД вычитает 1.000.000 из суммы на первом счету, затем пытается добавить ту же величину ко второму счету... и в этот момент происходит сбой.

Для СУБД этот пример выглядит очень тривиально, но мы выбрали его потому, что он похож на ситуацию в файловой системе: в каком бы порядке ни производились действия по переносу объекта из одной структуры в другую, сбой в неудачный момент приводит к крайне неприятной ситуации. В СУБД эта проблема была осознана очень давно и была очень острой – ведь миллион долларов всегда был намного дороже одного сектора на диске...

Удовлетворительное решение проблемы заключается в следующем:

Во-первых, все согласованные изменения в СУБД организуются в блоки, называемые **транзакциями** (**transaction**). Каждая транзакция осуществляется как неделимая (**атомарная**) операция, во время которой никакие другие операции над изменяемыми данными не разрешены.

Во-вторых, каждая транзакция осуществляется в три этапа.

1. Система записывает в специальный журнальный файл, что же она собирается делать.

2. Если запись в журнал была успешной, система выполняет транзакцию.

3. Если транзакция завершилась нормально, система помечает в журнале, что намерение было успешно реализовано.

Журнал часто называют **журналом регистрации намерений** (**intention log**), что очень хорошо отражает суть дела, потому что в этот журнал записываются именно **намерения** (**intentions**).

При использовании отложенной записи транзакция считается полностью завершенной, только когда последний блок измененных данных будет физически записан на диск. При этом в системе обычно будет одновременно существовать несколько недовыполненных транзакций. Легко понять, что при операциях с самим журналом отложенную запись вообще нельзя использовать.

Если произошел сбой системы, после перезагрузки запускается программа починки базы данных. Эта программа просматривает конец журнала.

- Если она видит там испорченную запись, то игнорирует ее: сбой произошел во время записи в журнал.
- Если все записи помечены как успешно выполненные транзакции, то сбой произошел между транзакциями: ничего особенно страшного не случилось; во всяком случае, ничего чинить не надо.
- Если найдена запись, которая отмечает начатую, но невыполненную транзакцию, то сбой произошел *во время* этой транзакции. Это наиболее неприятная ситуация, но журнал содержит достаточно информации, чтобы или восстановить состояние базы данных до начала транзакции (выполнить **откат назад** (**rollback**)), или же доделать предполагавшиеся изменения.

Нужно отметить, что данные, необходимые для выполнения отката, могут иметь большой объем. Фактически это копия всех данных, подвергающихся изменению в ходе транзакции. Многие СУБД, такие как ORACLE, хранят эти данные не в самом журнале, а в специальной области данных, называемой **сегмент отката** (**rollback segment**).

Более подробная информация о работе журналов намерений в базах данных может быть найдена в соответствующей литературе ([18] и др.). Необходимо только отметить, что книги и даже фирменная документация по простым СУБД типа dBase или FoxPro здесь не помогут, поскольку эти пакеты не содержат средств журналирования.

Легко понять, что идея журнала намерений достаточно естественно переносится в программу управления файловой системой. Но здесь возникает интересный вопрос – что же считать транзакцией: только операции по распределению пространства на диске или также все операции по изменению данных?

Первый вариант проще в реализации и оказывает меньшее влияние на производительность; зато он гарантирует только целостность самой ФС, но не может гарантировать целостности пользовательских данных, если сбой произойдет в момент записи в файл.

Второй вариант требует выделения сегмента отката и сильно замедляет работу. Действительно, ведь теперь данные пишутся на диск два раза: сначала в сегмент отката, а потом в сам файл. Зато он существенно снижает вероятность порчи пользовательских данных.

Ряд современных ФС с регистрацией намерений поддерживают оба режима работы и предоставляют выбор между этими вариантами администратору системы. Например, у файловой системы `vxfs` или Veritas, входящей в пакет UnixWare (версия Unix SVR4, поставляемая фирмой Novell), существует две версии. Одна версия, поставляемая вместе с системой по умолчанию, включает в транзакцию только системные данные. Другая, "advanced", версия, которая поставляется за отдельные деньги, осуществляет регистрацию намерений как для системных, так и для пользовательских данных.

В Veritas 2 дисковый том разбит на области, называемые **группами цилиндров** (**термин, унаследованный из** UFS – **файловой системы** BSD Unix). Каждая группа цилиндров имеет свою карту свободных блоков, участок динамической таблицы инодов и журнал намерений. Журнал намерений организован в виде кольцевого буфера записей о транзакциях. В журнал пишутся данные только о транзакциях над инодами, принадлежащими к этой группе цилиндров. При этом в каждой группе может исполняться одновременно несколько транзакций и окончанием транзакции считается физическое завершение записи модифицированны данных. Количество одновременно исполняющихся транзакций ограничено объемом журнала, но поскольку каждая группа цилиндров имеет свой журнал, это ограничение не играет большой роли.

8.4.4 Устойчивость ФС к сбоям диска

Кроме общесистемных сбоев, ФС должна обеспечивать средства восстановления при физических сбоях диска. Наиболее распространенным видом таких сбоев являются нечитаемые – **"плохие"** (**bad**) – блоки, появление которых обычно связано с физическими дефектами магнитного носителя.

Быстрее всего плохие блоки возникают на гибких магнитных дисках, которые соприкасаются с головкой чтения/записи и из-за этого подвержены физическому износу и повреждениям. Кроме того, гибкие диски подвергаются опасным воздействиям и вне дисковода. Например, при вносе дискеты с улицы в теплое помещение на поверхности диска будет конденсироваться влага, а соприкосновение головки дисковода с влажным диском практически наверняка повредит магнитный слой.

Жесткие магнитные диски помещены в герметичный корпус и не соприкасаются с головками дисковода, поэтому срок службы таких дисков намного больше. Появление одиночных плохих блоков на жестком диске скорее всего свидетельствует о заводском дефекте поверхности или же о том, что магнитный слой от старости начал деградировать.

Еще одной причиной порчи жестких дисков является соприкосновение головок чтения/записи с поверхностью вращающегося диска (**head crash**). Обычно это приводит к повреждению целой дорожки или нескольких дорожек диска, а зачастую и самой головки. Для несъемных жестких дисков в герметичных корпусах (т.наз. винчестерских дисков) это означает потерю целой рабочей поверхности.

Обычно ошибки данных обнаруживаются при чтении. Дисковые контроллеры используют при записи кодировку с исправлением ошибок, чаще всего, коды типа **CRC** (**Cyclic Redundancy Code** – циклический избыточный код), которые позволяют обнаруживать и исправлять множественные ошибки. Тем не менее, если при чтении была обнаружена ошибка, большинство ОС отмечают такой блок как плохой, даже если данные удалось восстановить на основании избыточного кода.

В файловой системе FAT плохой блок или кластер, содержащий такой блок, отмечается кодом `0xFFB` или `0xFFFB` для дисков с 16-разрядной `FAT`. Эта файловая система не способна компенсировать плохие блоки в самой `FAT` или в корневом каталоге диска. Такие диски просто считаются непригодными для использования.

В "сложных" файловых системах обычно используется более сложный, но зато и более удобный способ обхода плохих блоков, называемый **горячей заменой** (**hotfixing**). При создании файловой системы отводится небольшой пул блоков, предназначенных для горячей замены. В файловой системе хранится список всех обнаруженных плохих блоков, и каждому такому блоку поставлен в соответствие блок из пула горячей замены. При этом плохие блоки, на которые оказались отображены системные структуры данных, например участок таблицы инодов, также подвергаются горячей замене. Таблица горячей замены может быть как статической, так и динамической.

На первый взгляд, динамическая таблица горячей замены предпочтительна, однако не нужно забывать о двух немаловажных факторах.

- Файловая система вынуждена справляться с таблицей горячей замены при всех обращениях к диску, поэтому увеличение таблицы приводит к замедлению работы.
- Множественные плохие блоки на жестком диске свидетельствуют либо о том, что диск дефектный, либо о том, что магнитный слой начал разрушаться от старости (иными словами, что этот диск пора выкидывать), либо о каких-то других не менее серьезных проблемах, вроде разгерметизации корпуса и проникновения в него пыли.

С учетом обоих факторов кажется целесообразным установить предел количества плохих блоков, после достижения которого диск нуждается в замене. Нужно отметить также, что этот предел не может превышать нескольких процентов общей емкости диска. В свете этого небольшая статическая таблица блоков горячей замены представляется вовсе не такой уж плохой идеей.

8.5 Драйверы файловых систем

При эксплуатации ОС может возникнуть необходимость монтировать файловые системы, отличающиеся от "родной" ФС. Особенно часто она возникает в организациях, где используются ОС нескольких разных типов. Да и в организациях, работающих с монокультурой MS DOS/MS Windows, такая потребность возникает все чаще. Во-первых, доступ к файлам на файловом сервере осуществляется существенно иными способами, чем к файлам на локальном диске, даже если на сервере стоит та же ДОС с той же ФС типа FAT. Во-вторых, дисководы для CD-ROM становятся все дешевле и распространяются все шире. При этом стандартная ФС на CD-ROM – вовсе не FAT.

Решение этой проблемы приходит в голову сразу – необходим драйвер файловой системы со стандартным интерфейсом, подобный драйверу внешнего устройства. Естественно, набор функций такого драйвера должен быть существенно иным:

mount – монтирование ФС. В зависимости от типа ФС параметры этой функции должны различаться. Для ФС на локальных дисках достаточно передать системный идентификатор монтируемого диска. Для ФС на удаленной машине мы должны передать сетевой адрес этой машины и имя требуемой файловой системы. Во многих случаях проводится различие между монтированием ФС для чтения/модификации или только для чтения: при монтировании с модификацией устанавливается флаг загрязнения (dirty flag) ФС);

umount – размонтирование ФС;

GetFreeSpace или df – получение информации о ФС: общее пространство, свободное пространство, количество файлов и т.д;

FindFirst/FindNext или opendir/readdir – функции для чтения из каталога. Считанную информацию необходимо привести к формату, принятому в данной ОС. В частности, может оказаться необходимым сократить и/или преобразовать считанные имена файлов. Например, драйвер HPFS, используемый в эмуляторе MS DOS, в системе OS/2 не выполняет такого преобразования. В результате DOS-овские программы не видят файлов и каталогов с длинными именами;

access – проверить существование файла и возможность доступа к нему в заданном режиме, например, для чтения или для записи;

stat – функция для получения информации о файле с заданным именем. Драйвер ФС должен считать доступную информацию о файле и привести ее к принятому в ОС формату. При этом, возможно, придется проигнорировать часть считанной информации, а другую часть, напротив, драйвер вынужден сочинять сам. Так драйвер файловой системы FAT в ОС семейства Unix вынужден сочинять идентификатор хозяина файла, права доступа и т.д.;

open – открыть существующий или создать новый файл с заданным именем;

read и write – функции считывания данных из файла и записи в него;

lseek – позиционирование в файле;

lock – функция (в действительности, набор функций) блокировки для синхронизации доступа к файлу или его участкам. Подробнее о них см. в п. 4.3.3. При работе с локальными ФС операции блокировки могут прослеживаться ядром системы и не доходить до драйвера ФС, но при работе с разделяемыми по сети ФС блокировки необходимо отслеживать на уровне протокола разделения файлов, поэтому драйвер сетевой ФС обязан знать о блокировках.

close – закрыть файл;

delete или unlink – удалить файл или его имя;

link – создать связь с файлом (новое имя). Далеко не все ОС и ФС поддерживают эту операцию;

mkdir – создать каталог;

rmdir – удалить каталог. Обычно разрешено удаление только пустых каталогов.

Кроме собственно драйвера ФС, для ее полноценной поддержки нужны следующие программы:

- программа создания ФС – `mkfs` или `FORMAT`;
- программа контроля и починки ФС – `fsck` или `CHKDSK`;
- программа fstyp, которая смотрит на диск и пытается определить, "её"это диск или не её. Она полезна при монтировании ФС с автоматическим определением ее типа;
- программа mount, которая принимает из командной строки зависящие от типа ФС параметры, проверяет их допустимость и инициализирует драйвер ФС.

Например, дистрибутив ОС UnxiWare 2.0 фирмы SCO, основанной на UNIX System V R4.2, содержит драйверы следующих файловых систем:

memfs – файловая система, размещающая файлы в оперативной памяти. Может рассматриваться как эквивалент виртуального диска в MS DOS.

dosfs – файловая система FAT.

XENIX – файловая система, используемая ОС Xenix. Эта ФС по структуре похожа на остальные ФС ОС семейства Unix, но не поддерживает символических связей и ограничивает длину имени файла 12 символами. Неустойчива к сбоям.

s5 – "классическая" ФС, сохранившаяся почти без изменений с самых ранних версий системы – s5, по-видимому, означает Unix System 5. Ограничивает имя файла 14 символами. Неустойчива к сбоям.

ufs – файловая система, разработанная в университете Беркли, известная также как FFS (Fast File System) и Berkley FS. Является основной ФС в большинстве версий BSD UNIX и поддерживается многими другими ОС семейства Unix. Имеет более высокую производительность, чем s5, в первую очередь за счет разбиения таблицы инодов и списка свободных блоков на участки (**группы цилиндров**). Такое разбиение уменьшает перемещение головки дисковода при обращении к системным и пользовательским данным. Поддерживает **дисковые квоты** – ограничения на объем дискового пространства, занятого файлами того или иного пользователя. Ограничивает имя файла размером блока (обычно 512 символов). Неустойчива к сбоям.

bfs – Boot File System – загрузочная файловая система. Эта ФС имеет очень простую структуру, отчасти похожую на файловую систему RT-11: все файлы в ней обязаны занимать непрерывное пространство. Такая структура упрощает первичный загрузчик системы, которому теперь не нужно разбираться в каталогах и инодах. bfs имеет довольно низкую производительность и требует длительной процедуры размонтирования, если в нее были записаны новые данные. Фактически, при таком размонтировании происходит операция сжатия ФС, эквивалентная команде `SQEESE` в RT-11. Используется для хранения ядра системы, и нескольких конфигурационных файлов, используемых при загрузке. Все эти данные считываются только при загрузке системы и перезаписываются только при изменениях конфигурации ядра, поэтому высокая производительность от этой ФС не требуется.

vxfs – устойчивая к сбоям ФС Veritas с регистрацией намерений. Версия, входящая в стандартную поставку системы, журналирует только системные структуры данных. За отдельную плату можно приобрести "продвинутую" версию Veritas, которая обеспечивает журналирование пользовательских данных.

cdfs – Файловая система ???, используемая на CD-ROM.

nfs – Network File System – драйвер файловой системы, обеспечивающий разделение файлов с использованием сетевого протокола TCP/IP. Протокол NFS был предложен фирмой Sun Microsystems в середине 80-х гг. и в настоящее время поддерживается практически всеми членами семейства Unix. NFS-клиенты и NFS-серверы реализованы также для VMS, Windows NT, Novell Netware, OS/2, MacOS, MS/DR DOS и Windows 3.x/95.

rfs – Remote File Sharing – использование удаленной UNIX-системы в качестве файлового сервера. Этот протокол был разработан фирмой AT&T в 80-е гг. и пригоден только для соединения систем UNIX System V.

nucfs NetWare Unix Client File System. Этот драйвер предназначен для присоединения к файловым серверам Novell Netware. Он входит в состав системы UnixWare, поставляемой фирмой SCO, но не в остальные версии UNIX SVR4.

.

Любопытно, что даже MS/DR DOS версий выше 3.30 имеют возможность устанавливать драйвер файловой системы. Такой драйвер может быть реализован путем перехвата недокументированных функций прерывания `0x2F` – группы функций "Network Redirector" и "IFS". Таким образом реализован ряд сетевых клиентов для MS/DR DOS. К сожалению, авторы не смогли найти полноценного описания этих функций.

Глава 9

Безопасность

Just cause you're paranoid don't mean they aren't after you.

Nirvana.

То, что я параноик, еще не означает, что никто не намерен причинить мне вред.

Нирвана.

Проблема защиты пользовательских данных от нежелательного прочтения или модификации встает очень часто и в самых разнообразных ситуациях – от секретных баз данных Министерства обороны до архива писем к любимой женщине. Причин, по которым пользователь может желать скрыть или защитить свои данные от других, может быть очень много, и в подавляющем большинстве случаев эти причины достойны уважения. Нужно отметить также, что по мере компьютеризации общества в электронную форму переносится все больше и больше данных, конфиденциальных по своей природе: банковские счета и другая коммерческая информация, истории болезни и т.д.

Защита данных практически не имеет смысла без защиты самой системы и прикладного программного обеспечения: если злоумышленник имеет возможность модифицировать код системы или прикладной программы, он может встроить в него "троянские" подпрограммы, осуществляющие несанкционированный доступ к данным. Кроме того, следует учитывать возможность проникновения в систему с целью расстроить ее работу – из простого спортивного интереса, с целью диверсии или же чтобы отвлечь внимание администратора от несанкционированного доступа к данным.

Часто взлом систем и написание вирусов ради развлечения приравнивают к обычному вандализму. С точки зрения авторов, эти действия все-таки отличаются от битья стекол или отрывания трубок у телефонов-автоматов: вандализм представляет собой акт чистого разрушения, в то время как взлом компьютерной системы требует некоторой интеллектуальной деятельности, которую при желании можно расценить как созидательную. Впрочем, с точки зрения пострадавших разницы между этими дей-

ствиями практически нет: эмоции пользователя, у которого пропал день работы в результате вирусной активности, мало отличаются от эмоций человека, который не может вовремя позвонить по телефону, и сильнее всего в обоих случаях угнетает явная бессмысленность вредоносного действия.

Когда говорят о безопасности, всегда рассматривают задачи защиты как данных, так и программного обеспечения в комплексе. В некоторых случаях программное обеспечение можно рассматривать как особую форму данных и решать проблему его защиты теми же средствами, что и для данных.

Особой защитой всегда окружаются код системы и ее конфигурационные данные. Это необходимо не только для того, чтобы защититься от несанкционированного доступа, но и для защиты от повреждения системы "санкционированным" пользователем по ошибке.

В популярных публикациях, посвященных теме компьютерной безопасности, часто приходится видеть аналогию с известным соревнованием снаряда и брони: после появления более мощной брони всегда появляются способные ее преодолеть снаряды и так далее до бесконечности. На взгляд авторов, такая аналогия весьма неудачна.

Дело в том, человечеству неизвестно идеальной – абсолютно непробиваемой – брони, но очень легко можно представить себе абсолютно непроницаемую систему защиты данных: это система, в которой данные недоступны вообще никому и ни при каких обстоятельствах. Понятно, что практической ценности такая система не имеет, но проблема несанкционированного доступа в этом случае решена полностью.

Чтобы придать системе практическую ценность, необходимо кому-то и при каких-либо условиях все-таки разрешить доступ. Понятно, что такой отход от идеала делает систему потенциально уязвимой для атак. Понятно также, что все атаки должны быть нацелены на какое-то несовершенство системы защиты – нет и не может быть универсального типа атаки, способного преодолеть любую защиту. Даже расшифровка криптограмм методом "грубой силы" – полного перебора – как минимум требует получения доступа к зашифрованным данным, что может оказаться невозможным.

Сказанное справедливо лишь для систем, безопасность которых строится по принципу "запрещено все, кроме того, что было разрешено явным образом". По этому принципу строится защита практически всех современных систем общего назначения. В таких системах несанкционированный доступ может происходить лишь из-за несовершенства средств защиты и в каждом случае метод доступа оказывается основан на вполне конкретном несовершенстве.

Если же строить защиту на принципе "разрешено все, кроме того, что запрещено" , то аналогия с броней и снарядом оказывается вполне актуальна: разработчик системы безопасности перекрывает лишь пути, до которых дошла его фантазия, все же остальное пространство методов доступа оказывается неперекрыто. Характерным примером может служить эволюция антивирусных средств для DOS и параллельная эволюция технологий разработки вирусов.

Совершенствование средств доступа к данным и их разделения между пользователями всегда порождает и дополнительные возможности несанкционированного доступа. Наиболее ярким примером являются современные глобальные сети, которые предоставляют доступ к огромному богатству информационных сред. Эти же сети представляют серьезную угрозу безопасности подключающихся к сети организаций. Основная специфика угрозы заключается в том, что злоумышленнику значительно легче обеспечить свою анонимность даже в случае обнаружения факта взлома. Поэтому в наше время глобальных открытых информационных систем проблемы безопасности приобретают особую важность.

Идеальная система безопасности должна обеспечивать минимальные трудности для санкционированного доступа к данным и непреодолимые для несанкционированного. Кроме того, она должна обеспечивать легкую и гибкую систему управления санкциями; во многих случаях бывает также полезно отслеживать все попытки несанкционированного доступа.

Большинство современных многопользовательских систем оказываются весьма близки к идеалу – используемые модели выдачи санкций являются теоретически “непробиваемыми". К сожалению, практические реализации этих моделей подчас оказываются несовершенными и содержат ошибки.

Широко известна известна ошибка в программое `fingerd`, входившей в систему BSD Unix. Эта программа принимала данные в буфер, выделяемый в локальном стеке, не проверяя, входит ли принимаемая порция данных в буфер. Искусственно вызванное переполнение могло привести к перезаписи стекового кадра процедуры, которое в свою очередь при возврате из подпрограммы могло привести к передаче управления в нежелательное место.

“Червяк Морриса” ([23]) пользовался этой дырой, передавая кусок кода и поддельный стековый кадр, передававший управление на этот код. Код в свою очередь запускал на целевой системе копию командного интерпретатора, исполнявшуюся с привилегиями суперпользователя, Затем червь использовал полученный командный интерпретатор для втягивания в систему своего тела.

Ошибка была обнаружена в 1987 г. и вскоре после обнаружения была исправлена. Практически все современные системы используют безопасную в этом отношении версию `fingerd`.

Аналогичная ошибка была обнаружена в 1995 году в программе `syslogd` некоторых коммерческих версий Unix.

В обоих случаях ошибка связана с использованием библиотечной функции `gets(char * buf)`, которая принимает ограниченную символом `'\n'` строку данных из стандартного потока ввода, не проверяя буфер на переполнение. В настоящее время функция `gets` поддерживается лишь для совместимости и использовать ее не рекомендуется. Вместо нее стандарты ANSI C и POSIX рекомендуют использовать функцию `fgets(FILE * stream, size_t bufsize, char * buf);`, которая осуществляет проверку на переполнение.

В январе 1996 года была обнаружена проблема, создаваемая одним из антивирусных пакетов для Windows NT. Этот пакет предназначен не для ловли вирусов в самой NT, потому что аналоги вирусов ДОС в защищенных многопользовательских ОС практически нежизнеспособны. Он предназначен для использования на системах, работающих в качестве файлового сервера и хранящих на своих дисках прикладные программы для DOS/MS Windows, используемые станциями локальной сети.

Рабочая часть пакета состоит из процесса-демона (или, по терминологии NT, **сервиса** (**service**)), который периодически сканирует исполняемые файлы на дисках с целью поиска в них сигнатур известных вирусов. Сервисный процесс исполняется с привилегиями администратора для того, чтобы установленные пользователями атрибуты защиты от чтения или записи не мешали сканированию или "лечению" зараженных файлов.

При установке пакета программа-инсталлятор создавала в системе пользователя с административными привилегиями и генерировала для него пароль. Проблема состояла в том, что этот пароль было относительно легко угадать. При этом ни программа установки, ни документация ничего не сообщали администратору о создании пользователя, поэтому практически во всех случаях администраторы не меняли автоматически сгенерированный пароль.

Администратор любой многопользовательской системы обязан следить за появляющимися сообщениями об обнаруженных ошибках и "дырах"в системе безопасности и принимать меры для устранения и компенсации возникающих проблем.

Обычно выдача санкций (далее мы будем обычно использовать словосочетание **права доступа**) осуществляется на основе пользовательских идентификаторов. Каждому пользователю системы выделяется идентификатор. Обычно такой идентификатор имеет две формы: числовой код, используемый внутри системы, и мнемоническое символьное имя, используемое при общении с пользователем.

Например, в системах семейства Unix пользователь индентифицируется целочисленным значением **uid** (**user identifier**). Пользователь может иметь также символьное имя. Соответствие между символьным и числовым идентификаторами устанавливается на основе содержимого текстового файла `/etc/passwd`. Каждая строка этого файла описывает одного пользователя и состоит из семнадцати полей, разделенных символом `':'`. В первом поле содержится символьное именя пользователя, во втором – числовой идентификатор в десятичной записи. Остальные поля содержат другие сведения о пользователе, например, его полное имя.

Пользовательские программы могут устанавливать соответствие между числовым и символьным идентификаторами самостоятельно, путем просмотра файла `/etc/passwd`, или использовать библиотечные функции, определенные стандартом POSIX. Во многих реализациях эти функции используют вместо `/etc/passwd` индексированную базу данных, а сам файл `/etc/passwd` сохраняется лишь для совместимости со старыми программами.

Нужно отметить, что соответствие между символьным и числовым идентификаторами в Unix не является взаимно однозначным. Одному и тому же числовому идентификатору может соответствовать несколько имен. Кроме того, в Unix можно создать объекты с числовым uid, которому не соответствует никакого символьного имени.

Чаще всего используется принцип **сессий** работы с машиной. В начале работы пользователь устанавливает соединение и "входит" в систему. При "входе" происходит его идентификация. Затем пользователь проводит сеанс работы с системой, а по завершении этого сеанса разрегистрируется. Сеанс или сессия жестко связаны с определенным терминалом и/или определенной машиной в сети. При работе в сети есть определенные нюансы, но они будут обсуждаться ниже.

Большинство современных ОС позволяют также запускать задания без входа в систему и создания сессии. Так, системы семейства Unix предоставляют возможность пользователям запускать задачи в заданные моменты астрономического времени или периодически, например, в час ночи в пятницу каждой недели. Каждая такая задача исполняется от имени определенного пользователя – того, кто запросил запуск задачи. Для управления правами доступа в таких ситуациях идентификатор пользователя ассоциируется не с сессией, а с отдельными заданиями, а обычно даже с отдельными задачами.

Так, в системах семейства Unix с каждой задачей (процессом) связано два идентификатора пользователя: **реальный** и **эффективный**. В большинстве ситуаций эти идентификаторы совпадают. Таким образом, каждая задача обязательно исполняется от имени определенного пользователя.

Большинство современных ОС общего назначения также связывают с каждой задачей идентификатор пользователя или **контекст доступа**security context. Исключением в этом отношении является OS/2, в которой такого связывания не происходит. OS/2 позволяет ассоциировать пользовательские идентификаторы и права доступа с файлами, но не с задачами. Идентификаторы хозяина файла используются программами сетевого разделения файлов, например IBM LAN Server, но вся проверка прав доступа и идентификация пользователя происходит внутри задачи-сервера, а не на уровне ОС.

Отсутствие раздельных контекстов доступа у задач не позволяет использовать OS/2 как многопользовательскую систему. Это дает основание многим критикам трактовать название системы как "ОС пополам": система, в которой отсутствует половина функций, требуемых от современной ОС общего назначения.

9.1 Идентификация пользователя

Петли дверные
Многим скрипят, многим поют:
"Кто вы такие,
Вас здесь не ждут!"

В. Высоцкий.

Понятно, что если права доступа выделяются на основе машинного идентификатора пользователя, то возникает отдельная проблема установления соответствия между этим машинным идентификатором и реальным человеком. Такое соответствие не обязано быть взаимно однозначным: один человек может иметь несколько идентификаторов или наоборот, несколько человек могут пользоваться одним идентификатором. Тем не менее способ установления такого соответствия необходим.

По-английски процесс входа в систему называется **login** (**log in**) и происходит от слова **log**, которое обозначает регистрационный журнал или процесс записи в такой журнал. Наиболее точным переводом слова **login**[1] является **регистрация**. Соответственно, процесс выхода называется **logout**. Его точная русскоязычная калька – **разрегистрация** – к сожалению, очень неблагозвучна

Теоретически можно придумать много разных способов идентификации, например, с использованием механических или электронных ключей или даже тех или иных билогических параметров, например, рисунка глазного дна. Однако все такие способы требуют специальной и зачастую довольно дорогой аппаратуры. Наиболее широкое распространение получил более простой метод, основанный на символьных паролях.

Пароль представляет собой последовательность символов. Предполагается, что пользователь запоминает ее и никому не рассказывает. Этот метод хорош тем, что для его использования не нужно никакого дополнительного оборудования – только клавиатура, которая и без того необходима. Но этот метод имеет и ряд недостатков.

Использование паролей основано на следующих трех предположениях:

- пользователь может запомнить пароль;
- никто не сможет догадаться, какой пароль был выбран;
- пользователь никому не расскажет свой пароль.

Последнее предположение кажется разумным: если человек может добровольно кому-то рассказать свой пароль, с примерно той же вероятностью этот человек может сам сделать пакость. Впрочем, человека можно *заставить* рассказать пароль... Однако защита от таких ситуаций требует особых мер, которые не могут быть обеспечены на уровне операционной системы.

Если вдуматься, первые два требования отчасти противоречат друг другу. Количество легко запоминаемых паролей ограничено, и перебрать все такие пароли оказывается не так уж сложно. Известный в истории сети Internet "червяк Морриса" [2] [23]. использовал при подборе паролей следующие варианты:

- входное имя пользователя;
- входное имя с символами в обратном порядке – "задом наперед";
- компоненты полного имени пользователя – имя (first name), фамилию (last name) и другие компоненты, если они есть;

[1] В обычном английском языке такого слова нет, но в компьютерной лексике слова **login** и **logout** прижились очень прочно.

[2] "Червяк Морриса" заслуживает отдельного и весьма подробного обсуждения. Эта программа в 1987 г. поразила несколько сотен узлов сети Inernet. Червь производил комплексную и довольно хорошо продуманную атаку, продемонстрировавшую не только специфические изъяны безопасности тогдашних ОС семейства Unix, но и несколько фундаментальных проблем компьютерной безопасности. По мнению авторов, серьезное изучение проблем компьютерной безопасности в наше время невозможно без учета опыта "червя Морриса".

- те же компоненты, но задом наперед;
- слова из заранее определенной таблицы из 400 элементов; Большая часть успешно подобранных паролей была взята из этой таблицы. После поимки червя таблица была опубликована с тем, чтобы пользователи не повторяли старых ошибок[3].

С первого взгляда видно, что таких "легко запоминаемых" вариантов довольно много – никак не меньше нескольких сотен. Набор их вручную занял бы очень много времени, но для компьютера несколько сотен вариантов – это доли секунды. Хотя... Первый слой защиты от подбора пароля заключается именно в том, чтобы увеличить время подбора. Обнаружив неправильный пароль, наученные горьким опытом современные системы делают паузу, прежде чем позволят повторную попытку входа с той же линии. Такая пауза может длиться всего одну секунду, но даже этого достаточно, чтобы превратить процесс подбора пароля из долей секунды в десятки минут.

Второй слой защиты заключается в том, чтобы усложнить пароль и тем самым увеличить количество вариантов. Даже очень простые усложнения сильно увеличивают перебор. Так, простое требование использовать в пароле буквы и верхнего и нижнего регистров увеличивает перебор в 2^n раз, где n – это длина пароля. В большинстве современных систем пароль обязан иметь длину хотя бы шесть символов, т.е. количество вариантов увеличивается в 64 раза. Требование использовать в пароле хотя бы один символ, не являющийся буквой, увеличивает число вариантов в $6 \times 43 = 258$ раз (в наборе ASCII 43 небуквенных графических символа). Вместо десятков минут подбор пароля, который обязан содержать буквы разных регистров и хотя бы один спецсимвол, займет много дней.

Но если взломщику действительно *нужно* попасть в систему, он может подождать и несколько дней, поэтому необходим третий слой защиты – ограничение числа попыток. Все современные системы позволяют задать число неудачно набранных паролей, после которого имя блокируется. Это число всегда больше единицы, потому что пользователь – это человек, а людям свойственно ошибаться, но в большинстве случаев такой предел задается не очень большим – обычно 5 - 7 попыток. Однако, этот метод имеет и оборотную сторону – его можно использовать для блокировки пользователей.

Интересный вариант того же метода заключается в том, чтобы увеличивать паузу между последовательными неудачными попытками хотя бы в арифметической прогрессии.

Наконец, последний слой защиты – это оповещение пользователя (а иногда и администратора системы) о неудачных попытках входа. Если пользователь сам только что нажал не ту кнопку, он при входе увидит, что была одна неудачная попытка, и не будет беспокоиться; однако, если есть сообщения о дополнительных неудачных попытках, время побеспокоиться и разобраться, что же происходит.

Современная техника выбора паролей обеспечивает достаточно высокую для большинства практических целей безопасность. В случаях, где эта безопасность представляется недостаточной – напри-

[3]Эта таблица приводится в [23]. По мнению авторов, включение ее в данное издание нецелесообразно, поскольку она рассчитана на англоязычных пользователей и не очень актуальна в России.

мер, если следует всерьез рассматривать опасность принуждения пользователя к раскрытию пароля – можно применять методы идентификации, перечисленные выше, т.е. основанные на механических или электронных ключах или биометрических параметрах. Впрочем, как уже говорилось, такие методы требуют использования специальной аппаратуры.

При использовании паролей возникает отдельная проблема безопасного хранения базы данных со значениями паролей. Как правило, даже администратор системы не может непосредственно получить значения паролей пользователей. Можно привести несколько соображений в пользу такого ограничения.

- Если бы администратор системы знал пользовательские пароли, то взломщик, сумевший выдать себя за администратора, так же получал бы доступ к этим паролям. После этого взломщик мог бы легко маскироваться под других пользователей, что сильно усложнило бы обнаружение взлома.
- Если администратор знает только административный пароль, для лишения его административных привилегий достаточно сменить этот пароль. Если же бывший администратор имел доступ ко всей базе данных о паролях, он по прежнему будет иметь возможность доступа к системе, что может оказаться нежелательным.

Для обеспечения секретности паролей обычно используют одностороннее шифрование, при котором по зашифрованному значению нельзя восстановить исходное слово. При этом программа аутентикации кодирует введенный пароль и сравнивает полученное значение с хранящимся в базе данных. При использовании достаточно сильных алгоритмов шифрования, например DES, узнать реальное значение пароля можно только путем полного перебора всех возможных значений и сравнения зашифрованной строки со значением в базе данных.

Отцам-основателям Unix этот механизм обеспечения секретности показался настолько надежным, что они даже не стали ограничивать доступ обычных пользователей к базе данных, и выделили для хранения зашифрованных паролей поле в общедоступном для чтения файле `/etc/passwd`.

Однако "червь Морриса" продемонстрировал, что для подбора паролей не нужно перебирать все возможные комбинации символов, поскольку пользователи имеют тенденцию выбирать лишь ограниченное подмножество таких комбинаций. Пользователь, имеющий возможность непосредственно читать закодированные значения паролей, может осуществлять подбор, не обращаясь к системным механизмам аутентикации, то есть делать это очень быстро и практически незаметно для администратора. Этот тип атаки называется **словарной атакой** (**dictionary attack**) и весьма опасен для систем, где пользователи неосторожны в выборе паролей.

После осознания опасности словарных атак разработчики систем семейства Unix перенесли значения паролей в недоступный для чтения файл `/etc/shadow`, где пароли также хранятся в односторонне зашифрованном виде. Это значительно усложняет реализацию словарной атаки, поскольку перед тем, как начать атаку, взломщик должен получить доступ к системе с привилегиями администратора.

Практически все современные системы хранят данные о паролях в односторонне зашифрованном виде в файле, недоступном обычным пользователям для чтения. Поставщики некоторых систем, например Windows NT, даже отказываются публиковать информацию о формате этой базы данных, хотя это само по себе вряд ли способно помешать квалифицированному взломщику.

9.2 Идентификация пользователя в сети

Когда пользователь работает с консоли компьютера или с терминала, физически прикрепленного к терминальному порту (**hardwired**), модель сессий является вполне приемлемой. При регистрации пользователя создается сессия, ассоциированная с данным терминалом, и далее проблем нет.

Аналогично нет никаких проблем при подключении через сеть с коммутацией каналов, например, при дозвоне через модем, подключенный к телефонной сети. Когда соединение разрывается, сессия считается оконченной.

Напротив, в сетях с коммутацией пакетов, к которым относится большинство современных сетевых протоколов (TCP/IP, IPX/SPX, ISO/OSI и т.д.), нет физического понятия соединения. В лучшем случае сетевой протокол предоставляет возможность создавать виртуальные соединения с "надежной" связью, в которых гарантируется отсутствие потерь пакетов и сохраняется порядок их поступления. С таким виртуальным соединением вполне можно ассоциировать сессию, как это сделано в протоколах telnet, rlogin/rsh и ftp.

Протокол telnet используется для эмуляции алфавитно-цифрового терминала через сеть. Пользователь устанавливает соединение и регистрируется в удаленной системе таким же образом, как он регистрировался бы с физически подключенного терминала. Например, в системах семейства Unix создается виртуальное устройство, **псевдотерминал** (**pseudoterminal**) `/dev/ptyXX`, полностью эмулирующее работу физического терминала, и система запускает ту же программу идентификации пользователя `/usr/bin/login`, которая используется для физических терминалов. При окончании сессии соединение разрывается, и псевдотерминальное устройство освобождается.

Виртуальные сессии вынуждены полагаться на то, что сетевое программное обеспечивает действительно гарантированное соединение, т.е., что никакая "левая" машина не может вклиниться в соединение и прослушать его или послать свои поддельные пакеты. Обеспечение безопасности на этом уровне представляет собой отдельную проблему, обсуждение которой увело бы нас далеко от основной темы.

В протоколах, использующих датаграммные соединения, средствами протокола виртуальную сессию создать невозможно. Обычно в этом случае каждый пакет содержит идентификатор пользователя или сессии, от имени которого (в рамках которой) этот пакет послан. Такой подход используется в протоколах работы с файловыми серверами NFS, и NCP (Netware Core Protocol, используемый файловыми серверами Novell Netware). Понятно, что чисто датаграммные протоколы оказываются гораздо более уязвимы для поддельных пакетов и прочих вредоносных воздействий.

Например, NCP, работающий через датаграммный протокол IPX, реализует свой собственный механизм поддержки сессий: все пакеты данной сессии имеют 8-битный номер и пакеты с неправильными номерами просто игнорируются. Одна из программ взлома Netware в течении короткого времени посылала 256 пакетов с различными номерами – хотя бы один пакет оказывался удачным. Для борьбы с такими действиями в Netware 3.12 была введена криптографическая подпись пакетов в пределах сессии.

Проблема дополнительно усложняется тем обстоятельством, что в сети зачастую взаимодействие происходит между системами, каждая из которых позволяет одновременную работу нескольких пользователей. Часто возникает желание требовать от пользователя регистрации только в одной из систем, а доступ в остальные системы предоставлять автоматически.

Интересное решение этой задачи предлагается при регистрации на сервере Novell Netware 3.x: при входе пользователя на сервер исполняется специальный командный файл, так называемый **login script**. Если в нем содержится команда присоединения к другому серверу, программа сначала пробует тот же пароль, который был задан при входе на первый из серверов. Только если этот пароль не подходит, система требует у пользователя ввода пароля. Этот метод обеспечивает одновременную регистрацию на нескольких серверах, но не подходит для динамического создания и разрыва соединений.

Обычно для автоматической регистрации используется модель "доверяемых" систем (**trusted hosts**). Если система **B доверяет** системе **A**, то все пользователи, зарегистрированные на системе **A**, автоматически получают доступ к системе **B** под теми же именами. Иногда аналогичную возможность можно предоставлять для каждого пользователя отдельно – пользователь сообщает, что при регистрации входа с системы **A** пароля запрашивать не надо.

Например, протоколы rlogin/rsh, обеспечивающие запуск отдельных команд или командного процессора на удаленной системе, используют файл `/etc/hosts.equiv` или `.rhosts` в домашнем каталоге пользователя на удаленной системе. Файл `/etc/hosts.equiv` содержит имена всех машин, которым наша система полностью доверяет. Файл `.rhosts` состоит из строк формата.

```
имя.удаленной.машины          имя_пользователя
```

При этом `имя.удаленной.машины` не может быть произвольным, оно обязано содержаться в файле `/etc/hosts`, в котором собраны имена и адреса всех удаленных машин, "известных" системе. То же требование обязательно и для машин, перечисленных в `/etc/hosts.equiv`.

Например, пользователь `fat` на машине `iceman.cnit.nsu.ru` набирает команду `rlogin -l fat Indy.cnit.nsu.ru`: войти в систему `Indy.cnit.nsu.ru` под именем `fat`. Если домашний каталог пользователя `fat` на целевой машине содержит файл `.rhosts`, в котором есть строка

```
iceman.cnit.nsu.ru          fat
```

то `fat` получит доступ к системе `Indy` без набора пароля. Того же эффекта можно добиться для всех одноименных пользователей, если `/etc/hosts.equiv` содержит строку iceman.cnit.nsu.ru

Если же fat наберет команду rlogin -l root Indy.cnit.nsu.ru, а в домашнем каталоге пользователя root файла .rhosts нет или он не содержит вышеприведенной строки, команда rlogin потребует ввода пароля, независимо от содержимого файла /etc/hosts.equiv. Нужно отметить, что администраторы обычно вообще не разрешают использовать rlogin для входа под именем root, потому что этот пользователь является администратором системы и обладает очень большими привилегиями.

Модель доверяемых систем обеспечивает большое удобство для пользователей и администраторов и в различных формах предоставляется многими сетевыми ОС. Например, в протоколе разделения файлов SMB (в обиходе этот протокол часто называют NetBIOS, хотя это и не совсем правильно), используемом в Windows for Workgroups, OS/2, Windows NT, Linux и др., используется своеобразная модель аутентикации, которую можно рассматривать как специфический случай доверяемых систем.

Аутентикация в SMB основана на понятии **домена** (**domain**). Каждый разделяемый ресурс (каталог, принтер и т.д.) принадлежит к определенному домену, хотя и может быть защищен собственным паролем. При доступе к каждому новому ресурсу необходимо подтвердить имя пользователя и пароль, после чего создается сессия, связанная с этим ресурсом. Для создания сессии используется надежное соединение, предоставляемое сетевым протоколом нижнего уровня – именованная труба NetBEUI или соединение TCP. Ввод пароля при каждом доступе неудобен для пользователя, поэтому большинство клиентов – Windows for Workgroups, Windows 95, Windows NT, OS/2 – просто запоминают пароль, введенный при регистрации в домене, и при подключении к ресурсу первым делом пробуют этот пароль. Благодаря этому удается создать у пользователя иллюзию однократной регистрации. Кроме того, если сессия по каким-то причинам оказалась разорвана, например из-за перезагрузки сервера, то можно реализовать прозрачное для пользователя восстановление этой сессии.

С точки зрения клиента нет смысла говорить о межмашинном доверии – клиенту в среде SMB никто не доверяет, и вполне справедливо: обычно это система класса ДОС, не заслуживающая доверия.

Однако серверы обычно передоверяют проверку пароля и идентификацию пользователя выделенной машине, называемой **контроллером домена**domain controller. Домен обязан иметь один основной (primary) контроллер и может иметь несколько резервных (backup). При поступлении запроса на соединение сервер получает у клиента имя пользователя и пароль, но вместо сверки с собственной базой данных он пересылает их контроллеру домена и принимает решение о принятии или отвергании сессии на основании вердикта, вынесенного контроллером.

Только контроллеры домена хранят у себя базу данных о пользователях и паролях. При этом основной контроллер хранит основную копию базы, а резервные серверы – ее дубликаты, используемые лишь в тех случаях, когда основной сервер выключен или потерян. Благодаря тому, что все данные собраны в одном месте, можно централизованно управлять доступом ко многим серверам, поэтому домены представляют неоценимые преимущества при организации больших многосерверных сетей.

С точки зрения безопасности доверяемые системы имеют два серьезных недостатка:

1. Прорыв безопасности на одной из систем означает, по существу, прорыв на всех системах, которые доверяют первой.

2. Возникает дополнительный тип атаки на систему безопасности: машина, которая выдает себя за доверяемую, не являясь таковой.

Первая проблема является практически неизбежной платой за разрешение автоматической регистрации. Наиболее ярко эта проблема была проиллюстрирована упомянутым выше "Червем Морриса" [23], который, проникнув в одну из систем, использовал содержимое файлов `.rhosts` и `/etc/hosts.equiv` для проникновения в доверяющие системы, полагаясь на то, что межсистемное доверие обычно делают взаимным.

Поэтому в средах с высокими требованиями к безопасности часто стремятся ограничить число доверяемых систем, подобно тому, как отсеки кораблей разделяют водонепроницаемыми переборками.

Вторая проблема может быть отчасти решена использованием средств, предоставляемых сетевыми протоколами, например, привязкой всех логических имен доверяемых систем к их физическим сетевым адресам. В протоколах TCP/IP это может быть сделано с использованием протокола arp (**Address Resolution Protocol** – протокол разрешения адресов).

Более изощренный и, по-видимому, намного более надежный метод основан на использовании алгоритма двухключевого шифрования RSA или родственных ему механизмов.

9.2.1 Криптографические методы идентификации

Алгоритм шифрования RSA использует два парных ключа. Сообщение, зашифрованное любым из ключей, может быть расшифровано другим. При этом если нам доступен только один из ключей, мы не можем восстановить другой. Длина ключей составляет несколько десятков или даже сотен байт, поэтому подбор ключей на современной вычислительной технике оказывается сильно затруднен. Обычно более длинный ключ называют **"частным"** (**private**), а более короткий – **общедоступным** (**public**), но для нас это различие не очень существенно. Подробное описание алгоритма RSA можно найти в документе [21].

Способы аутентикации, основанные на RSA, сводятся к следующему механизму:

- Система **A** генерирует последовательность байт, обычно случайную, кодирует ее своим ключом и посылает системе **B**.
- Система **B** раскодирует ее своим ключом. Это возможно, только если системы владеют парными ключами.
- Системы тем или иным способом обмениваются "правильными" значениями зашифрованной посылки.

Для примера рассмотрим принцип RSA-аутентикации в пакете `ssh` [22] – Secure Shell, разработанной Тату Илоненом (Tatu Ylonen). Пакет представляет собой функциональную замену программ `rlogin/rsh` и соответствующего этим программам демона `rshd`. В пакет входят программы `ssh` (клиент) и `sshd` (сервер), а также утилиты для генерации ключей RSA и управления ими. `ssh` использует RSA для прозрачной аутентикации пользователя при входе на удаленную систему. Кроме того, `ssh/sshd` могут осуществлять шифрование данных, передаваемых по линии во время сеанса связи, и выполнять ряд других полезных функций.

Сервер хранит список известных общедоступных ключей для каждого из пользователей в файле `$HOME/.ssh/authorized_keys`, где `$HOME` обозначает домашний каталог пользователя. Файл состоит из строк формата `host_name:key` – по строке для каждого из разрешенных клиентов. В свою очередь, каждый клиент хранит в файле `$HOME/.ssh/private_key` свой частный ключ.

Когда с удаленной системы-клиента приходит запрос на аутентикацию, `sshd` запрашивает общедоступный ключ. Если полученный ключ совпадает с хранящимся в файле значением для этой системы, сервер генерирует случайную последовательность из 256 бит, шифрует ее публичным ключом и посылает клиенту. Клиент расшифровывает посылку своим личным ключом, вычисляет 128-битовую контрольную сумму и возвращают ее серверу. Сервер сравнивает полученную последовательность с правильной контрольной суммой и принимает аутентикацию в случае совпадения. Теоретически контрольные суммы могут совпасть и в случае несовпадения ключей, но вероятность такого события крайне мала.

Методы, основанные на RSA и других алгоритмах шифрования, не могут решить проблемы распространения прорыва безопасности между доверяемыми системами: проникший в доверяемую систему взломщик получает доступ к "частным" ключам и может использовать их для немедленной регистрации в любой из доверяющих систем или даже скопировать ключи для проникновения в эти системы в более удобное время. Однако, как уже говорилось выше, это является практически неизбежной платой за разрешение автоматической регистрации в нескольких системах.

В то же время, криптографические методы практически устраняют опасность имитации доверяемой системы путем подмены сетевого адреса и значительно увеличивают надежность других методов аутентикации. Например, передача пароля по сети в зашифрованном виде, особенно при использовании двухключевого шифрования или динамических ключей, практически устраняет возможность утечки пароля путем его подслушивания и т.д.

Есть основания утверждать, что использование криптографических методов может решить многие проблемы безопасности компьютерных сетей и даже одиночных компьютеров.

9.3 Права доступа

Существует несколько моделей предоставления прав доступа к файлам и другим объектам. Наиболее простая модель используется в системах семейства Unix.

В этих системах каждый файл или каталог имеют идентификаторы хозяина и **группы**. Определено три набора прав доступа: для хозяина, группы (т.е., для пользователей, входящих в группу, к которой принадлежит файл) и всех остальных. Пользователь может принадлежать к нескольким группам одновременно, файл всегда принадлежит только одной группе.

Бывают три права: чтения, записи и исполнения. Для каталога право исполнения означает право на поиск файлов в этом каталоге. Каждое из прав обозначается битом в маске прав доступа, т.е. все три группы прав представляются девятью битами или тремя восьмеричными цифрами.

Пра́ва на удаление или переименование файла не существует; вообще, в Unix не определено операции удаления файла как таковой, а существует лишь операция удаления имени `unlink`[4]. Для удаления или изменения имени достаточно иметь право записи в каталог, в котором это имя содержится.

В традиционных системах семейства Unix все глобальные объекты – внешние устройства и именованные программные каналы – являются файлами (точнее, имеют имена в файловой системе), и управление доступом к ним охватывается файловым механизмом. В современных версиях Unix адресные пространства исполняющихся процессов также доступны как файлы в специальной файловой (или псевдофайловой, если угодно) системе `proc`. Файлы в этой ФС могут быть использованы, например, отладчиками для доступа к коду и данным отлаживаемой программы. Управление таким доступом также осуществляется стандартным файловым механизмом. Кроме доступа к адресному пространству, над процессом в Unix определена, по существу, только операция посылки сигнала. Обычный пользователь может посылать сигналы только своим процессам; только суперпользователь (`root`) может посылать их чужим процессам. Все остальные операции осуществимы только между процессами, связанными отношением родитель/потомок, и распределение прав доступа в этой ситуации вообще не нужно. Таким образом, файловые права доступа используются для управления доступом к практически любым объектам ОС.

В Unix System V появились объекты, не являющиеся файлами и идентифицируемые численными ключами доступа вместо имен. Все эти объекты являются средствами межпроцессного взаимодействия: это семафоры, очереди сообщений и сегменты разделяемой памяти. Каждый такой объект имеет маску прав доступа, аналогичную файловой, и доступ к ним контролируется точно так же, как и к файлам.

Основное преимущество этого подхода состоит в его простоте. Фактически это наиболее простая из систем привилегий, пригодная для практического применения. Иными словами, более простые системы непригодны вообще. Кроме того, практика эксплуатации систем, использующих эту модель, показывает, что она вполне адекватна подавляющему большинству реальных ситуаций.

Многие современные системы, не входящие в семейство Unix, а также и некоторые версии Unix, например, HP/UX или SCO UnixWare 2.x, используют более сложную и гибкую систему управления доступом, основанную на **списках управления доступом** (**Access Control Lists** – ACL). С каждым защищаемым объектом, кроме идентификатора его хозяина, связан список записей. Каждая запись состоит из идентификатора пользователя или группы и списка прав для этого пользователя или группы. Понятие группы в таких системах не играет такой большой роли, как в Unix, а служит лишь для

[4]Это связано с тем, что файл в Unix может иметь несколько имен, и собственно удаление происходит только при уничтожении последнего имени. Подробнее см. гл. 8.

сокращения ACL, позволяя задать права для многих пользователей одним элементом списка. Многие системы, использующие эту модель, например Novell Netware, даже не ассоциируют с файлом идентификатора группы.

Обычно список возможных прав включает в себя право на изменение ACL. Таким образом, не только хозяин объекта может изменять права доступа к нему. Это право может быть дано и другим пользователям системы или даже группам пользователей, если это окажется для чего-то необходимо.

Однако за дополнительную гибкость приходится платить снижением производительности. В системах семейства Unix проверка прав доступа осуществляется простой битовой операцией над маской прав. В системах же, использующих ACL, необходим просмотр списка, который может занять намного больше времени. Самое плохое состоит в том, что мы не можем гарантировать завершения этого поиска за какое-либо время, не устанавливая ограничений на длину списка.

В современных системах накладные расходы, связанные с использованием ACL, считаются достаточно малыми, поэтому эта модель распределения прав приобретает все бóльшую популярность.

Кроме прав доступа к объектам, система должна управлять выдачей некоторых привилегий. Так, для выполнения резервного копирования и восстановления файлов необходим пользователь, способный осуществлять доступ к файлам и операции на ними, не обращая внимания на права доступа.

Во всех системах необходимы пользователи, имеющие право изменять конфигурацию системы, заводить новых пользователей и группы и т.д. Если система обеспечивает запуск процессов реального времени, создание таких процессов тоже должно контролироваться, поскольку процесс РВ имеет более высокий приоритет, чем все процессы разделенного времени, и может, просто не отдавая процессор, заблокировать все остальные задачи.

В большинстве современных многопользовательских ОС, не входящих в семейство Unix, с каждым пользователем ассоциирован список привилегий, которыми этот пользователь обладает. В системах семейства Unix все гораздо проще: обычные пользователи не обладают никакими привилегиями. Для выполнения привилегированных функций существует пользователь с численным идентификатором 0 – **суперпользователь** (**superuser**), который обладает, подобно христианскому Господу Богу, всеми мыслимыми правами, привилегиями и атрибутами. По традиции суперпользователь имеет символьное имя `root` – “корень”.

Система списков привилегий предоставляет бóльшую гибкость, чем один сверхпривилегированный суперпользователь, потому что позволяет администратору системы передать часть своих функций, например выполнение резервного копирования, другим пользователям, не давая им при этом других привилегий. Однако, как и ACL, эта схема приводит к бóльшим накладным расходам при контроле прав доступа: вместо сравнения идентификатора пользователя с нулем мы должны сканировать список привилегий, что несколько дольше.

В некоторых ситуациях нужен более тонкий контроль за доступом, чем управление доступом на уровне файлов. Например, для изменения информации о пользователе необходим доступ на запись к соответствующей базе данных, но не ко всей, а только к определенной записи. Вполне естественно и даже необходимо дать пользователю возможность менять пароль, не обращаясь к администратору. С другой стороны, совершенно недопустима возможность менять пароли других пользователей. Одним из решений было бы хранение пароля для каждого из пользователей в отдельном файле, но это во многих отношениях неудобно. Другое решение может состоять в использовании модели клиент-сервер с процессом-сервером, исполняющимся с привилегиями администратора, который является единственным средством доступа к паролям. Например, в Windows NT весь доступ к пользовательской базе данных осуществляется через системные вызовы, то есть функции процесса-сервера исполняет само ядро системы. Этот подход позволяет решить проблему контроля доступа именно к пользовательской базе данных, но аналогичная проблема возникает и в других ситуациях.

В системах семейства Unix для этой цели был предложен оригинальный механизм, известный как **setuid** (**setting of user id** – установка [эффективного] идентификатора пользователя).

В Unix каждая задача имеет два пользовательских идентификатора: **реальный** и **эффективный**. Реальный идентификатор обычно совпадает с идентификатором пользователя, запустившего задание. Для проверки прав доступа к файлам и другим объектам, однако, используется эффективный идентификатор. При запуске обычных задач реальный и эффективный идентификаторы совпадают. Несовпадение может возникнуть при запуске программы с установленным признаком `setuid`. При этом эффективный идентификатор для задачи устанавливается равным идентификатору хозяина файла, содержавшего программу.

Признак `setuid` *является атрибутом файла, содержащего загрузочный модуль программы. Только хозяин может установить этот признак, таким образом передавая другим пользователям право исполнять эту программу от своего имени. При модификации файла или при передаче его другому пользователю признак* `setuid` *автоматически сбрасывается.*

Так, например, программа `/bin/passwd` принадлежит пользователю `root`, и у нее установлен признак `setuid`. Любой пользователь, запустивший эту программу, получает задачу, имеющую право модификации пользовательской базы данных – файлов `/etc/passwd` и `/etc/shadow`. Однако прежде чем произвести модификацию, программа `passwd` проверяет допустимость модификации. Например, при смене пароля она требует ввести старый пароль. Сменить пароль пользователю, который его забыл, может только `root`.

Другим примером `setuid`-программы может служить программа `/bin/ps` (process status – состояние процессов). Системы семейства Unix не предоставляют системных вызовов для получения статистики об исполняющихся процессах. Программа `/bin/ps` анализирует виртуальную память ядра системы, доступную как файл устройства `/dev/kmem`, находит в ней список процессов и выводит содержащиеся в списке данные. Естественно, только привилегированная программа может осуществлять доступ к `/dev/kmem`.

Механизм **setuid** был изобретен одним из отцов-основателей Unix Деннисом Ритчи и запатенован фирмой AT&T в 1975 г. Через несколько месяцев после получения патенту был дан статус public domain. Официально фирма AT&T объяснила это тем, что плату за использование данного патента нецелесообразно делать высокой, а сбор небольшой платы с большого числа пользователей неудобен и тоже нецелесообразен.

Механизм **setuid** позволяет выдавать привилегии, доступные только суперпользователю, как отдельным пользователям (при этом `setuid`-программа должна явным образом проверять реальный идентификатор пользователя и сравнивать его с собственной базой данных), так и группам (при этом достаточно передать `setuid`-программу соответствующей группе и дать ей право исполнения на эту программу), таким образом отчасти компенсируя недостаточную гибкость стандартной системы прав доступа и привилегий в системе Unix.

9.4 Защита оперативной памяти

Легко понять, что разделение доступа к данным невозможно без контроля над доступом к другим объектам системы, в первую очередь к оперативной памяти, физическим внешним устройствам и драйверам этих устройств. Без такого контроля все средства управления доступом к данным во внешней памяти оказываются бессмысленными, как запертая дверь в сарае без одной стены. Поэтому нельзя всерьез говорить о безопасности в системах, не разделяющих доступ задач к физической памяти, таких как DOS, MS Windows 3.x, Windows 95, MacOS и др.

При этом пользовательские задачи должны иметь возможность обмениваться данными с модулями ОС и друг с другом[5].

9.4.1 Кольца защиты

В современных системах чаще всего используется аппаратно реализованное разделение системного и пользовательского уровней привилегий. Процессор может работать в одном из двух режимов: **пользовательском** и **системном**. Иногда системный режим называют режимом **супервизора** (**supervisor** – надзиратель). Обычно такую архитектуру называют **кольцами защиты**: структуру привилегий изображают в виде двух концентрических кругов, где пользовательский круг является внешним и менее привилегированным, а системный – внутренним. В ряде современных систем количество колец может быть увеличено, например процессоры семейства x86 имеют четыре уровня привилегий.

В системном режиме разрешено управление уровнем прерываний процессора и доступ к регистрам диспетчера памяти. Если процессор использует отдельное адресное пространство для ввода/вывода, то команды доступа к этому адресному пространству также обычно разрешены только в системном режиме. Если же используется отображение регистров устройств на адреса памяти, система может управлять доступом к внешним устройствам более простым и гибким способом, отображая страницы ввода/вывода только привилегированным задачам.

[5] На самом деле задачам достаточно иметь возможность обмениваться данными с ядром. Межзадачное взаимодействие в этом случае может быть организовано при посредстве ядра. Примерами такого опосредованного взаимодействия являются трубы в системах семейства Unix и аналогичные трубам средства в других ОС.

Однако опосредованное взаимодействие требует дополнительных накладных расходов, поэтому часто непосредственное разделение частей адресного пространства между задачами оказывается более удобным.

Обычно программа, исполняющаяся в системном режиме, имеет доступ к адресным пространствам пользовательских программ.

Напротив, в пользовательском режиме процессор может исполнять только обычные команды обработки данных и не имеет доступа к системному адресному пространству. Переключение из пользовательского режима в системный осуществляется специальной командой. Обычно такая команда сразу же передает управление одному из модулей ядра системы.

Например, в PDP-11 системные вызовы реализуются через команду `EMT` – Emulate Trap (программное прерывание). Эта команда имеет 6-битовое поле операнда, которое, однако, не интерпретируется процессором. Пользовательская программа проталкивает параметры системного вызова в стек и исполняет `EMT` с операндом, равным коду исполняемого системного вызова. При этом вызывается обработчик прерывания по вектору 8.

Прерывание 8 обрабатывается диспетчером системных вызовов. Обработчики прерываний в PDP-11 всегда исполняются в системном режиме, поэтому при вызове диспетчера автоматически происходит переключение в режим ядра. Диспетчер извлекает из пользовательского стека адрес команды `EMT` и параметры. Затем он извлекает код команды `EMT`, анализирует ее операнд и в зависимости от результатов анализа вызывает соответствующую процедуру ядра. Для копирования параметров используются специальные команды доступа к пользовательскому адресному пространству: `MFPI/MFPD` (Move From Previous Instruction/Data – копировать из предыдущего [адресного пространства] команд/данных).

В RSX-11, использующей разделение памяти, fork-процесс драйвера все время имеет доступ к пользовательскому адресному пространству через команды `MFPI/MFPD`, поэтому не возникает необходимости копировать данные в системные буфера, как в Linux.

Несколько иная схема обмена данными используется в Windows NT и некоторых микроядерных системах: вместо прямого отображения пользовательских адресов в системные или специальных команд копирования между адресными пространствами используются небольшие (4 кбайта в случае NT) разделяемые буфера. Такое решение позволяет отчасти защитить пользовательские программы от ошибок в модулях ОС и упрощает синхронизацию доступа к разделяемым данным, но приводит к значительным накладным расходам.

В Windows NT для обмена данными между пользовательскими программами и ядром используются буфера, однако многие модули ядра обмениваются данными путем прямого разделения частей адресного пространства для повышения производительности. Несмотря на это, NT близка к статусу чемпиона среди современных ОС по потребностям в памяти и накладным расходам, хотя и трудно определить, с чем это связано: с буферизацией или с чем-либо еще, например, с плохо сбалансированным дисковым кэшем.

Легко понять, что такие модули, как планировщик, менеджер памяти и драйверы внешних устройств, могут работать только в системном режиме. Обычно в системном режиме исполняется все ядро ОС, в том числе и те модули, для которых это необязательно – файловые системы, сервисные модули разного рода и т.д.

В результате мы получаем двухуровневую систему прав, напоминающую систему привилегий в ОС семейства Unix: пользовательские программы имеют доступ только к своим внутренним объектам и тем внешним объектам, доступ к которым им разрешен ядром, ядро же обладает всеми мыслимыми правами и привилегиями.

При этом пользовательские программы вынуждены полагаться на порядочность модулей ядра. Ядро же обычно даже не может эффективно защитить себя от ошибок в собственных модулях.

Каждая система обычно исполняет по крайней мере несколько дополнительных модулей, которые хотя и являются частью ядра, но разрабатывались и отлаживались отдельно от самой ОС. В первую очередь такими модулями являются драйверы внешних устройств, которые обычно разрабатываются фирмами-изготовителями аппаратуры, а не поставщиками ОС. Кроме того, система может использовать внешние драйверы файловых систем, сетевых протоколов и т.д. Все эти модули исполняются с привилегиями системного режима и, таким образом, могут в случае ошибки нарушить работу ядра. Есть основания утверждать, что ошибки в таких модулях являются основной причиной сбоев современных ОС общего назначения.

Желание защитить ядро и пользовательские программы от ошибок в системных модулях вынуждает разработчиков современных процессоров усложнять структуру аппаратных привилегий. Некоторые процессоры, например семейство x86, предлагают несколько уровней привилегий в отличие от двух в модели пользователь/супервизор, но и они сохраняют "концентрическую" структуру привилегий, когда более привилегированный уровень автоматически получает доступ ко всем объектам предыдущего уровня. Таким образом, код, исполняющийся в нулевом кольце защиты, мало чем отличается по возможностям от кода супервизора в системах с двумя уровнями привилегий.

Альтернативный подход состоит в том, чтобы, не накладывая дополнительных требований на аппаратуру, как можно сильнее упростить само ядро и дополнительные модули, исполняющиеся в режиме ядра. Этот подход, реализованный, например, в Unix, не решает проблемы в принципе, но по крайней мере позволяет уменьшить вероятность опасных ошибок в ядре и системных модулях.

Описанная архитектура с различными вариациями используется в большинстве систем семейства Unix, OS/2, Windows NT, VAX/VMS-OpenVMS и ОС для больших компьютеров фирмы IBM: MVS, VM/ESA, OS/390. В совокупности эти системы образуют подавляющее большинство современных ОС общего назначения.

Отчасти похожую архитектуру памяти использует и Windows 95, но эта ОС не защищает значительную часть системных адресов от пользовательских задач. Никак не защищен оказывается первый мегабайт физической памяти, который зачем-то отображается в адресные пространства всех 32-разрядных программ, значительная часть ядра и даже глобальная таблица дескрипторов.

Поражают воображение заявления фирмы Microsoft по этому поводу. В одном из пресс-релизов, опубликованных еще до выпуска системы, официальные представители фирмы на полном серьезе утверждали, что на самом деле защищать надо лишь первые 64 килобайта адресного пространства, поскольку, по их измерениям (?!), 90 % ошибочных доступов приходятся именно на эти адреса. Комментарии излишни.

Широкая распространенность обсуждаемой архитектуры вовсе не означает, что она лишена недостатков. Безусловно, она лучше систем с открытой памятью, которые во многих случаях оказываются неприменимы или неудобны из-за низкой надежности и невозможности создания эффективных систем безопасности, но этого недостаточно, чтобы утверждать, что такая архитектура оказалась лучше всех альтернатив.

Важным недостатком раздельных адресных пространств является относительная сложность разделения памяти между задачами, что особенно заметно при разделении кода. Прослеживание множественных ссылок на разделяемую библиотеку подпрограмм приводит к снижению производительности алгоритмов подкачки и поиска жертвы, а также к более сложному управлению файлом подкачки. Из-за этого у систем с динамической сборкой (OS/2 и Windows NT) виртуальная память заметно менее эффективна, чем у систем семейства Unix, использующих статические загрузочные модули. Статические же загрузочные модули обеспечивают гораздо меньшую гибкость, чем динамическая сборка. Такой выбор между "бедным, но здоровым" и "богатым, но больным" трудно назвать привлекательным.

9.4.2 Альтернативные подходы к защите памяти

Большинство альтернативных решений, в том числе и i432, так или иначе восходят к системам Burroughs, первые из которых были разработаны в конце 60-х гг. В этих системах использовалась сегментная виртуальная память, но все задачи разделяли общую таблицу дескрипторов сегментов. Таким образом, можно сказать, что все задачи имели общее адресное пространство. Разделение же доступа к сегментам осуществлялось путем избирательной выдачи селекторов.

В Burroughs пользовательская программа не могла самостоятельно формировать указатели, так как эти машины имели теговую архитектуру, когда каждая ячейка данных снабжена описателем-**тегом** (**tag**), указывающим на тип данных, хранящихся в данной ячейке.

Благодаря тегам команды модификации скалярных данных не могут оперировать с указателями. Команды же создания и модификации указателей являются привилегированными и доступны только специализированному модулю ядра.

Простая передача указателя между задачами в Burroughs приводила к тому, что задача-получатель получала доступ к указуемому объекту, что в системах с раздельными таблицами дескрипторов соответствовало бы отображению объекта в два адресных пространства, но без связанных с отображением сложностей и накладных расходов. Это делает архитектуру типа Burroughs очень привлекательной альтернативой системам с разделением адресных пространств.

Однако на машинах, не использующих тегов, такая архитектура приводит фактически к полному отказу от защиты памяти. Теги же, в свою очередь, приводят к значительному усложнению аппаратуры процессора и системы команд, что привело к почти полному вытеснению теговых архитектур с рынка в 70-е гг. Несмотря на это, идея использовать контроль за передачей указателей вместо разделения адресных пространств продолжала интриговать разработчиков и исследователей на протяжении 70-х гг. Неясно было только, как эффективно реализовать такой контроль.

Одна из остроумных идей состояла в том, чтобы выполнять контроль за модификацией указателей статически, еще на этапе ассемблирования или компиляции. Получалось, что система доверяет языковым процессорам в том, что они всегда генерируют только безопасный код. Этот подход был реализован во многих экспериментальных системах, но оказался непригоден для реальных приложений, где часто возникает необходимость использовать языковые процессоры, разработанные третьими организациями.

Успехи микропрограммирования привели во второй половине 70-х – начале 80-х гг. к буйному расцвету CISC-архитектур. В микрокод переносили интерпретаторы Lisp, SmallTalk и целые модули операционных систем. На фоне этого расцвета возникла новая волна интереса к архитектурам с контролем за передачей указателя. Например, идею i432, кратко обсуждаемого в п. 9.4.3 тоже можно проследить вплоть до желания улучшить и усовершенствовать архитектуру Burroughs.

Среди возникшей волны CISC-архитектур такого типа нельзя не упомянуть System/38 фирмы IBM, кратко описанную в [25] и ряде других работ.

Значительная часть ОС System/38 реализована микропрограммно. Точнее, в этой системе трудно сказать, где кончается ОС и начинается система команд. Система использует одноуровневое адресное пространство для оперативной и внешней памяти. Все адресное пространство разбито на объекты различных типов. Каждый объект имеет набор атрибутов, включая имя объекта и идентификатор его владельца. Структура большинства объектов скрыта от прикладных программ. Доступ к объектам осуществляется специальными командами. Операции делятся на типовые, приложимые к любым объектам, и уникальные, приложимые только к объектам определенного типа.

Доступ к объектам может осуществляться как по адресам, так и по символьным именам. Роль каталогов при преобразовании имен в адреса выполняют специальные объекты, называемые **контекстами** (**context**). Адрес, называемый **системным указателем**, состоит из расположения объекта (т.е. собственно указателя) и прав доступа к объекту. Изменение прав доступа осуществляется специальными командами. Пользователь не может изменить эти права прямыми битовыми манипуляциями. Таким образом, System/38 вместо разделения адресных пространств использует контроль передачи указателя, как и Burroughs. Указатель с ограниченными правами приблизительно соответствует мандату в i432.

Пользовательская программа может собирать составные объекты из нескольких элементарных объектов. Например, реляционная база данных может состоять из нескольких таблиц данных, индексов для каждой из таблиц, курсора и сегмента памяти для скалярных переменных. Пользователь может рассматривать такую сложную составную сущность как единый объект.

Часть типов объектов может быть сопоставлена с сегментами кода и данных в традиционных архитектурах – это объекты, содержащие скалярные данные и код. Другие объекты, такие как описатели логических устройств и сетевых портов или пользовательские профили, лишь с большой натяжкой сопоставляются с обычными сегментами. Многие типы объектов ориентированы на работу с реляционными базами данных. Это такие типы, как:

Data Space – область данных разбитая на записи фиксированного формата, соответствующая таблице в обычных реляционных СУБД;

Data Space Index – индекс поиска для таблицы;

Cursor – средство для адресации в таблице.

Концепция System/38 получила несколько неожиданное развитие в машинах серии AS/400 (Application Server – сервер приложений). System/38 использовала микропрограммное обеспечение большого объема. Бо́льшая часть микрокода хранилась в основной памяти, что в значительной мере размывало границу между микрокодом и обычными программами. При разработке AS/400 разработчики фирмы IBM сделали смелый шаг, переместив эту границу из процессора в загрузчик программ.

В традиционных CISC-системах центральный процессор извлекает из основной памяти команды машинного языка и преобразует их в последовательности микрокоманд. В традиционных RISC-системах микрокоманд как таковых нет, и аппаратура процессора непосредственно интерпретирует машинный язык. В AS/400, как и в CISC-системах, существует различие между "внешним" машинным языком, команды которого хранятся в бинарных загрузочных модулях, и командами аппаратуры ЦПУ. Однако преобразование внешнего машинного кода в язык команд процессора происходит в момент загрузки программы. Загрузчик считывает из загрузочного модуля последовательность высокоуровневых объектно-ориентированных команд и преобразует их в последовательность команд реального ЦПУ. В современных моделях AS/400 используется RISC-процессор с системой команд Power, в основном аналогичный процессорам, используемым в RS/6000, PowerPC и PowerMAC. Пользователь не может сформировать и запустить произвольную последовательность команд этого ЦПУ и вообще не имеет доступа к реальному процессору.

Можно сказать, что AS/400 представляет собой удачный компромисс между упоминавшейся выше идеей о статическом контроле за передачей указателей на этапе компиляции/ассемблирования и требованиями практики эксплуатации вычислительных систем. Языковые процессоры в AS/400 генерируют высокоуровневые команды, реальный же процессор исполняет код, созданный системным загрузчиком на основе этих команд. Большая часть проверок "безопасности" исполняемых операций выполняется загрузчиком статически и не оказывает влияния на производительность системы.

Такой подход имеет еще одно большое преимущество: теперь оказывается возможным произвольно менять систему команд центрального процессора, не теряя совместимости со старыми загрузочными модулями.

9.4.3 Взаимно недоверяющие подсистемы

С точки зрения безопасности основной проблемой систем с кольцами защиты является неспособность таких систем защитить себя от ошибок в модулях, исполняющихся в высшем кольце защиты. В свете этого, очень привлекательной концепцией представляется идея **взаимно недоверяющих подсистем**.

Согласно этой концепции, пользовательская задача не должна предоставлять системе доступа ко всем своим данным. Вместо этого задача должна выделять **мандат** на доступ к буферу или нескольким буферам, предназначенным для обмена данными. Все акты обмена данными, как между пользовательской задачей и системой, так и между двумя пользовательскими задачами или двумя модулями системы, также осуществляются при помощи передачи мандатов.

Например, при исполнении системного вызова `int read(int file, void * buf, size_t size)` программа должна передать системе мандат на право записи в буфер `buf` размером `size` байт. При этом буфер будет отображен в адресное пространство подсистемы ввода/вывода, но эта подсистема не получит права записи в остальное адресное пространство нашей задачи. Впрочем, этот подход имеет две очевидные проблемы:

- При использовании страничных и двухслойных сегментно-страничных диспетчерах памяти мы можем отображать в чужие адресные пространства только объекты, выровненные на границу страницы и имеющие размер, кратный размеру страницы.
- Мы все равно не можем избавиться от супервизора. В данном случае это должен быть модуль, ответственный за формирование мандата и размещение отображенного модуля в адресном пространстве задачи, получающей мандат.

Первая проблема является чисто технической – она приводит лишь к тому, что мы не можем пользоваться страничными диспетчерами и оказываемся ограничены сегментными диспетчерами памяти, в которых размер сегмента задается с точностью до байта или хотя бы до слова. Чисто сегментная память страдает от фрагментации и имеет ряд специфических недостатков при организации виртуальной памяти, но иногда такую цену стоит заплатить за повышение надежности.

Однако вторая проблема является принципиальной и очень глубокой. Фактически с аналогичной проблемой нередко сталкиваются многие бюрократические организации, в которых клерк, выдающий бумажки, на практике оказывается облечен ни с чем не сообразной властью.

Функции модуля, управляющего выдачей мандатов, оказываются слишком сложны для аппаратной или даже микропрограммной реализации. Этот модуль должен выполнять следующие функции:

- Убедиться в том, что передаваемый объект целиком входит в один сегмент исходного адресного пространства.
- Если объект состоит из нескольких сегментов, разумным образом обработать такую ситуацию. Для программной реализации может оказаться желательным умение объединить все элементы в один сегмент. Для аппаратной или микропрограммной реализации достаточно хотя бы уметь сгенерировать соответствующее исключение.
- Сформировать содержимое дескриптора сегмента для объекта и записать в него соответствующие права. Эта операция требует формирования 4 - 5 битовых полей, и запись ее алгоритма на

псевдокоде займет около десятка операторов. По сложности алгоритма одна только эта операция сравнима с наиболее сложными командами современных коммерческих CISC-процессоров, таких как VAX или транспьютер.

- Отметить общесистемной базе данных, что на соответствующую область физической памяти существует две ссылки. Это нужно для того, чтобы процессы дефрагментации и управления виртуальной памятью правильно обрабатывали перемещения сегмента по памяти и его перенос на диск, отмечая изменения физического адреса или признака присутствия во *всех* дескрипторах, ссылающихся на данный сегмент. Ниже эта проблема будет обсуждаться подробнее.

- Найти свободную запись в таблице дескрипторов сегментов задачи-получателя. Эта операция аналогична строковым командам современных CISC-процессоров, которые сами по себе считаются сложными командами.

- Разумным образом обработать ситуацию отсутствия такой записи.

- Записать сформированный дескриптор в таблицу.

Но еще больше проблем создает операция уничтожения мандата. Легко показать необходимость требования, чтобы уничтожение мандата осуществлялось той же задачей, которая его создала.

Например, представим себе, что пользовательская программа передала какому-либо модулю мандат на запись в динамически выделенный буфер. Представим себе также, что этот модуль вместо уничтожения мандата после передачи данных сохранил его, поскольку наша система предполагает, что ни одному из модулей нельзя полностью доверять! Если мы вынуждены принимать в расчет такую возможность, мы не можем ни повторно использовать такой буфер для других целей, ни даже возвратить память из-под буфера системе, потому что недоверяемый модуль по прежнему может сохранять право записи в него.

В свете этого возможность для задачи, выдавшей мандат, в одностороннем порядке прекращать его действие представляется жизненно необходимой. Отказ от этой возможности или ее ограничение приводит к тому, что задача, выдающая мандат, вынуждена доверять тем задачам, которым мандат был передан, т.е. можно не городить огород и вернуться к обычной двухуровневой или многоуровневой системе колец защиты.

Для прекращения действия мандата мы должны отыскать в нашей общесистемной базе данных все ссылки на интересующий нас объект и объявить их недействительными. При этом мы должны принять во внимание возможности многократной передачи мандата и повторной выдачи мандатов на отдельные части объекта. Аналогичную задачу нужно решать при перемещении объектов по памяти во время дефрагментации, сбросе на диск и поиске жертвы для такого сброса.

Структура этой базы данных представляется нам совершенно непонятной, потому что она должна отвечать следующим требованиям:

- обеспечивать быстрый поиск всех селекторов сегментов, ссылающихся на заданный байт или слово памяти;
- обеспечивать быстрое и простое добавление новых селекторов, ссылающихся на произвольные части существующих сегментов;
- занимать не больше места, чем память, отведенная под сегменты.

Последнее требование непосредственно не следует из концепции взаимно недоверяющих подсистем, но диктуется простым здравым смыслом. Если мы и смиримся с двух- или более кратным увеличением потребностей в памяти, оправдывая его повышением надежности, нельзя забывать, что даже простой просмотр многомегабайтных таблиц не может быть *быстрым*.

В системах, использующих страничные диспетчеры памяти, решить эти проблемы намного проще, потому что минимальным квантом разделения памяти является страница размером несколько сотен байт или несколько килобайт. Благодаря этому мы смело можем связать с каждым разделяемым квантом несколько десятков байт данных, обеспечивающих выполнение первых двух требований. Если же минимальным квантом является байт или слово памяти, наши структуры данных окажутся во много раз больше распределяемой памяти.

Авторы оставляют читателю возможность попробовать самостоятельно разработать соответствующую структуру данных. В качестве более сложного упражнения можно рекомендовать записать алгоритм работы с такой базой данных на псевдокоде, учитывая, что мы хотели бы иметь возможность реализовать этот алгоритм аппаратно или по крайней мере микропрограммно.

В качестве дополнительного стимула для читателя, решившего взяться за эту задачу, скажем еще, что разработчики фирмы Intel не смогли найти удовлетворительного решения.

Авторам известен только один процессор, пригодный для полноценной реализации взаимно недоверяющих подсистем: iAPX432 фирмы Intel. Вместо создания системной базы данных о множественных ссылках на объекты, специалисты фирмы Intel усложнили диспетчер памяти и соответственно алгоритм разрешения виртуальных адресов.

Виртуальный адрес в i432 состоит из селектора объекта и смещения в этом объекте. Селектор объекта ссылается на таблицу доступа текущего **домена**[6]. Элемент таблицы состоит из прав доступа к объекту и указателя на таблицу объектов процесса.

Элемент таблицы объектов может содержать непосредственную ссылку на область данных объекта – такой объект аналогичен сегменту в обычных диспетчерах памяти, однако обращение к нему происходит через два уровня косвенности. Иными словами, вместо

```
segment_table[seg].phys_address + offset
```

[6]Домен представляет собой программный модуль вместе со всеми его статическими и динамическими данными. По идее разработчиков, домен соответствует замкнутому модулю языков высокого уровня, например **пакету** (**package**) языка Ada.

мы имеем

```
object_table[access_table[selector]].phys_address + offset.
```

Дополнительный уровень косвенности позволяет нам упростить отслеживание множественных ссылок на объект: вместо использования структуры данных, которую мы так и не сумели изобрести, мы можем следить только за дескриптором исходного объекта, поскольку все ссылки на объект через дескрипторы доступа вынуждены в итоге проходить через дескриптор объекта.

Более сложным типом объектов являются **уточнения** – объекты, ссылающиеся на отдельные элементы других объектов. Вместо физического адреса и длины дескриптор такого объекта содержит *селектор целевого сегмента*, смещение уточнения в целевом объекте и его длину. При ссылке на уточнение диспетчер памяти сначала проверяет допустимость смещения, а затем повторяет полную процедуру разрешения и проверки прав доступа для целевого объекта. Целевой объект также может оказаться уточнением, и диспетчер памяти будет повторять процедуру до тех пор, пока не дойдет до объекта, ссылающегося на физическую память. Более подробно этот механизм описан в работе [24].

Уточнения позволяют решить проблему выдачи мандатов – мандат реализуется как уточнение исходного объекта (например, сегмента данных модуля, в котором выделен буфер), а программе-получателю передается даже не само уточнение, а лишь дескриптор доступа к нему. Для прекращения действия мандата достаточно удалить дескриптор уточнения. После этого все переданные другим модулям дескрипторы доступа будут указывать в пустоту, т.е. станут недействительными. Задачи же перемещения и сброса объектов на диск решаются путем изменения физического адреса или признака присутствия целевого объекта; для этого не требуется прослеживать цепочку уточнений.

Однако накладные расходы, связанные с реализацией этого сложного механизма, оказались очень большими. Процессор i432 остался экспериментальным и не получил практического применения. Трудно сказать, какой из факторов оказался важнее: катастрофически низкая производительность системы[7] или просто неспособность фирмы Intel довести чрезмерно сложный кристалл до массового промышленного производства. Так или иначе едва ли не единственной практической пользой, извлеченной из этого амбициозного проекта, оказалось тестирование системы автоматизированного проектирования, использованной впоследствии при разработке процессора 80386 и сопутствующих микросхем.

После завершения проекта i432 других попыток полностью реализовать взаимно недоверяющие подсистемы не делалось – изготовители коммерческих систем, по-видимому, сочли эту идею нецелесообразной, а исследовательским организациям проекты такого масштаба просто не по плечу.

Простой анализ показывает еще одно уязвимое место идеи взаимно недоверяющих подсистем: любая современная вычислительная система находится в зависимости от собственной дисковой подсистемы. Какие бы мы ни нагромождали средства защиты и взаимного недоверия, практически все данные, с которыми работает система, проходят через дисковый драйвер и драйвер файловой системы. Если с этими подсистемами что-либо случится, то вся система в целом будет совершенно бесполезна; если же в них будут встроены троянские подпрограммы, то безопасность данных окажется под угро-

[7] Авторам не удалось найти официальных заявлений фирмы Intel по этому поводу, но фольклор утверждает, что экспериментальные образцы i432 с тактовой частотой 20 МГц исполняли около 20.000 операций в секунду.

зой. То же самое можно сказать о сетевой подсистеме в сетевых многомашинных конфигурациях и подсистеме пользовательского интерфейса в интерактивных системах.

Интересный альтернативный подход предлагает микроядерная архитектура: в микроядре вместо отображения передаваемых данных в адресное пространство получателя используется копирование данных через разделяемый буфер. Обычно вместо простого копирования используются различные примитивы, обеспечивающие передачу данных вместе с синхронизацией, что позволяет решить сразу две проблемы разделения доступа к данным: с точки зрения синхронизации и с точки зрения безопасности. Однако использование буферов приводит к относительно большим накладным расходам.

Глава 10

Пользовательский интерфейс

Как сказано в нашем определении ОС, операционная система должна предоставлять пользовательский интерфейс. Как минимум она должна предоставлять **командную оболочку** (**shell**), которая дает пользователю возможность тем или иным способом запустить его прикладную программу. Однако в некоторых случаях, например, во встраиваемых контроллерах и других специализированных приложениях, такая оболочка может отсутствовать. При этом либо система вообще функционирует без вмешательства человека, либо пользователь работает только с одной прикладной программой.

Кроме того, ОС часто предоставляют средства – разделяемые библиотеки, серверы и т.д. для реализации графического пользовательского интерфейса прикладными программами. Часто оказывается сложно провести границу между ядром ОС и этими средствами, особенно если стандартная оболочка ОС реализована с их использованием. В некоторых системах, например в MS Windows 3.x и MacOS, практически все ядро состоит из средств реализации графического интерфейса.

В настоящее время оформилось два принципиально различных подхода к организации пользовательского интерфейса. Первый, исторически более ранний подход состоит в предоставлении пользователю командного языка, в котором запуск программ оформлен в виде отдельных команд. Этот подход известен как **интерфейс командной строки** (**Command Line Interface** – CLI).

Альтернативный подход состоит в символическом изображении доступных действий в виде картинок – **икон**icons на экране и предоставлении пользователю возможности выбирать действия при помощи мыши или другого координатного устройства ввода. Этот подход известен как **графический пользовательский интерфейс** (**Graphical User Interface** – GUI). Мы в дальнейшем будем использовать английские аббревиатуры, потому что писать полное название долго, русскоязычные аббревиатуры-кальки очень уж неблагозвучны, а выдумать короткий, корректный и благозвучный русскоязычный термин мы – скажем честно – слабы.

Разработчики современных ОС обычно предоставляют средства для реализации обоих подходов и, зачастую, оболочки, использующие оба типа интерфейсов. Однако среди пользователей предпочтение разных подходов вызывает горячие споры и, подчас, настоящие религиозные войны. Не желая вступать в такую войну ни на одной из сторон, мы попытаемся изложить аргументы в пользу каждого из подходов.

10.1 Аргументы в пользу CLI

Язык представляет одно из величайших достижений человечества. Многие ученые и философы обоснованно считают, что именно язык определяет границу, по которой проходит граница между человеком и животным. По современным представлениям, человеческий мозг содержит врожденные структуры, облегчающие понимание текстов на естественных и искусственных языках и генерацию таких текстов.

В свою очередь, компьютер, во всяком случае универсальный неймановский компьютер, представляет собой языковую машину: устройство, которое исполняет последовательность предложений некоторого формализованного языка.

Язык компьютера может быть только функциональным подмножеством естественного языка, потому что вычислительных возможностей современных компьютеров явно не достаточно для отображения всего богатства и сложности реального мира. В свете этого кажется разумным использование полностью синтетических языков, а не подмножеств естественных: ведь человеку проще выучить совершенно новый язык, чем приучаться пользоваться ограниченным подмножеством родного языка.

Изучение синтетического языка ненамного сложнее изучения естественного иностранного, потому что человеческий мозг от рождения обладает способностью к изучению и пониманию языков.

Удачно спроектированные искусственные языки удобны и для компьютера: интерпретаторы полнофункциональных командных языков обходятся десятками, в худшем случае сотнями килобайт оперативной памяти и обеспечивают очень высокую скорость и эффективность использования ресурсов системы. Поэтому синтетический язык как средство человеко-машинного общения является наилучшим и наиболее естественным выбором для обеих сторон.

Командные языки позволяют естественным образом перейти к написанию командных файлов или **скриптов** (**scripts**), позволяющих автоматизировать часто исполняемые задачи. Трудно провести границу между написанием скриптов и программированием; можно даже сказать, что написание скриптов и даже просто интерактивное использование командного языка и есть частный случай программирования. Но ведь смысл компьютера и состоит в том, что это универсальная программируемая машина!

Современные интерактивные командные процессоры решают практически все проблемы командных языков предыдущих поколений.

- Исправление опечаток в командах и набор последовательностей одинаковых или похожих команд осуществляется с использованием средств вспоминания ранее набранных строк – **“истории”**. Современные командные процессоры обеспечивают гибкие средства поиска команд в историческом списке, их редактирования, повторного использования отдельных частей этих команд и т.д.

- Набор длинных имен файлов, каталогов и других объектов облегчается автоматическим расширением имен. Это средство, реализованное во многих командных процессорах ОС семейства Unix и программах типа 4DOS/4OS2/4NT, позволяет, набрав первые несколько символов имени, расширить это имя до полного или получить список всех имен, начинающихся с данной последовательности букв.

- Неудобные, плохо запоминаемые или почему-либо не устраивающие пользователя команды могут быть переименованы с использованием **синонимов** (**aliases**). Этот же механизм может быть использован для сокращения часто исполняемых сложных команд.

Важными преимуществом хороших командных языков по сравнению с GUI является их алгоритмическая полнота: в GUI пользователь ограничен теми возможностями, для которых разработчик программы нарисовал иконки или сочинил пункты в меню. Командные же языки могут использоваться для решения любых алгоритмизуемых задач, в том числе и таких, о которых разработчики языка никогда и не задумывались.

Любопытно, что последнее достижение в области пользовательского интерфейса – распознавание речи и голосовое управление – означает, по существу, возврат к командному языку, с той лишь разницей, что команды произносятся, а не вводятся с клавиатуры. Возможно, следует ожидать появления нового поколения синтетических командных языков, соответствующих требованиям речевого ввода команд.

10.2 Аргументы в пользу GUI

Командные языки требуют затрат времени и усилий для изучения. Нужно отметить, что при изучении требуются не только интеллектуальные, но и эмоциональные усилия. Наиболее неприятны в этом смысле первые минуты общения с незнакомой системой, когда пользователь ощущает растерянность и не может сформулировать желаемое действие не только на незнакомом синтетическом языке но даже и на родном естественном.

Любопытно, что большинство женщин, впервые севших за компьютер задают один и тот же вопрос: "А на что тут нажимать?". Встречный вопрос: "А чего, собственно, вы хотите получить?" обычно повергает их в еще большую растерянность.

Первое негативное впечатление может создать устойчивый страх перед компьютером, затрудняя его эффективное использование и изучение. Напротив, графический интерфейс предоставляет новому пользователю возможность быстро окинуть взглядом доступные возможности и выбрать желаемую. Во многих случаях наглядность вариантов оказывается важнее богатства возможностей.

В некоторых случаях излишнее богатство вариантов может просто запутать пользователя. Не нужно забывать, что человек способен одновременно оперировать лишь довольно ограниченным количеством объектов и параметров; по современным представлениям для человека и большинства теплокровных животных это количество ограничено 6 - 7 объектами.

Даже после освоения базовых возможностей системы человек может забыть команду для исполнения какой-либо операции; в этом смысле графические интерфейсы, где все возможности перед глазами, оказываются предпочтительны.

Утверждение о том, что GUI ограничивает возможности пользователя заранее предопределенными возможностями просто не соответствует действительности: хорошо продуманные интерфейсы обеспечивают почти такую же гибкость в комбинации операций, как и командные языки. Возможность же записывать и вновь проигрывать последовательности действий во многих ситуациях может отлично заменить командные файлы.

Относительно высокие накладные расходы, неразрывно связанные с графическими интерфейсами, не являются таким уж большим злом. Современные цены на аппаратуру достаточно низки для того, чтобы бóльшая эффективность изучения и использования графической системы быстро себя окупáла. Кроме того, многие современные настольные системы используются для приложений, которые сами по себе требуют высококачественной графики и большой вычислительной мощности: САПР, полиграфические работы, синтез и обработка видеоданных и т.д. На фоне потребностей таких приложений накладные расходы, связанные с графическим интерфейсом, оказываются просто не стоящими упоминания.

Еще одно немаловажное обстоятельство: хорошо продуманный графический интерфейс с правильно подобранными цветами, красиво нарисованными управляющими элементами окон и т.д., просто сам по себе приятен для глаз.

10.3 Попытка сделать выводы

Изложив, в соответствии с правилами диалектики, тезис и антитезис, попробуем перейти к синтезу. Излагая аргументы сторон, мы намеренно выпустили ряд убийственных доводов типа таких: “Вы, сторонники CLI, хотите всех нас вернуть в каменный век и заставить пользоваться перфокартами” или “разработайте систему, которой сможет пользоваться даже идиот – и только идиот захочет ей пользоваться”.

Если попытаться непредвзято сравнить доводы, становится видно, что основное противоречие состоит во взглядах на сложность изучения командного языка: сторонники CLI считают его простым делом, в то время как для сторонников GUI это едва ли не основное препятствие к использованию компьютера[1].

[1] По наблюдениям авторов, программистам и сторонникам командной строки легче среднего дается изучение ино-

Возможно, такое различие свидетельствует о том, что разные категории пользовательских интерфейсов предпочтительны для людей с разным складом мышления. Например, можно предположить, что командные интерфейсы удобнее для людей с логическим складом мышления, а графические – с образным. Отчасти это подтверждается тем, что ориентированные на GUI компьютеры Macintosh в основном используются художниками, дизайнерами и другими представителями "образных" творческих профессий.

К сожалению, большинство открытых исследований психологических аспектов человеко-машинного взаимодействия либо до ужаса непрофессиональны, либо финансируются фирмами, продающими программное обеспечение. Такие "исследования" обычно лишь подтверждают, что продукт фирмы X на 50 % быстрее при исполнении "типичных", что бы под этим ни подразумевалось, задач, чем продукт конкурирующей фирмы Y.

Авторы не располагают ни ресурсами, ни квалификацией, необходимыми для проведения объективного независимого исследования, однако есть основания утверждать, что командные и графические интерфейсы оптимальны в разных ситуациях и для разных целей. Командный интерфейс хорош, когда пользователь ясно представляет себе, чего он хочет, а особенно для автоматизации регулярно исполняемых рутинных задач. Графические же интерфейсы удобнее при решении нечетко сформулированных или плохо алгоритмизуемых проблем.

Поэтому хорошая система должна предоставлять *оба* интерфейса. Например, разработчики фирмы Apple долгое время пытались избежать включения в систему командного интерпретатора но в конце концов под давлением пользователей и особенно специалистов по технической поддержке они были вынуждены реализовать командный язык AppleScript.

В свете этого представляется, что утверждения, которые часто приходится слышать в спорах между сторонниками различных категорий интерфейсов – "CLI безнадежно устарели" с одной стороны и "GUI никому не нужны и вообще это пустая трата машинных ресурсов" с другой – свидетельствуют либо о чрезмерном накале спора, либо о том, что говорящий не представляет себе всей ситуации или попросту глуп. Впрочем, если такая фраза произносится в беседе между продавцом и покупателем, это может свидетельствовать об остром желании продать именно свой продукт или, напротив, о желании опорочить систему, предлагаемую конкурентами.

странных языков. Мы не можем сказать, что здесь является причиной, а что – следствием.

Литература

[1] Barron D. W., Computer Operating Systems, Chapman and Hall, London, 1971.

[2] Кнут Дональд Э., Искусство программирования для ЭВМ. Мир, 1976

[3] С. Новосельцев, предисловие к ст. Мир Apple, //КомпьютерПресс, 1993, no.11.

[4] Фирменное руководство по Zortech C/C++ V3.x

[5] Хайнлайн Р, Космический Рейнджер, пиратские издания.

[6] Краковяк С., Основы организации и функционирования ОС ЭВМ, М, Мир, 1988.

[7] Дейкстра Э., Дисциплина программирования. М, Мир, 1978.

[8] The Transputer Databook, INMOS document number: 72 TRN 203 02.

[9] Транспьютеры. Архитектура и программное обеспечение. /Под ред. Г. Харпа. М, Радио и связь, 1993.

[10] Сиборг

[11] Малые ЭВМ высокой производительности. Архитектура и программирование. /Под ред. Н. Л. Прохорова. М, Радио и связь, 1990.

[12] Красная книжка про POSIX

[13] Результаты Tmpc

[14] Johnson Michael K., The Linux Kernel Hacker Guide. `ftp://tsx11.mit.edu/pub/linux/docs/LPD/khg-0.6.tar`

[15] Дейтел Г., Операционные системы, М, Мир, 1984.

[16] Брайан Проффит. Высокопроизводительная файловая система HPFS,//PC Magazine russian edition, 1995, no.10.

[17] PC Magazine

[18] Дейт К. Руководство по реляционной СУБД DB2. Финансы и Статистика, 1988.

[19] Исида Х. Программирование для микрокомпьютеров, М, Мир, 1988.

[20] Паппас К., Марри У., Микропроцессор 80386, М, Радио и Связь, 1993.

[21] RSA's FAQ About Today's Cryptography: RSA. http://www.rsa.com/rsalabs/faq/faq_rsa.html

[22] Ssh (Secure Shell) Home Page. http://www.cs.hut.fi/ssh/

[23] Моисеенков И. Суета вокруг Роберта или Моррис-сын и все, все, все,//КомпьютерПресс, 1991, no.8,9.

[24] Органик С. Организация системы Интел 432. М, Мир, 1987.

[25] Дейтел Г., Введение в операционные системы. М, Мир, 1987.

Оглавление

Т.Б.Большаков Д.В.Иртегов

Операционные системы

Часть II

Учебное пособие

Подписано в печать	Формат 60x84/16.
Печать офсетная	Уч.-изд.л.
Заказ # Тираж экз	Цена р.

Редакционно-издательский отдел Новосибирского Университета;
участок оперативной полиграфии НГУ; 630090, Новосибирск 90,
ул. Пирогова, 2