Лекции по дисциплине ОП.01 Операционные системы и среды

Специальность 090207 Информационные системы и программирование

Нальчик, 2023г.

Оглавление

Введение Лекция 1. Место операционной системы в структуре программного обеспечения.

Современный компьютер состоит из одного или нескольких процессоров, оперативной памяти, дисков, принтера, клавиатуры, мыши, дисплея, сетевых интерфейсов и других разнообразных устройств ввода-вывода. В итоге получается довольно сложная система. Если каждому программисту, создающему прикладную программу, нужно будет разбираться во всех тонкостях работы всех этих устройств, то он не напишет ни строчки кода.

Более того, управление всеми этими компонентами и их оптимальное использование представляет собой очень непростую задачу. По этой причине компьютеры оснащены специальным уровнем программного обеспечения, который называется операционной системой, в чью задачу входит управление пользовательскими программами, а также всеми ранее упомянутыми ресурсами. Именно такие системы и являются предметом рассмотрения данной книги.

Наверное, уже имеете некоторый опыт работы с такими операционными системами, как Windows, Linux, FreeBSD или Max OS X, но их внешний облик может быть разным. Программы, с которыми взаимодействуют пользователи, обычно называемые оболочкой, когда они основаны на применении текста, и графическим пользовательским интерфейсом (Graphical User Interface (GUI)), когда в них используются значки, фактически не являются частью операционной системы, хотя задействуют эту систему в своей работе.

Схематично основные рассматриваемые здесь компоненты представлены на рис. 0-1

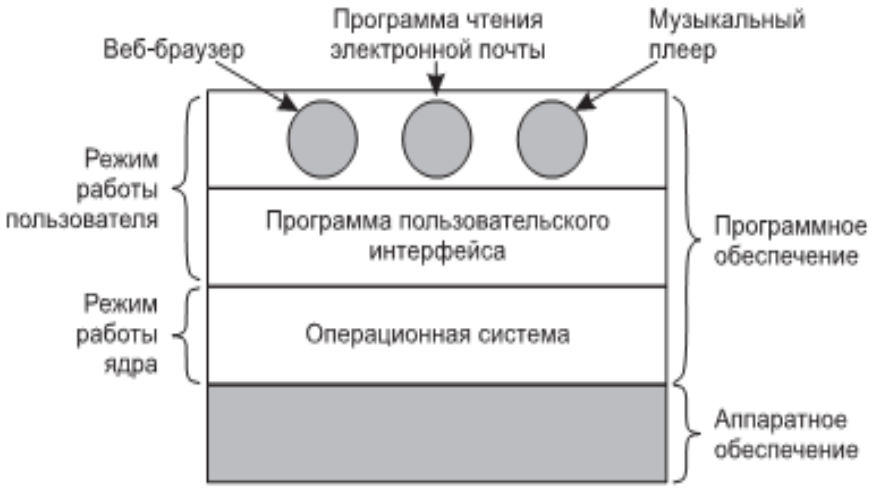

Рисунок 1 Место операционной системы в структуре программного обеспечения

В нижней части рисунка показано аппаратное обеспечение. Оно состоит из микросхем, плат, дисков, клавиатуры, монитора и других физических объектов. Над аппаратным обеспечением находится программное обеспечение. Большинство компьютеров имеют два режима работы: режим ядра и режим пользователя. Операционная система — наиболее фундаментальная часть программного обеспечения, работающая в режиме ядра (этот режим называют еще режимом супервизора). В этом режиме она имеет полный доступ ко всему аппаратному обеспечению и может задействовать любую инструкцию, которую машина в состоянии выполнить. Вся остальная часть программного обеспечения работает в режиме пользователя, в котором доступно лишь подмножество инструкций машины. В частности, программам, работающим в режиме пользователя, запрещено использование инструкций, управляющих машиной или осуществляющих операции ввода-вывода (Input/Output — I/O). К различиям между режимами ядра и пользователя мы еще не раз вернемся на страницах этой книги. Эти различия оказывают решающее влияние на порядок работы операционной системы.

Программы пользовательского интерфейса — оболочка или GUI — находятся на самом низком уровне программного обеспечения, работающего в режиме пользователя, и позволяют пользователю запускать другие программы, такие как веб-браузер, программа чтения электронной почты или музыкальный плеер. Эти программы также активно пользуются операционной системой.

Местонахождение операционной системы показано на рис. 0-1. Она работает непосредственно с аппаратным обеспечением и является основой остального программного обеспечения.

Важное отличие операционной системы от обычного (работающего в режиме пользователя) программного обеспечения состоит в следующем: если пользователь недоволен конкретной программой чтения электронной почты, то он может выбрать другую программу или, если захочет, написать собственную программу, но не может написать собственный обработчик прерываний системных часов, являющийся частью операционной системы и защищенный на аппаратном уровне от любых попыток внесения изменений со стороны пользователя. Это различие иногда не столь четко выражено во встроенных системах (которые могут не иметь режима ядра) или интерпретируемых системах (таких, как системы, построенные на основе языка Java, в которых для разделения компонентов используется не аппаратное обеспечение, а интерпретатор).

Во многих системах также есть программы, работающие в режиме пользователя, но помогающие работе операционной системы или выполняющие особые функции. К примеру, довольно часто встречаются программы, позволяющие пользователям изменять их пароли. Они не являются частью операционной системы и не работают в режиме ядра, но всем понятно, что они выполняют важную функцию и должны быть особым образом защищены. В некоторых системах эта идея доведена до крайней формы, и те области, которые

традиционно относились к операционной системе (например, файловая система), работают в пространстве пользователя. В таких системах трудно провести четкую границу. Все программы, работающие в режиме ядра, безусловно, являются частью операционной системы, но некоторые программы, работающие вне этого режима, возможно, также являются ее частью или, по крайней мере, имеют с ней тесную связь.

Операционные системы отличаются от пользовательских программ (то есть приложений) не только местоположением. Их особенности — довольно большой объем, сложная структура и длительные сроки использования. Исходный код основы операционной системы типа Linux или Windows занимает порядка 5 млн строк. Чтобы представить себе этот объем, давайте мысленно распечатаем 5 млн строк в книжном формате по 50 строк на странице и по 1000 страниц в каждом томе (что больше этой книги). Чтобы распечатать такое количество кода, понадобится 100 томов, а это практически целая книжная полка. Можете себе представить, что вы получили задание по поддержке операционной системы и в первый же день ваш начальник подвел вас к книжной полке и сказал: «Вот это все нужно выучить». И это касается только той части, которая работает в режиме ядра. При включении необходимых общих библиотек объем Windows превышает 70 млн строк кода (напечатанные на бумаге, они займут 10–20 книжных полок), и это не считая основных прикладных программ (таких, как Windows Explorer, Windows Media Player и т. д.).

Теперь понятно, почему операционные системы живут так долго, — их очень трудно создавать, и, написав одну такую систему, владелец не испытывает желания ее выбросить и приступить к созданию новой. Поэтому операционные системы развиваются в течение долгого периода времени. Семейство Windows 95/98/Me по своей сути представляло одну операционную систему, а семейство Windows NT/2000/XP/Vista/ Windows 7 — другую. Для пользователя они были похожи друг на друга, поскольку Microsoft позаботилась о том, чтобы пользовательский интерфейс Windows 2000/XP/Vista/Windows 7 был очень похож на ту систему, которой он шел на замену, а чаще всего это была Windows 98. Тем не менее у Microsoft были довольно веские причины, чтобы избавиться от Windows 98. Другим примером, будет операционная система UNIX, ее варианты и клоны. Она также развивалась в течение многих лет, существуя в таких базирующихся на исходной системе версиях, как System V, Solaris и FreeBSD.

Раздел 1. Основы теории операционных систем Тема 1.1. История, назначение и функции операционных систем

Лекция 2. Операционная система как расширенная машина. Операционная система в качестве менеджера ресурсов.

Архитектура большинства компьютеров (система команд, организация памяти, ввод-вывод данных и структура шин) на уровне машинного языка слишком примитивна и неудобна для использования в программах, особенно это касается систем ввода-вывода. Рассмотрим современные жесткие диски SATA (Serial ATA), используемые на большинстве компьютеров. Оборудованием занимается та часть программного обеспечения, которая называется драйвером диска и предоставляет, не вдаваясь в детали, интерфейс для чтения и записи дисковых блоков. Операционные системы содержат множество драйверов для управления устройствами ввода-вывода.

Но для большинства приложений слишком низким является даже этот уровень. Поэтому все операционные системы предоставляют еще один уровень абстракции для использования дисков — файлы. Используя эту абстракцию, программы могут создавать, записывать и читать файлы, не вникая в подробности реальной работы оборудования.

Файл представляет собой полезный объем информации, скажем, цифровую фотографию, сохраненное сообщение электронной почты или веб-страницу. Работать с фотографиями, сообщениями электронной почты и веб-страницами намного легче, чем с особенностями SATA-дисков (или других дисковых устройств). Задача операционной системы заключается в создании хорошей абстракции, а затем в реализации абстрактных объектов, создаваемых в рамках этой абстракции, и управлении ими.

Одна из главных задач операционной системы — скрыть аппаратное обеспечение и существующие программы (и их разработчиков) под создаваемыми взамен них и приспособленными для нормальной работы красивыми, элегантными, неизменными абстракциями. Операционные системы превращают уродство в красоту (рис. 1.2).

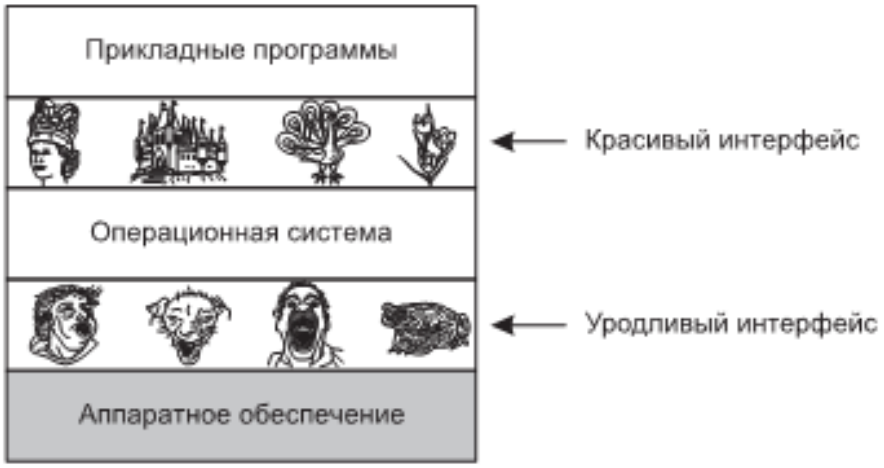

Рисунок 2 Операционная система превращает уродливое аппаратное обеспечение в красивые абстракции

Следует отметить, что реальными «заказчиками» операционных систем являются прикладные программы (разумеется, не без помощи прикладных программистов).

Именно они непосредственно работают с операционной системой и ее абстракциями.

А конечные пользователи работают с абстракциями, предоставленными пользовательским интерфейсом, — это или командная строка оболочки, или графический интерфейс.

Абстракции пользовательского интерфейса могут быть похожими на абстракции, предоставляемые операционной системой, но так бывает не всегда. Чтобы пояснить это положение, рассмотрим обычный рабочий стол Windows и командную строку.

И то и другое — программы, работающие под управлением операционной системы Windows и использующие предоставленные этой системой абстракции, но они предлагают существенно отличающиеся друг от друга пользовательские интерфейсы. Точно так же пользователи Linux, работающие в Gnome или KDE, видят совершенно иной интерфейс, чем пользователи Linux, работающие в X Window System, но положенные в основу абстракции операционной системы в обоих случаях одни и те же.

Представление о том, что операционная система главным образом предоставляет абстракции для прикладных программ, — это взгляд сверху вниз. Сторонники альтернативного взгляда, снизу вверх, придерживаются того мнения, что операционная система существует для управления всеми частями сложной системы. Современные компьютеры состоят из процессоров, памяти, таймеров, дисков, мышей, сетевых интерфейсов, принтеров и широкого спектра других устройств. Сторонники взгляда снизу вверх считают, что задача операционной системы заключается в обеспечении упорядоченного и управляемого распределения процессоров, памяти и устройств ввода-вывода между различными программами, претендующими на их использование.

Современные операционные системы допускают одновременную работу нескольких программ. Представьте себе, что будет, если все три программы, работающие на одном и том же компьютере, попытаются распечатать свои выходные данные одновременно на одном и том же принтере. Первые несколько строчек распечатки могут быть от программы № 1, следующие несколько строчек — от программы № 2, затем несколько строчек от программы № 3 и т. д. В результате получится полный хаос. Операционная система призвана навести

порядок в потенциально возможном хаосе за счет буферизации на диске всех выходных данных, предназначенных для принтера. После того как одна программа закончит свою работу, операционная система сможет скопировать ее выходные данные с файла на диске, где они были сохранены, на принтер, а в то же самое время другая программа может продолжить генерацию данных, не замечая того, что выходные данные фактически (до поры до времени) не попадают на принтер.

Когда с компьютером (или с сетью) работают несколько пользователей, потребности в управлении и защите памяти, устройств ввода-вывода и других ресурсов значительно возрастают, поскольку иначе пользователи будут мешать друг другу работать. Кроме этого, пользователям часто требуется совместно использовать не только аппаратное обеспечение, но и информацию (файлы, базы данных и т. п.). Короче говоря, сторонники этого взгляда на операционную систему считают, что ее первичной задачей является отслеживание того, какой программой какой ресурс используется, чтобы удовлетворять запросы на использование ресурсов, нести ответственность за их использование и принимать решения по конфликтующим запросам от различных программ и пользователей.

Управление ресурсами включает в себя мультиплексирование (распределение) ресурсов двумя различными способами: во времени и в пространстве. Когда ресурс разделяется во времени, различные программы или пользователи используют его по очереди: сначала ресурс получают в пользование одни, потом другие и т. д. К примеру, располагая лишь одним центральным процессором и несколькими программами, стремящимися на нем выполняться, операционная система сначала выделяет центральный процессор одной программе, затем, после того как она уже достаточно поработала, центральный процессор получает в свое распоряжение другая программа, затем еще одна программа, и, наконец, его опять получает в свое распоряжение первая программа. Определение того, как именно ресурс будет разделяться во времени — кто будет следующим потребителем и как долго, — это задача операционной системы. Другим примером мультиплексирования во времени может послужить совместное использование принтера. Когда в очереди для распечатки на одном принтере находятся несколько заданий на печать, нужно принять решение, какое из них будет выполнено следующим.

Другим видом разделения ресурсов является пространственное разделение. Вместо поочередной работы каждый клиент получает какую-то часть разделяемого ресурса.

Например, оперативная память обычно делится среди нескольких работающих программ, так что все они одновременно могут постоянно находиться в памяти (например, используя центральный процессор по очереди). При условии, что памяти достаточно для хранения более чем одной программы, эффективнее разместить в памяти сразу несколько программ, чем выделять всю память одной программе, особенно если ей нужна лишь небольшая часть от общего пространства. Разумеется, при этом возникают проблемы равной доступности, обеспечения безопасности и т. д., и их должна решать операционная система. Другим ресурсом с разделяемым пространством является жесткий диск. На многих системах на одном и том же диске могут одновременно храниться файлы, принадлежащие многим пользователям. Распределение дискового пространства и отслеживание того, кто какие дисковые блоки использует, — это типичная задача операционной системы по управлению ресурсами.

Лекция 3. История операционных систем: первое поколение (1945–1955): электронные лампы; второе поколение (1955–1965): транзисторы и системы пакетной обработки; третье поколение (1965–1980): интегральные схемы и многозадачность;

1) На заре компьютерной эры каждую машину проектировала, создавала, программировала, эксплуатировала и обслуживала одна и та же группа людей (как правило, инженеров). Все программирование велось исключительно на машинном языке или, и того хуже, за счет сборки электрических схем, а для управления основными функциями машины приходилось подключать к коммутационным панелям тысячи проводов. О языках программирования (даже об ассемблере) тогда еще ничего не было известно. Об операционных системах вообще никто ничего не слышал.

2) В середине 1950-х годов изобретение и применение транзисторов радикально изменило всю картину. Компьютеры стали достаточно надежными, появилась высокая вероятность того, что машины будут работать довольно долго, выполняя при этом полезные функции. Впервые сложилось четкое разделение между проектировщиками, сборщиками, операторами, программистами и обслуживающим персоналом.

Чтобы выполнить задание (то есть программу или комплект программ), программист сначала должен был записать его на бумаге (на Фортране или ассемблере), а затем перенести на перфокарты. После этого он должен был принести колоду перфокарт в комнату ввода данных, передать одному из операторов и идти пить кофе в ожидании, когда будет готов результат.

Если учесть высокую стоимость оборудования, неудивительно, что люди довольно скоро занялись поиском способа повышения эффективности использования машинного времени. Общепринятым решением стала система пакетной обработки. Первоначально замысел состоял в том, чтобы собрать полный поднос заданий (колод перфокарт) в комнате входных данных и затем переписать их на магнитную ленту, используя небольшой и (относительно) недорогой компьютер, например IBM 1401, который был очень хорош для считывания карт, копирования лент и печати выходных данных, но не подходил для числовых вычислений.

Другие, более дорогостоящие машины, такие как IBM 7094, использовались для настоящих вычислений. Этот процесс изображен на рис. 3.

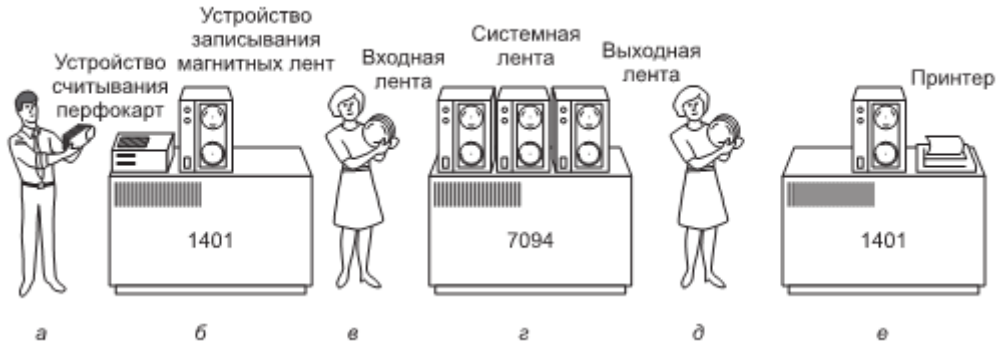

Рисунок 3 Ранняя система пакетной обработки: а — программист приносит карты для IBM 1401; б — IBM 1401 записывает пакет заданий на магнитную ленту; в — оператор переносит входные данные на ленте к IBM 7094; г — IBM 7094 выполняет вычисления; д — оператор переносит ленту с выходными данными на IBM 1401; е — IBM 1401 печатает выходные данные

Структура типичного входного задания показана на рис. 4.

Оно начиналось с карты $JOB, на которой указывались максимальное время выполнения задания в минутах, загружаемый учетный номер и имя программиста. Затем поступала карта $FORTRAN, дающая операционной системе указание загрузить компилятор языка Фортран с системной магнитной ленты. За этой картой следовала программа, которую нужно было компилировать, а после нее — карта $LOAD, указывающая операционной системе загрузить только что скомпилированную объектную программу. Следом шла карта $RUN, дающая операционной системе команду на выполнение программы с использованием данных, следующих за ней. Наконец, карта завершения $END отмечала конец задания. Эти примитивные управляющие перфокарты были предшественниками современных оболочек и интерпретаторов командной строки.

Рисунок 4. Структура типичного задания для операционной системы FMS

Большие компьютеры второго поколения использовались главным образом для научных и технических вычислений, таких как решение дифференциальных уравнений в частных производных, часто встречающихся в физике и инженерных задачах. В основном программы для них составлялись на языке Фортран и ассемблере, а типичными операционными системами были FMS (Fortran Monitor System) и IBSYS (операционная система, созданная корпорацией IBM для компьютера IBM 7094).

3) К началу 1960-х годов большинство производителей компьютеров имели два различных, не совместимых друг с другом семейства.

Семейство компьютеров IBM/360 стало первой основной серией, использующей малые интегральные схемы, дававшие преимущество в цене и качестве по сравнению с машинами второго поколения, собранными на отдельных транзисторах.

Решение этой проблемы заключалось в разбиении памяти на несколько частей, называемых разделами, в каждом из которых выполнялось отдельное задание (рис. 5).

Пока одно задание ожидало завершения работы устройства ввода-вывода, другое могло использовать центральный процессор. Если в оперативной памяти содержалось достаточное количество заданий, центральный процессор мог быть загружен почти на все 100 % времени. Множество одновременно хранящихся в памяти заданий требовало наличия специального оборудования для защиты каждого задания от возможного незаконного присваивания областей памяти и нанесения вреда со стороны других заданий. Для этой цели компьютеры 360-й серии и другие системы третьего поколения были оборудованы специальными аппаратными средствами.

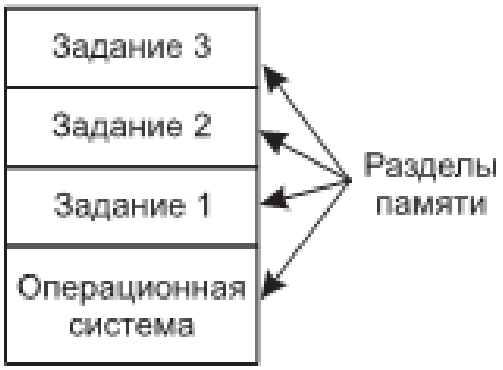

Рисунок 5. Многозадачная система с тремя заданиями в памяти

Другим важным плюсом операционных систем третьего поколения стала способность считывать задание с перфокарт на диск по мере того, как их приносили в машинный зал. При окончании выполнения каждого текущего задания операционная система могла загружать новое задание с диска в освободившийся раздел памяти и запускать это задание. Этот технический прием называется подкачкой данных, или спулингом (spooling — английское слово, которое произошло от Simultaneous Peripheral Operation On Line, то есть совместная периферийная операция в интерактивном режиме), и его также используют для выдачи полученных данных.

Лекция 4. История операционных систем: четвертое поколение (с 1980 года по наши дни): персональные компьютеры; пятое поколение (с 1990 года по наши дни): мобильные компьютеры

4) Следующий период эволюции операционных систем связан с появлением БИС — больших интегральных схем (LSI, Large Scale Integration) — кремниевых микросхем, содержащих тысячи транзисторов на одном квадратном сантиметре. С точки зрения архитектуры персональные компьютеры (первоначально называемые микрокомпьютерами) были во многом похожи на мини-компьютеры класса PDP-11, но, конечно же, отличались по цене. Если появление мини-компьютеров позволило отделам компаний и факультетам университетов иметь собственный компьютер, то с появлением микропроцессоров возможность купить персональный компьютер получил каждый человек.

В начале 1980-х корпорация IBM разработала IBM PC (Personal Computer — персональный компьютер), и начала искать для него программное обеспечение. Сотрудники IBM связались с Биллом Гейтсом, чтобы получить лицензию на право использования его интерпретатора языка Бейсик. Они также поинтересовались, не знает ли он операционную систему, которая работала бы на IBM PC. Гейтс посоветовал обратиться к Digital Research, тогда главенствующей компании в области операционных систем.

Но Килдэлл отказался встречаться с IBM, послав вместо себя своего подчиненного. Что еще хуже, его адвокат даже отказался подписывать соглашение о неразглашении, касающееся еще не выпущенного IBM PC, чем полностью испортил дело. Корпорация IBM снова обратилась к Гейтсу с просьбой обеспечить ее операционной системой. После повторного обращения Гейтс выяснил, что у местного изготовителя компьютеров, Seattle Computer Products, есть подходящая операционная система DOS (Disk Operating System — дисковая операционная система). Он направился в эту компанию с предложением выкупить DOS (предположительно за $50 000), которое компания Seattle Computer Products с готовностью приняла. Затем Гейтс создал пакет программ DOS/BASIC, и пакет был куплен IBM. Когда корпорация IBM захотела внести в операционную систему ряд усовершенствований, Билл Гейтс пригласил для этой работы Тима Патерсона (Tim Paterson), человека, написавшего DOS и ставшего первым служащим Microsoft — еще не оперившейся компании Гейтса. Видоизмененная система была переименована в MS-DOS (MicroSoft Disk Operating System) и быстро заняла доминирующее положение на рынке IBM PC. Самым важным оказалось решение Гейтса (как оказалось, чрезвычайно мудрое) продавать MS-DOS компьютерным компаниям для установки вместе с их оборудованием в отличие от попыток Килдэлла продавать CP/M конечным пользователям (по крайней мере, на начальной стадии).

CP/M, MS-DOS и другие операционные системы для первых микрокомпьютеров полностью основывались на командах, вводимых пользователем с клавиатуры. Со временем благодаря исследованиям, проведенным в 1960-е годы Дагом Энгельбартом (Doug Engelbart) в научно-исследовательском институте Стэнфорда (Stanford Research Institute), ситуация изменилась. Энгельбарт изобрел графический интерфейс пользователя (GUI, Graphical User Interface) вкупе с окнами, значками, системами меню и мышью. Эту идею переняли исследователи из Xerox PARC и воспользовались ею в создаваемых ими машинах.

В 1999 году компания Apple позаимствовала ядро, происходящее из микроядра Mach, первоначально разработанного специалистами университета Карнеги — Меллона для замены ядра BSD UNIX. Поэтому Mac OS X является операционной системой, построенной на основе UNIX, хотя и с весьма своеобразным интерфейсом.

Другой операционной системой Microsoft была Windows NT (NT означает New Technology — новая технология), которая на определенном уровне совместима с Windows 95.

Но полностью этим планам также не суждено было сбыться, поэтому Microsoft выпустила еще одну версию Windows 98 под названием Windows Me (Millennium edition — выпуск тысячелетия). В 2001 году была выпущена слегка обновленная версия Windows 2000, названная Windows XP. Эта версия выпускалась намного дольше, по существу заменяя все предыдущие версии Windows.

Хотя многие пользователи UNIX, особенно опытные программисты, отдают предпочтение интерфейсу на основе командной строки, практически все UNIX-системы поддерживают систему управления окнами X Window System (или X11), созданную в Массачусетском технологическом институте. Эта система выполняет основные операции по управлению окнами, позволяя пользователям создавать, удалять, перемещать окна и изменять их размеры, используя мышь. Зачастую в качестве надстройки над X11 можно использовать полноценный графический пользовательский интерфейс, например Gnome или KDE, придавая UNIX внешний вид и поведение, чем-то напоминающие Macintosh или Microsoft Windows.

В середине 1980-х годов начало развиваться интересное явление — рост сетей персональных компьютеров, работающих под управлением сетевых операционных систем и распределенных операционных систем (Tanenbaum and Van Steen, 2007). В сетевых операционных системах пользователи знают о существовании множества компьютеров и могут войти в систему удаленной машины и скопировать файлы с одной машины на другую. На каждой машине работает своя локальная операционная система и имеется собственный локальный пользователь (или пользователи).

5) С тех пор как в комиксах 1940-х годов детектив Дик Трейси стал переговариваться с помощью радиостанции, вмонтированной в наручные часы, у людей появилось желание иметь в своем распоряжении устройство связи, которое можно было бы брать с собой в любое место. Первый настоящий мобильный телефон появился в 1946 году, и тогда он весил около 40 кг. Его можно было брать с собой только при наличии автомобиля, в котором его можно было перевозить.

Первый по-настоящему переносной телефон появился в 1970-х годах и при весе приблизительно 1 кг был воспринят весьма позитивно. Его ласково называли «кирпич».

Хотя идея объединения в одном устройстве и телефона и компьютера вынашивалась еще с 1970-х годов, первый настоящий смартфон появился только в середине 1990-х годов, когда Nokia выпустила свой N9000, представлявший собой комбинацию из двух отдельных устройств: телефона и КПК. В 1997 году в компании Ericsson для ее изделия GS88 «Penelope» был придуман термин «смартфон».

В первое десятилетие после своего появления большинство смартфонов работало под управлением Symbian OS. Эту операционную систему выбрали такие популярные бренды, как Samsung, Sony Ericsson, Motorola и Nokia. Но долю рынка Symbian начали отбирать другие операционные системы, например RIM Blackberry OS (выпущенная для смартфонов в 2002 году) и Apple iOS (выпущенная для первого iPhone в 2007 году).

Для производителей телефонов Android обладала тем преимуществом, что имела открытый исходный код и была доступна по разрешительной лицензии. В результате компании получили возможность без особого труда подстраивать ее под свое собственное оборудование. Кроме того, у этой операционной системы имеется огромное сообщество разработчиков, создающих приложения в основном на общеизвестном языке программирования Java.

Лекция 5. Виды операционных систем: операционные системы мейнфреймов, серверные операционные системы, многопроцессорные операционные системы, операционные системы персональных компьютеров, операционные системы карманных персональных компьютеров, встроенные операционные системы, операционные системы сенсорных узлов, операционные системы реального времени, операционные системы смарт-карт.

Операционные системы мейнфреймов ориентированы преимущественно на одновременную обработку множества заданий, большинство из которых требует колоссальных объемов ввода-вывода данных. Обычно они предлагают три вида обслуживания: пакетную обработку, обработку транзакций и работу в режиме разделения времени.

Пакетная обработка — это одна из систем обработки стандартных заданий без участия пользователей. В пакетном режиме осуществляется обработка исков в страховых компаниях или отчетов о продажах сети магазинов. Системы обработки транзакций справляются с большим количеством мелких запросов, к примеру обработкой чеков в банках или бронированием авиабилетов. Каждая элементарная операция невелика по объему, но система может справляться с сотнями и тысячами операций в секунду.

Серверные операционные системы. Чуть ниже по уровню стоят серверные операционные системы. Они работают на серверах, которые представлены очень мощными персональными компьютерами, рабочими станциями или даже универсальными машинами. Они одновременно обслуживают по сети множество пользователей, обеспечивая им общий доступ к аппаратным и программным ресурсам. Серверы могут предоставлять услуги печати, хранения файлов или веб-служб. Интернет-провайдеры для обслуживания своих клиентов обычно задействуют сразу несколько серверных машин. При обслуживании веб-сайтов серверы хранят веб-страницы и обрабатывают поступающие запросы. Типичными представителями серверных операционных систем являются Solaris, FreeBSD, Linux и Windows Server 201x.

Многопроцессорные операционные системы. С появлением многоядерных процессоров для персональных компьютеров операционные системы даже обычных настольных компьютеров и ноутбуков стали работать по меньшей мере с небольшой многопроцессорной системой. Со временем, похоже, число ядер будет только расти. К счастью, за годы предыдущих исследований были накоплены обширные знания о многопроцессорных операционных системах, и использование этого арсенала в многоядерных системах не должно вызвать особых осложнений. Труднее всего будет найти приложения, которые смогли бы использовать всю эту вычислительную мощь. На многопроцессорных системах могут работать многие популярные операционные системы, включая Windows и Linux.

Операционные системы персональных компьютеров. К следующей категории относятся операционные системы персональных компьютеров. Все их современные представители поддерживают многозадачный режим. При этом довольно часто уже в процессе загрузки на одновременное выполнение запускаются десятки программ. Задачей операционных систем персональных компьютеров является качественная поддержка работы отдельного пользователя. Они широко используются для обработки текстов, создания электронных таблиц, игр и доступа к Интернету. Типичными примерами могут служить операционные системы Linux, FreeBSD, Windows 7, Windows 8 и OS X компании Apple. Операционные системы персональных компьютеров известны настолько широко, что в особом представлении не нуждаются. По сути, многим людям даже невдомек, что существуют другие разновидности операционных систем.

Операционные системы карманных персональных компьютеров. Продолжая двигаться по нисходящей ко все более простым системам, мы дошли до планшетов, смартфонов и других карманных компьютеров. Эти компьютеры, изначально известные как КПК, или PDA (Personal Digital Assistant — персональный цифровой секретарь), представляют собой небольшие компьютеры, которые во время работы держат в руке. Самыми известными их представителями являются смартфоны и планшеты. Как уже говорилось, на этом рынке доминируют операционные системы Android от Google и iOS от Apple, но у них имеется множество конкурентов. Большинство таких устройств могут похвастаться многоядерными процессорами, GPS, камерами и другими датчиками, достаточным объемом памяти и

сложными операционными системами. Более того, у всех них имеется больше сторонних приложений (apps) для USB-носителей, чем вы себе можете представить.

Встроенные операционные системы. Встроенные системы работают на компьютерах, которые управляют различными устройствами. Поскольку на этих системах установка пользовательских программ не предусматривается, их обычно компьютерами не считают. Примерами устройств, где устанавливаются встроенные компьютеры, могут послужить микроволновые печи, телевизоры, автомобили, пишущие DVD, обычные телефоны и MP3-плееры. В основном встроенные системы отличаются тем, что на них ни при каких условиях не будет работать стороннее программное обеспечение. В микроволновую печь невозможно загрузить новое приложение, поскольку все ее программы записаны в ПЗУ. Следовательно, отпадает необходимость в защите приложений друг от друга и операционную систему можно упростить. Наиболее популярными в этой области считаются операционные системы Embedded Linux, QNX и VxWorks.

Операционные системы сенсорных узлов. Сети, составленные из миниатюрных сенсорных узлов, связанных друг с другом и с базовой станцией по беспроводным каналам, развертываются для различных целей. Такие сенсорные сети используются для защиты периметров зданий, охраны государственной границы, обнаружения возгораний в лесу, измерения температуры и уровня осадков в целях составления прогнозов погоды, сбора информации о перемещениях противника на поле боя и многого другого.

Операционные системы реального времени. Еще одна разновидность операционных систем — это системы реального времени. Эти системы характеризуются тем, что время для них является ключевым параметром. Например, в системах управления производственными процессами компьютеры, работающие в режиме реального времени, должны собирать сведения о процессе и использовать их для управления станками на предприятии. Довольно часто они должны отвечать очень жестким временным требованиям. Например, когда автомобиль перемещается по сборочному конвейеру, то в определенные моменты времени должны осуществляться вполне конкретные операции. Если, к примеру, сварочный робот приступит к сварке с опережением или опозданием, машина придет в негодность. Если операция должна быть проведена точно в срок (или в определенный период времени), то мы имеем дело с системой жесткого реального времени. Множество подобных систем встречается при управлении производственными процессами, в авиационно-космическом электронном оборудовании, в военной и других подобных областях применения. Эти системы должны давать абсолютные гарантии того, что определенные действия будут осуществляться в конкретный момент времени. Другой разновидностью подобных систем является система мягкого реального времени, в которой хотя и нежелательно, но вполне допустимо несоблюдение срока какого-нибудь действия, что не наносит непоправимого вреда. К этой категории относятся цифровые аудио- или мультимедийные системы. Смартфоны также являются системами мягкого реального времени.

Операционные системы смарт-карт. Самые маленькие операционные системы работают на смарт-картах. Смарт-карта представляет собой устройство размером с кредитную карту, имеющее собственный процессор. На операционные системы для них накладываются очень жесткие ограничения по требуемой вычислительной мощности процессора и объему памяти. Некоторые из смарт-карт получают питание через контакты считывающего устройства, в которое вставляются, другие — бесконтактные смарт-карты — получают питание за счет эффекта индукции, что существенно ограничивает их возможности. Некоторые из них способны справиться с одной-единственной функцией, например с электронными платежами, но существуют и многофункциональные смарт-карты. Зачастую они являются патентованными системами. Некоторые смарт-карты рассчитаны на применение языка Java.

Лекция 6. Понятия операционной системы: процессы, адресные пространства, адресные пространства, файлы, ввод-вывод данных, безопасность, оболочка

Процессы. Ключевым понятием во всех операционных системах является процесс. Процессом, по существу, является программа во время ее выполнения. С каждым процессом связано его адресное пространство — список адресов ячеек памяти от нуля до некоторого

максимума, откуда процесс может считывать данные и куда может записывать их. Адресное пространство содержит выполняемую программу, данные этой программы и ее стек. Кроме этого, с каждым процессом связан набор ресурсов, который обычно включает регистры (в том числе счетчик команд и указатель стека), список открытых файлов, необработанные предупреждения, список связанных процессов и всю остальную информацию, необходимую в процессе работы программы. Таким образом, процесс — это контейнер, в котором содержится вся информация, необходимая для работы программы.

Во многих операционных системах вся информация о каждом процессе, за исключением содержимого его собственного адресного пространства, хранится в таблице операционной системы, которая называется таблицей процессов и представляет собой массив (или связанный список) структур, по одной на каждый из существующих на данный момент процессов. Таким образом, процесс (в том числе приостановленный) состоит из собственного адресного пространства, которое обычно называют образом памяти, и записи в таблице процессов с содержимым его регистров, а также другой информацией, необходимой для последующего возобновления процесса.

Главными системными вызовами, используемыми при управлении процессами, являются вызовы, связанные с созданием и завершением процессов. Рассмотрим простой пример. Процесс, называемый интерпретатором команд, или оболочкой, считывает команды с терминала. Пользователь только что набрал команду, требующую компиляции программы. Теперь оболочка должна создать новый процесс, запускающий компилятор. Когда этот процесс завершит компиляцию, он произведет системный вызов для завершения собственного существования.

Если процесс способен создавать несколько других процессов (называющихся дочерними процессами), а эти процессы в свою очередь могут создавать собственные дочерние процессы, то перед нами предстает дерево процессов, подобное изображенному на рис. 6.

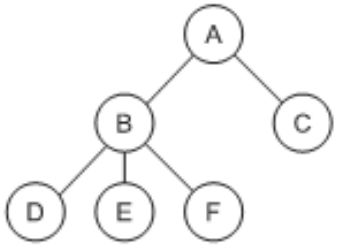

Рисунок 6. Дерево процессов. Процесс A создал два дочерних процесса, B и C. Процесс B создал три дочерних процесса, D, E и F

Связанные процессы, совместно работающие над выполнением какой-нибудь задачи, зачастую нуждаются в обмене данными друг с другом и синхронизации своих действий. Такая связь называется межпроцессным взаимодействием.

Каждому пользователю, которому разрешено работать с системой, системным администратором присваивается идентификатор пользователя (User IDentification (UID)).

Каждый запущенный процесс имеет UID того пользователя, который его запустил. Дочерние процессы имеют такой же UID, как и у родительского процесса. Пользователи могут входить в какую-нибудь группу, каждая из которых имеет собственный идентификатор группы (Group IDentification (GID)).

Адресные пространства. Каждый компьютер обладает определенным объемом оперативной памяти, используемой для хранения исполняемых программ. В самых простых операционных системах в памяти присутствует только одна программа. Для запуска второй программы сначала нужно удалить первую, а затем на ее место загрузить в память вторую.

Более изощренные операционные системы позволяют одновременно находиться в памяти нескольким программам. Чтобы исключить взаимные помехи (и помехи работе операционной

системы), нужен какой-то защитный механизм. Несмотря на то что этот механизм должен входить в состав оборудования, управляется он операционной системой.

При этом на многих компьютерах используется 32- или 64-разрядная адресация, позволяющая иметь адресное пространство размером 2^{32} или 2^{64} байт соответственно.

Что произойдет, если адресное пространство процесса превышает объем оперативной памяти, установленной на компьютере, а процессу требуется использовать все свое пространство целиком? На первых компьютерах такой процесс неизменно терпел крах.

В наше время, как уже упоминалось, существует технология виртуальной памяти, при которой операционная система хранит часть адресного пространства в оперативной памяти, а часть — на диске, по необходимости меняя их фрагменты местами. По сути, операционная система создает абстракцию адресного пространства в виде набора адресов, на которые может ссылаться процесс. Адресное пространство отделено от физической памяти машины и может быть как больше, так и меньше нее.

Файлы. Другим ключевым понятием, поддерживаемым практически всеми операционными системами, является файловая система. Как отмечалось ранее, основная функция операционной системы — скрыть специфику дисков и других устройств ввода-вывода и предоставить программисту удобную и понятную абстрактную модель, состоящую из независимых от устройств файлов. Вполне очевидно, что для создания, удаления, чтения и записи файлов понадобятся системные вызовы. Перед тем как файл будет готов к чтению, он должен быть найден на диске и открыт, а после считывания — закрыт. Для проведения этих операций предусмотрены системные вызовы.

Чтобы предоставить место для хранения файлов, многие операционные системы персональных компьютеров используют каталог как способ объединения файлов в группы.

Например, у студента может быть по одному каталогу для каждого изучаемого курса (для программ, необходимых в рамках данного курса), каталог для электронной почты и еще один — для своей домашней веб-страницы. Для создания и удаления каталогов нужны системные вызовы. Они также нужны для помещения в каталог существующего файла и удаления его оттуда. Элементами каталога могут быть либо файлы, либо другие каталоги. Эта модель стала прообразом иерархической структуры файловой системы, один из вариантов которой показан на рис. 7.

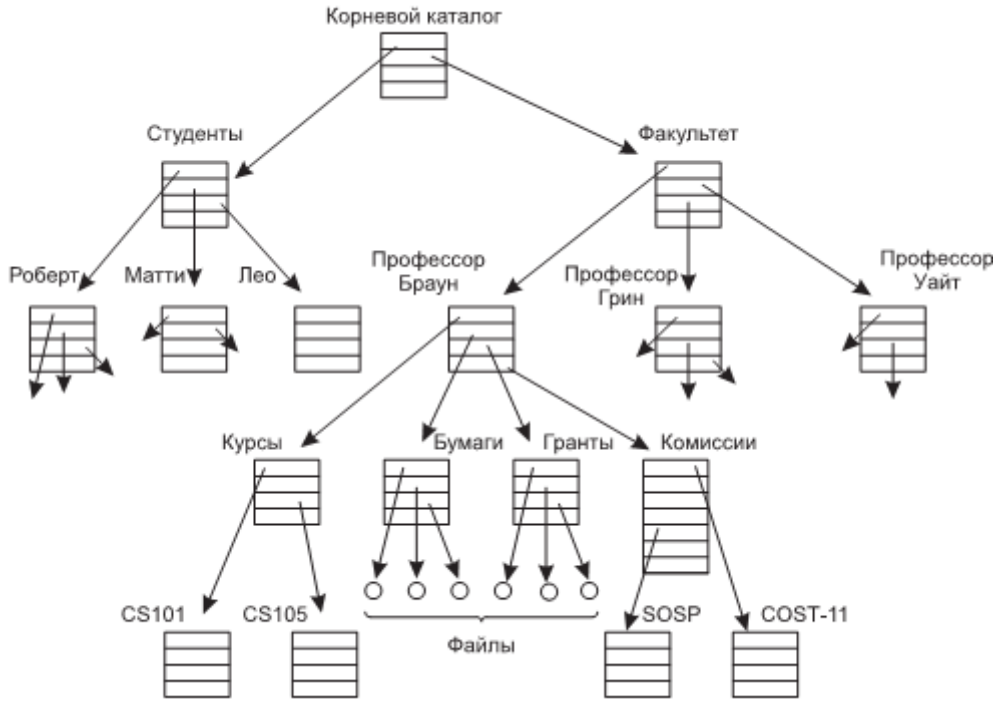

Рисунок 7. Файловая система факультета университета

Ввод-вывод данных. У всех компьютеров имеются физические устройства для получения входной и вывода выходной информации. Действительно, какой будет прок от компьютера, если пользователи не смогут поставить ему задачу и получить результаты по завершении заданной работы? Существует масса разнообразных устройств ввода-вывода: клавиатуры, мониторы, принтеры и т. д. Управление всеми этими устройствами возлагается на операционную систему.

Поэтому у каждой операционной системы для управления такими устройствами существует своя подсистема ввода-вывода. Некоторые программы ввода-вывода не зависят от конкретного устройства, то есть в равной мере подходят для применения со многими или со всеми устройствами ввода-вывода. Другая часть программ, например драйверы устройств, предназначена для определенных устройств ввода-вывода.

Безопасность. Компьютеры содержат большой объем информации, и часто пользователям нужно защитить ее и сохранить ее конфиденциальность. Возможно, это электронная почта, бизнес-планы, налоговые декларации и многое другое. Управление безопасностью системы также возлагается на операционную систему: например, она должна обеспечить доступ к файлам только пользователям, имеющим на это право.

В каждом поле есть бит, определяющий доступ для чтения, бит, определяющий доступ для записи, и бит, определяющий доступ для выполнения. Эти три бита называются rwx-битами (read, write, execute). Например, код защиты rwxr-x--x означает, что владельцу доступны чтение, запись или выполнение файла, остальным представителям его группы разрешается чтение или выполнение файла (но не запись), а всем остальным разрешено выполнение файла (но не чтение или запись). Для каталога x означает разрешение на поиск. Дефис (минус) означает, что соответствующее разрешение отсутствует.

Кроме защиты файлов существует множество других аспектов безопасности. Один из них — это защита системы от нежелательных вторжений как с участием, так и без участия людей (например, путем вирусных атак).

Оболочка. Операционная система представляет собой программу, выполняющую системные вызовы. Редакторы, компиляторы, ассемблеры, компоновщики, утилиты и интерпретаторы команд по определению не являются частью операционной системы при всей своей важности и приносимой пользе. Не являясь частью операционной системы, оболочка нашла широкое применение как средство доступа ко многим ее функциям и служит хорошим примером использования системных вызовов. Когда не применяется графический пользовательский интерфейс, она также является основным интерфейсом между пользователем, сидящим за своим терминалом, и операционной системой.

Оболочка запускается после входа в систему любого пользователя. В качестве стандартного устройства ввода и вывода оболочка использует терминал. Свою работу она начинает с вывода приглашения — знака доллара, сообщающего пользователю, что оболочка ожидает приема команды.

Раздел 2. Машинно-зависимые свойства операционных систем Тема 2.1. Архитектура операционной системы

Лекция 7. Структура операционной системы(монолитные системы). Виртуальные машины.

Монолитные системы. Несомненно, такая организация операционной системы является самой распространенной. Здесь вся операционная система работает как единая программа в режиме ядра. Операционная система написана в виде набора процедур, связанных вместе в одну большую исполняемую программу. При использовании этой технологии каждая

процедура может свободно вызвать любую другую процедуру, если та выполняет какое-нибудь полезное действие, в котором нуждается первая процедура. Возможность вызвать любую нужную процедуру приводит к весьма высокой эффективности работы системы, но наличие нескольких тысяч процедур, которые могут вызывать друг друга сколь угодно часто, нередко делает ее громоздкой и непонятной. Кроме того, отказ в любой из этих процедур приведет к аварии всей операционной системы.

Для построения исполняемого файла монолитной системы необходимо сначала скомпилировать все отдельные процедуры (или файлы, содержащие процедуры), а затем связать их вместе, воспользовавшись системным компоновщиком. Здесь, по существу, полностью отсутствует сокрытие деталей реализации — каждая процедура видна любой другой процедуре (в отличие от структуры, содержащей модули или пакеты, в которых основная часть информации скрыта внутри модулей и за пределами модуля его процедуры можно вызвать только через специально определяемые точки входа).

Такая организация предполагает следующую базовую структуру операционной системы:

1. Основная программа, которая вызывает требуемую служебную процедуру.

2. Набор служебных процедур, выполняющих системные вызовы.

3. Набор вспомогательных процедур, содействующих работе служебных процедур.

В этой модели для каждого системного вызова имеется одна ответственная за него служебная процедура, которая его и выполняет. Вспомогательные процедуры выполняют действия, необходимые нескольким служебным процедурам, в частности извлечение данных из пользовательских программ. Таким образом, процедуры делятся на три уровня (рис. 6).

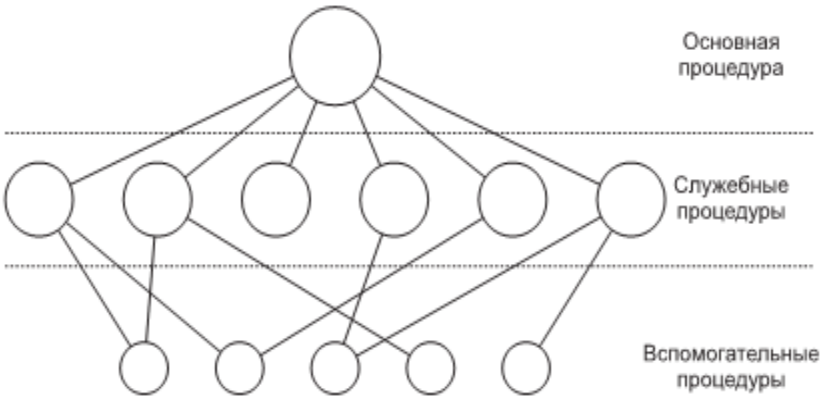

Рисунок 8 Простая структурированная модель монолитной системы

В дополнение к основной операционной системе, загружаемой во время запуска компьютера, многие операционные системы поддерживают загружаемые расширения, в числе которых драйверы устройств ввода-вывода и файловые системы. Эти компоненты загружаются по мере надобности. В UNIX они называются библиотеками общего пользования. В Windows они называются DLL-библиотеками (Dynamic-Link Libraries — динамически подключаемые библиотеки). Они находятся в файлах с расширениями имен .dll, и в каталоге C:\Windows\system32 на системе Windows их более 1000.

Виртуальные машины. Клиенты, арендовавшие виртуальную машину, могут запускать на ней какие угодно операционную систему и программное обеспечение, но за часть стоимости выделенного сервера (поскольку та же самая физическая машина одновременно поддерживает множество виртуальных машин).

Другой вариант использования виртуализации предназначен для конечных пользователей, которым необходима возможность одновременного запуска двух или более операционных систем, например Windows и Linux, поскольку некоторые из любимых ими приложений работают под управлением только одной из этих операционных систем. Такая ситуация показана на рис. 9, а, при этом термин «монитор виртуальной машины» заменен на «гипервизор первого типа» (type 1 hypervisor), который широко используется в наши дни из-за

краткости при наборе по сравнению с первым вариантом. Но для многих авторов они являются взаимозаменяемыми.

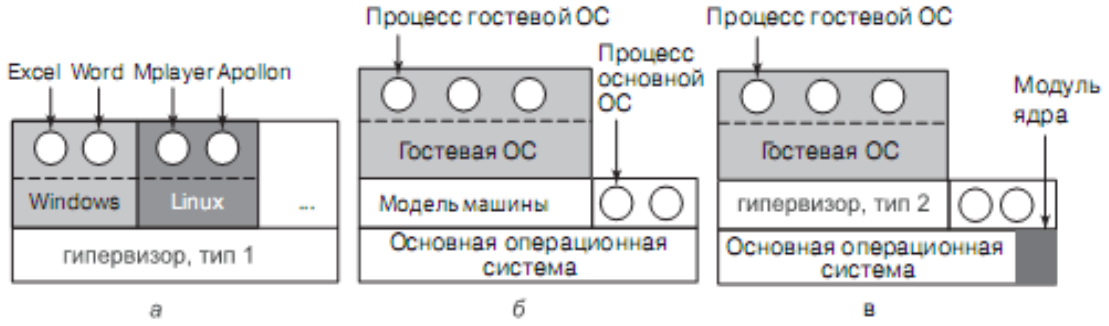

Рисунок 9 Гипервизор: а — тип 1; б — чистый гипервизор, тип 2; в — практический гипервизор, тип 2 (перевод)

Некоторые из этих ранних исследовательских проектов улучшили производительность по сравнению с интерпретаторами типа Bochs путем трансляции блоков кода на лету, сохранения их во внутреннем кэше и повторного использования результата трансляции в случае их нового исполнения. Это существенно повысило производительность и привело к созданию того, что сейчас называется моделями машин (machine simulators) (рис. 9, б). Но хотя эта технология, известная как двоичная трансляция (binary translation), помогла улучшить ситуацию, получившиеся системы, несмотря на то что они неплохо подходили для публикаций на академических конференциях, по-прежнему не отличались быстротой для использования в коммерческих средах, где производительность имеет весьма большое значение.

Следующим шагом в улучшении производительности стало добавление модуля ядра (рис. 9, в) для выполнения ряда трудоемких задач. Сложившаяся сейчас практика показывает, что все коммерчески доступные гипервизоры, такие как VMware Workstation, используют эту гибридную стратегию (а также имеют множество других усовершенствований).

На практике действительным различием между гипервизорами типа 1 и типа 2 является то, что в типе 2 для создания процессов, сохранения файлов и т. д. используется основная операционная система (host operating system) и ее файловая система. Гипервизор типа 1 не имеет основной поддержки и должен выполнять все эти функции самостоятельно.

После запуска гипервизор типа 2 считывает установочный компакт-диск (или файл образа компакт-диска) для выбора гостевой операционной системы (guest operation system) и установки гостевой ОС на виртуальный диск, который является просто большим файлом в файловой системе основной операционной системы. Гипервизор типа 1 этого делать не может по причине отсутствия основной операционной системы, в которой можно было бы хранить файлы.

Лекция 8. Микроядра. Клиент-серверная модель.

Микроядра. При использовании многоуровневого подхода разработчикам необходимо выбрать, где провести границу между режимами ядра и пользователя. Традиционно все уровни входили в ядро, но это было не обязательно. Существуют очень весомые аргументы в пользу того, чтобы в режиме ядра выполнялось как можно меньше процессов, поскольку ошибки в ядре могут вызвать немедленный сбой системы. Для сравнения: пользовательские процессы могут быть настроены на обладание меньшими полномочиями, чтобы их ошибки не носили фатального характера.

Различные исследователи неоднократно определяли количество ошибок на 1000 строк кода (например, Basilli and Perricone, 1984; Ostrand and Weyuker, 2002). Плотность ошибок зависит от размера модуля, его возраста и других факторов, но приблизительная цифра для солидных промышленных систем — 10 ошибок на 1000 строк кода.

Следовательно, монолитная операционная система, состоящая из 5 000 000 строк кода, скорее всего, содержит от 10 000 до 50 000 ошибок ядра. Разумеется, не все они имеют фатальный характер, некоторые ошибки могут представлять собой просто выдачу неправильного сообщения об ошибке в той ситуации, которая складывается крайне редко.

Тем, не менее операционные системы содержат столько ошибок, что производители компьютеров снабдили свою продукцию кнопкой перезапуска (которая зачастую находится на передней панели), чего не делают производители телевизоров, стереосистем и автомобилей, несмотря на большой объем программного обеспечения, имеющийся в этих устройствах.

Замысел, положенный в основу конструкции микроядра, направлен на достижение высокой надежности за счет разбиения операционной системы на небольшие, вполне определенные модули. Только один из них — микроядро — запускается в режиме ядра, а все остальные запускаются в виде относительно слабо наделенных полномочиями обычных пользовательских процессов. В частности, если запустить каждый драйвер устройства и файловую систему как отдельные пользовательские процессы, то ошибка в одном из них может вызвать отказ соответствующего компонента, но не сможет вызвать сбой всей системы. Таким образом, ошибка в драйвере звукового устройства приведет к искажению или пропаданию звука, но не вызовет зависания компьютера.

В отличие от этого в монолитной системе, где все драйверы находятся в ядре, некорректный драйвер звукового устройства может запросто сослаться на неверный адрес памяти и привести систему к немедленной вынужденной остановке.

Микроядро MINIX 3 занимает всего лишь около 12 000 строк кода на языке С и 1400 строк кода на ассемблере, который использован для самых низкоуровневых функций, в частности для перехвата прерываний и переключения процессов. Код на языке С занимается управлением процессами и их распределением, управляет межпроцессным взаимодействием (путем обмена сообщениями между процессами) и предлагает набор примерно из 40 вызовов ядра, позволяя работать остальной части операционной системы. Эти вызовы выполняют функции подключения обработчиков к прерываниям, перемещения данных между адресными пространствами и установки новых схем распределения памяти для только что созданных процессов. Структура процесса MINIX 3 показана на рис. 9, где обработчики вызовов ядра обозначены Sys. В ядре также размещен драйвер часов, потому что планировщик работает в тесном взаимодействии с ними. Все остальные драйверы устройств работают как отдельные пользовательские процессы.

За пределами ядра структура системы представляет собой три уровня процессов, которые работают в режиме пользователя. Самый нижний уровень содержит драйверы устройств. Поскольку они работают в пользовательском режиме, у них нет физического доступа к пространству портов ввода-вывода и они не могут вызывать команды ввода-вывода напрямую. Вместо этого, чтобы запрограммировать устройство ввода-вывода, драйвер создает структуру, сообщающую, какие значения в какие порты ввода-вывода следует записать. Затем драйвер осуществляет вызов ядра, сообщая ядру, что нужно произвести запись. При этом ядро может осуществить проверку, использует ли драйвер то устройство ввода-вывода, с которым он имеет право работать. Следовательно (в отличие от монолитной конструкции), дефектный драйвер звукового устройства не может случайно осуществить запись на диск.

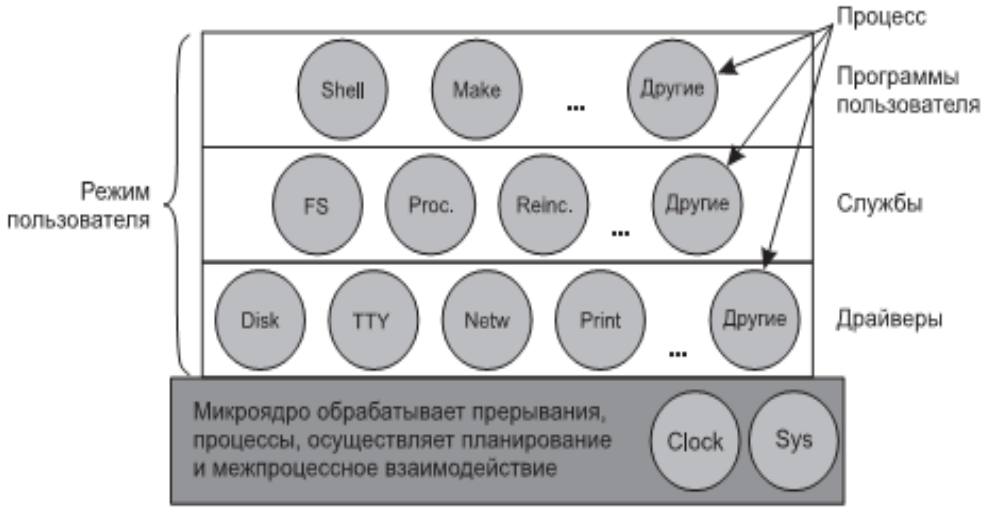

Рисунок 10. Упрощенная структура системы MINIX 3

Над драйверами расположен уровень, содержащий службы, которые осуществляют основной объем работы операционной системы. Все они работают в режиме пользователя.

Одна или более файловых служб управляют файловой системой (или системами), диспетчер процессов создает и уничтожает процессы, управляет ими и т. д. Пользовательские программы получают доступ к услугам операционной системы путем отправки коротких сообщений этим службам, которые запрашивают системные вызовы POSIX.

Например, процесс, нуждающийся в выполнении вызова read, отправляет сообщение одной из файловых служб, предписывая ей, что нужно прочитать.

Клиент-серверная модель. Небольшая вариация идеи микроядер выражается в обособлении двух классов процессов: серверов, каждый из которых предоставляет какую-нибудь службу, и клиентов, которые пользуются этими службами. Эта модель известна как клиент-серверная. Довольно часто самый нижний уровень представлен микроядром, но это не обязательно.

Суть заключается в наличии клиентских процессов и серверных процессов. Связь между клиентами и серверами часто организуется с помощью передачи сообщений. Чтобы воспользоваться службой, клиентский процесс составляет сообщение, в котором говорится, что именно ему нужно, и отправляет его соответствующей службе. Затем служба выполняет определенную работу и отправляет обратно ответ.

Если клиент и сервер запущены на одной и той же машине, то можно провести определенную оптимизацию, но концептуально здесь речь идет о передаче сообщений.

Очевидным развитием этой идеи будет запуск клиентов и серверов на разных компьютерах, соединенных локальной или глобальной сетью (рис. 1.23). Поскольку клиенты связываются с серверами путем отправки сообщений, им не обязательно знать, будут ли эти сообщения обработаны локально, на их собственных машинах, или же они будут отправлены по сети на серверы, расположенные на удаленных машинах.

Что касается интересов клиента, следует отметить, что в обоих случаях происходит одно и то же: отправляются запросы и возвращаются ответы. Таким образом, клиент-серверная модель является абстракцией, которая может быть использована как для отдельно взятой машины, так и для машин, объединенных в сеть.

Становится все больше и больше систем, привлекающих пользователей, сидящих за домашними компьютерами, в качестве клиентов, а большие машины, работающие где-нибудь в другом месте, — в качестве серверов. Фактически по этой схеме работает большая часть Интернета. Персональные компьютеры отправляют запросы на получение веб-страницы на

сервер, и эта веб-страница им возвращается. Это типичная картина использования клиент-серверной модели при работе в сети.

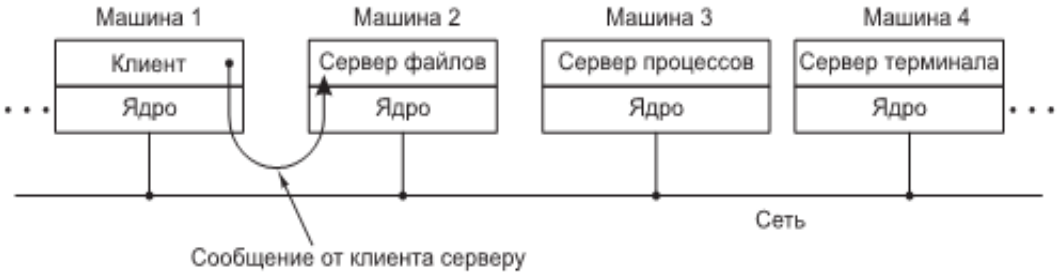

Рисунок 11 Клиент-серверная модель, реализованная с помощью сети

Раздел 3. Машинно-независимые свойства операционных систем Тема 3.1. Файловая система и ввод, и вывод информации Лекция 9. Файлы. Имена файлов. Типы файлов.

Имена файлов. Файл является механизмом абстрагирования. Он предоставляет способ сохранения информации на диске и последующего ее считывания, который должен оградить пользователя от подробностей о способе и месте хранения информации и деталей фактической работы дисковых устройств.

Наверное, наиболее важной характеристикой любого механизма абстрагирования является способ управления объектами и их именования, поэтому исследование файловой системы начнется с вопроса, касающегося имен файлов. Когда процесс создает файл, он присваивает ему имя. Когда процесс завершается, файл продолжает существовать, и к нему по этому имени могут обращаться другие процессы.

Некоторые файловые системы различают буквы верхнего и нижнего регистров, а некоторые не делают таких различий. Система UNIX подпадает под первую категорию, а старая MS-DOS — под вторую. (Кстати, при всей своей древности MS-DOS до сих пор довольно широко используется во встроенных системах, так что она отнюдь не устарела.) Поэтому система UNIX может рассматривать сочетания символов maria, Maria и MARIA как имена трех разных файлов. В MS-DOS все эти имена относятся к одному и тому же файлу.

Наверное, будет кстати следующее отступление, касающееся файловых систем. Обе операционные системы, Windows 95 и Windows 98, использовали файловую систему MS-DOS под названием FAT-16, и поэтому они унаследовали множество ее свойств, касающихся, например, построения имен файлов. В Windows 98 было представлено расширение FAT-16, которое привело к системе FAT-32, но обе эти системы очень похожи друг на друга. Вдобавок к этому Windows NT, Windows 2000, Windows XP, Windows Vista, Windows 7 и Windows 8 по-прежнему поддерживают обе файловые системы FAT, которые к настоящему времени фактически уже устарели. Но новые операционные системы имеют собственную намного более совершенную файловую систему NTFS, которая обладает несколько иными свойствами (к примеру, допускает имена файлов в кодировке Unicode).

Многие операционные системы поддерживают имена файлов, состоящие из двух частей, разделенных точкой, как, например, prog.c. Та часть имени, которая следует за точкой, называется расширением имени файла и, как правило, несет в себе некоторую информацию о файле. К примеру, в MS-DOS имена файлов состоят из 1–8 символов и имеют (необязательно) расширение, состоящее из 1–3 символов.

Некоторые широко распространенные расширения и их значения показаны на рис. 12.

Расширение	Значение
.bak	Резервная копия файла
.c	Исходный текст программы на языке C
.gif	Изображение формата GIF
.hlp	Файл справки
.html	Документ в формате HTML
.jpg	Статическое растровое изображение в формате JPEG
.mp3	Музыка в аудиоформате MPEG layer 3
.mpg	Фильм в формате MPEG
.o	Объектный файл (полученный на выходе компилятора, но еще не прошедший компоновку)
.pdf	Документ формата PDF
.ps	Документ формата PostScript
.tex	Входной файл для программы форматирования TEX
.txt	Обычный текстовый файл
.zip	Архив, сжатый программой zip

Рисунок 12. Некоторые типичные расширения имен файлов

Типы файлов. Многие операционные системы поддерживают несколько типов файлов. К примеру, в системах UNIX (опять же включая OS X) и Windows имеются обычные файлы и каталоги. В системе UNIX имеются также символьные и блочные специальные файлы.

Обычными считаются файлы, содержащие информацию пользователя. Все файлы на рис. 13 являются обычными. Каталоги — это системные файлы, предназначенные для поддержки структуры файловой системы. Мы рассмотрим их чуть позже. Символьные специальные файлы имеют отношение к вводу-выводу и используются для моделирования последовательных устройств ввода-вывода, к которым относятся терминалы, принтеры и сети. Блочные специальные файлы используются для моделирования дисков. В данной главе нас в первую очередь будут интересовать обычные файлы.

Файлы могут быть структурированы несколькими различными способами. Три наиболее вероятные структуры показаны на рис. 13. Файл на рис. 13, а представляет собой бессистемную последовательность байтов. В сущности, операционной системе все равно, что содержится в этом файле, — она видит только байты. Какое-либо значение этим байтам придают программы на уровне пользователя. Такой подход используется как в UNIX, так и в Windows.

Первый шаг навстречу некой структуре показан на рис. 13, б. В данной модели файл представляет собой последовательность записей фиксированной длины, каждая из которых имеет собственную внутреннюю структуру. Основная идея файла как последовательности записей состоит в том, что операция чтения возвращает одну из записей, а операция записи перезаписывает или дополняет одну из записей. В качестве исторического отступления заметим, что несколько десятилетий назад, когда в компьютерном мире властвовали перфокарты на 80 столбцов, многие операционные системы универсальных машин в основе своей файловой системы использовали файлы, состоящие из 80-символьных записей, — в сущности, образы перфокарт.

Третья разновидность структуры файла показана на рис. 13, в. При такой организации файл состоит из дерева записей, необязательно одинаковой длины, каждая из которых в конкретной позиции содержит ключевое поле. Дерево сортируется по ключевому полю, позволяя выполнять ускоренный поиск по конкретному ключу.

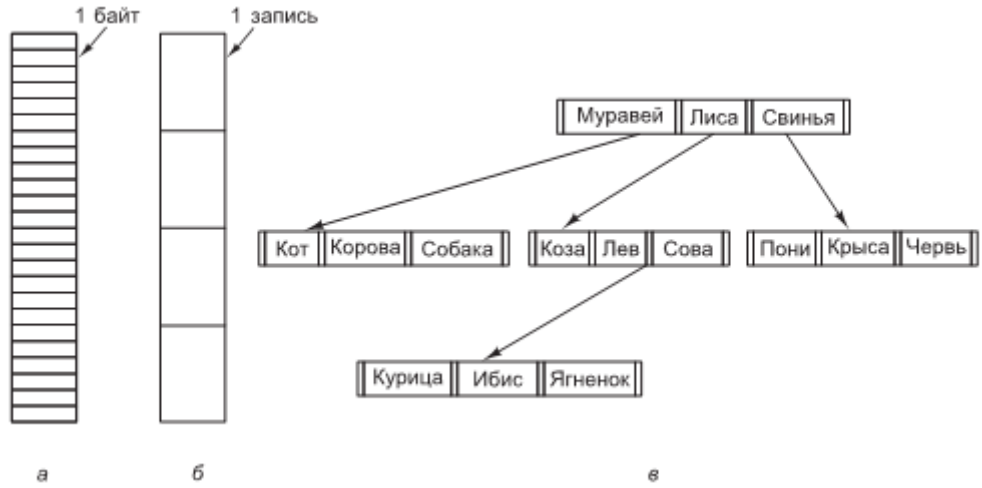

Рисунок 13. Три типа файлов: а — последовательность байтов; б — последовательность записей; в — дерево

Лекция 10. Операции с файлами. Операции с каталогами.

Файлы предназначены для хранения информации с возможностью ее последующего извлечения. Разные системы предоставляют различные операции, позволяющие сохранять и извлекать информацию. Далее рассматриваются наиболее распространенные системные вызовы, относящиеся к работе с файлами.

\ Create (Создать). Создает файл без данных. Цель вызова состоит в объявлении о появлении нового файла и установке ряда атрибутов.

\ Delete (Удалить). Когда файл больше не нужен, его нужно удалить, чтобы освободить дисковое пространство. Именно для этого и предназначен этот системный вызов.

\ Open (Открыть). Перед использованием файла процесс должен его открыть. Цель системного вызова open — дать возможность системе извлечь и поместить в оперативную память атрибуты и перечень адресов на диске, чтобы ускорить доступ к ним при последующих вызовах.

\ Close (Закрыть). После завершения всех обращений к файлу потребность в его атрибутах и адресах на диске уже отпадает, поэтому файл должен быть закрыт, чтобы освободить место во внутренней таблице. Многие системы устанавливают максимальное количество открытых процессами файлов, определяя смысл существования этого вызова. Информация на диск пишется блоками, и закрытие файла вынуждает к записи последнего блока файла, даже если этот блок и не заполнен.

\ Read (Произвести чтение). Считывание данных из файла. Как правило, байты поступают с текущей позиции. Вызывающий процесс должен указать объем необходимых данных и предоставить буфер для их размещения.

\ Write (Произвести запись). Запись данных в файл, как правило, с текущей позиции. Если эта позиция находится в конце файла, то его размер увеличивается. Если текущая позиция находится где-то в середине файла, то новые данные пишутся поверх существующих, которые утрачиваются навсегда.

\ Append (Добавить). Этот вызов является усеченной формой системного вызова write. Он может лишь добавить данные в конец файла. Как правило, у систем, предоставляющих минимальный набор системных вызовов, вызов append отсутствует, но многие системы предоставляют множество способов получения того же результата, и иногда в этих системах присутствует вызов append.

⑊ Seek (Найти). При работе с файлами произвольного доступа нужен способ указания места, с которого берутся данные. Одним из общепринятых подходов является применение системного вызова seek, который перемещает указатель файла к определенной позиции в файле. После завершения этого вызова данные могут считываться или записываться с этой позиции.

⑊ Get attributes (Получить атрибуты). Процессу для работы зачастую необходимо считать атрибуты файла. К примеру, имеющаяся в UNIX программа make обычно используется для управления проектами разработки программного обеспечения, состоящими из множества сходных файлов. При вызове программа make проверяет время внесения последних изменений всех исходных и объектных файлов и для обновления проекта обходится компиляцией лишь минимально необходимого количества файлов. Для этого ей необходимо просмотреть атрибуты файлов, а именно время внесения последних изменений.

⑊ Set attributes (Установить атрибуты). Значения некоторых атрибутов могут устанавливаться пользователем и изменяться после того, как файл был создан. Такую возможность дает именно этот системный вызов. Характерным примером может послужить информация о режиме защиты. Под эту же категорию подпадает большинство флагов.

⑊ Rename (Переименовать). Нередко пользователю требуется изменить имя существующего файла. Этот системный вызов помогает решить эту задачу. Необходимость в нем возникает не всегда, поскольку файл может быть просто скопирован в новый файл с новым именем, а старый файл затем может быть удален.

Операции с каталогами. Допустимые системные вызовы для управления каталогами имеют большее количество вариантов от системы к системе, чем системные вызовы, управляющие файлами. Рассмотрим примеры, дающие представление об этих системных вызовах и характере их работы (взяты из системы UNIX).

⑊ Create (Создать каталог). Каталог создается пустым, за исключением точки и двойной точки, которые система помещает в него автоматически (или в некоторых случаях при помощи программы mkdir).

⑊ Delete (Удалить каталог). Удалить можно только пустой каталог. Каталог, содержащий только точку и двойную точку, рассматривается как пустой, поскольку они не могут быть удалены.

⑊ Opendir (Открыть каталог). Каталоги могут быть прочитаны. К примеру, для вывода имен всех файлов, содержащихся в каталоге, программа ls открывает каталог для чтения имен всех содержащихся в нем файлов. Перед тем как каталог может быть прочитан, он должен быть открыт по аналогии с открытием и чтением файла.

⑊ Closedir (Закрыть каталог). Когда каталог прочитан, он должен быть закрыт, чтобы освободить пространство во внутренних таблицах системы.

⑊ Readdir (Прочитать каталог). Этот вызов возвращает следующую запись из открытого каталога. Раньше каталоги можно было читать с помощью обычного системного вызова read, но недостаток такого подхода заключался в том, что программист вынужден был работать с внутренней структурой каталогов, о которой он должен был знать заранее. В отличие от этого, readdir всегда возвращает одну запись в стандартном формате независимо от того, какая из возможных структур каталогов используется.

⑊ Rename (Переименовать каталог). Во многих отношениях каталоги подобны файлам и могут быть переименованы точно так же, как и файлы.

⑊ Link (Привязать). Привязка представляет собой технологию, позволяющую файлу появляться более чем в одном каталоге. В этом системном вызове указываются существующий файл и новое имя файла в некотором существующем каталоге и создается привязка существующего файла к указанному каталогу с указанным новым именем. Таким образом, один и тот же файл может появиться в нескольких каталогах, возможно, под разными именами. Подобная привязка, увеличивающая показания файлового счетчика i-узла (предназначенного для отслеживания количества записей каталогов, в которых фигурирует файл), иногда называется жесткой связью, или жесткой ссылкой (hard link).

\ Unlink (Отвязать). Удалить запись каталога. Если отвязываемый файл присутствует только в одном каталоге (что чаще всего и бывает), то этот вызов удалит его из файловой системы. Если он фигурирует в нескольких каталогах, то он будет удален из каталога, который указан в имени файла. Все остальные записи останутся.

Фактически системным вызовом для удаления файлов в UNIX (как ранее уже было рассмотрено) является unlink.

Лекция 11. Структура файловой системы. Реализация файлов.

Структура файловой системы. Файловые системы хранятся на дисках. Большинство дисков может быть разбито на один или несколько разделов, на каждом из которых будет независимая файловая система. Сектор 0 на диске называется главной загрузочной записью (Master Boot Record (MBR)) и используется для загрузки компьютера. В конце MBR содержится таблица разделов. Из этой таблицы берутся начальные и конечные адреса каждого раздела. Один из разделов в этой таблице помечается как активный. При загрузке компьютера BIOS (базовая система ввода-вывода) считывает и выполняет MBR. Первое, что делает программа MBR, — находит расположение активного раздела, считывает его первый блок, который называется загрузочным, и выполняет его. Программа в загрузочном блоке загружает операционную систему, содержащуюся в этом разделе. Для достижения единообразия каждый раздел начинается с загрузочного блока, даже если он не содержит загружаемой операционной системы. Кроме того, в будущем он может содержать какую-нибудь операционную систему.

Во всем остальном, кроме того, что раздел начинается с загрузочного блока, строение дискового раздела значительно различается от системы к системе. Зачастую файловая система будет содержать некоторые элементы, показанные на рис. 14. Первым элементом является суперблок. В нем содержатся все ключевые параметры файловой системы, которые считываются в память при загрузке компьютера или при первом обращении к файловой системе. Обычно в информацию суперблока включаются «магическое» число, позволяющее идентифицировать тип файловой системы, количество блоков в файловой системе, а также другая важная административная информация.

Далее может находиться информация о свободных блоках файловой системы, к примеру, в виде битового массива или списка указателей. За ней могут следовать i-узлы, массив структур данных — на каждый файл по одной структуре, в которой содержится вся информация о файле. Затем может размещаться корневой каталог, содержащий вершину дерева файловой системы. И наконец, оставшаяся часть диска содержит все остальные каталоги и файлы.

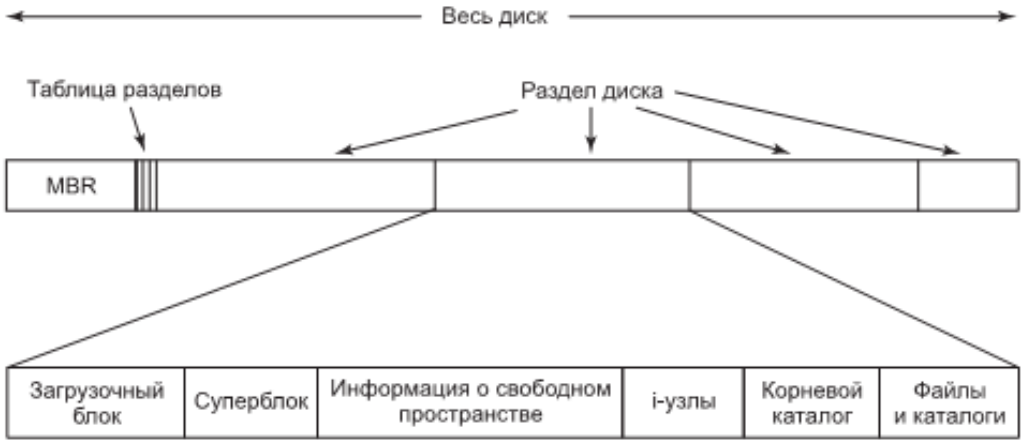

Рисунок 14 Возможная структура файловой системы

Непрерывное размещение. Простейшая схема размещения заключается в хранении каждого файла на диске в виде непрерывной последовательности блоков. Таким образом, на диске с

блоками, имеющими размер 1 Кбайт, файл размером 50 Кбайт займет 50 последовательных блоков. При блоках, имеющих размер 2 Кбайт, под него будет выделено 25 последовательных блоков.

Пример хранилища с непрерывным размещением приведен на рис. 15, а. На нем показаны 40 первых блоков, начинающихся с блока 0 слева. Изначально диск был пустым. Затем на него начиная с блока 0 был записан файл A длиной четыре блока. Затем правее окончания файла A записан файл B, занимающий шесть блоков. Следует заметить, что каждый файл начинается от границы нового блока, поэтому, если файл A фактически имел длину 3,5 блока, то в конце последнего блока часть пространства будет потеряна впустую. Всего на рисунке показаны семь файлов, каждый из которых начинается с блока, который следует за последним блоком предыдущего файла. Затенение использовано только для того, чтобы упростить показ деления пространства на файлы. В отношении самого хранилища оно не имеет никакого практического значения.

У непрерывного распределения дискового пространства есть два существенных преимущества. Во-первых, его просто реализовать, поскольку отслеживание местонахождения принадлежащих файлу блоков сводится всего лишь к запоминанию двух чисел: дискового адреса первого блока и количества блоков в файле. При наличии номера первого блока номер любого другого блока может быть вычислен путем простого сложения.

Во-вторых, у него превосходная производительность считывания, поскольку весь файл может быть считан с диска за одну операцию. Для нее потребуется только одна операция позиционирования (на первый блок). После этого никаких позиционирований или ожиданий подхода нужного сектора диска уже не потребуется, поэтому данные поступают на скорости, равной максимальной пропускной способности диска. Таким образом, непрерывное размещение характеризуется простотой реализации и высокой производительностью.

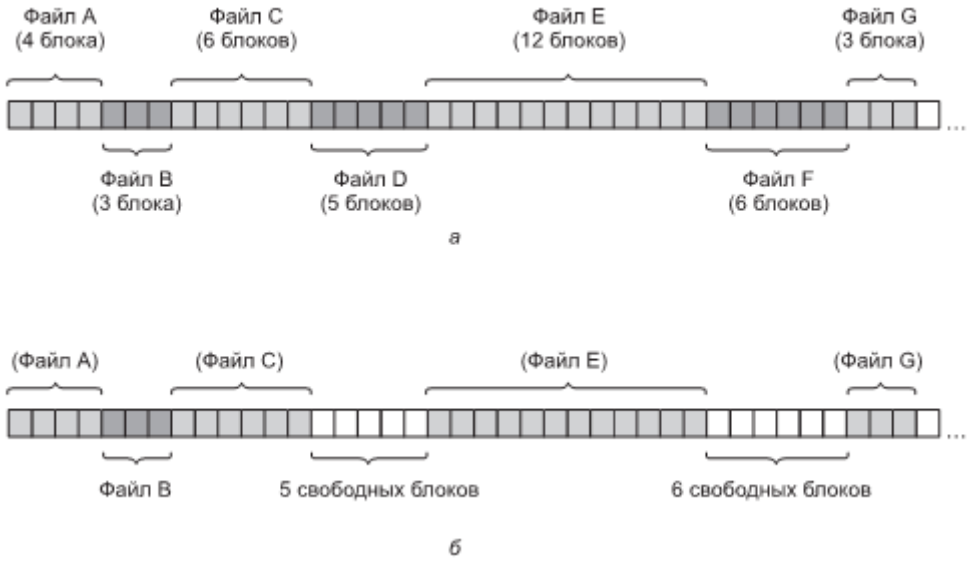

Рисунок 15 Дисковое пространство: а — непрерывное размещение семи файлов; б — состояние диска после удаления файлов D и F

К сожалению, у непрерывного размещения есть также очень серьезный недостаток: со временем диск становится фрагментированным. Как это происходит, показано на рис. 15, б. Были удалены два файла — D и F. Естественно, при удалении файла его блоки освобождаются и на диске остается последовательность свободных блоков.

Немедленное уплотнение файлов на диске для устранения такой последовательности свободных блоков («дыры») не осуществляется, поскольку для этого потребуется скопировать все блоки, — а их могут быть миллионы, — следующие за ней, что при использовании больших дисков займет несколько часов или даже дней. В результате, как показано на рис. 15, б, диск содержит вперемешку файлы и последовательности свободных блоков.

Лекция 12. Резервное копирование файловой системы. Дефрагментация дисков.

Резервное копирование файловой системы. Выход из строя файловой системы зачастую оказывается куда более серьезной неприятностью, чем поломка компьютера. Если компьютер ломается из-за пожара, удара молнии или чашки кофе, пролитой на клавиатуру, это, конечно же, неприятно и ведет к непредвиденным расходам, но обычно дело обходится покупкой новых комплектующих и не причиняет слишком много хлопот.

Если же на компьютере по аппаратной или программной причине будет безвозвратно утрачена его файловая система, восстановить всю информацию будет делом нелегким, небыстрым и во многих случаях просто невозможным. Для людей, чьи программы, документы, налоговые записи, файлы клиентов, базы данных, планы маркетинга или другие данные утрачены навсегда, последствия могут быть катастрофическими. Хотя файловая система не может предоставить какую-либо защиту против физического разрушения оборудования или носителя, она может помочь защитить информацию. Решение простое: нужно делать резервные копии.

Резервное копирование на ленту производится обычно для того, чтобы справиться с двумя потенциальными проблемами:

- восстановлением после аварии;
- восстановлением после необдуманных действий (ошибок пользователей).

Первая из этих проблем заключается в возвращении компьютера в строй после поломки диска, пожара, наводнения или другого стихийного бедствия. На практике такое случается нечасто, поэтому многие люди даже не задумываются о резервном копировании. Эти люди по тем же причинам также не склонны страховать свои дома от пожара.

Вторая причина восстановления вызвана тем, что пользователи часто случайно удаляют файлы, потребность в которых в скором времени возникает опять. Эта происходит так часто, что когда файл «удаляется» в Windows, он не удаляется навсегда, а просто перемещается в специальный каталог — Корзину, где позже его можно отловить и легко восстановить. Резервное копирование делает этот принцип более совершенным и дает возможность файлам, удаленным несколько дней, а то и недель назад, восстанавливаться со старых лент резервного копирования.

Резервное копирование занимает много времени и пространства, поэтому эффективность и удобство играют в нем большую роль. В связи с этим возникают следующие вопросы. Во-первых, нужно ли проводить резервное копирование всей файловой системы или только какой-нибудь ее части? Во многих эксплуатирующихся компьютерных системах исполняемые (двоичные) программы содержатся в ограниченной части дерева файловой системы. Если все они могут быть переустановлены с веб-сайта производителя или установочного DVD-диска, то создавать их резервные копии нет необходимости. Также у большинства систем есть каталоги для хранения временных файлов. Обычно включать их в резервную копию также нет смысла.

Во-вторых, бессмысленно делать резервные копии файлов, которые не изменились со времени предыдущего резервного копирования, что наталкивает на мысль об инкрементном резервном копировании. Простейшей формой данного метода будет периодическое создание полной резервной копии, скажем, еженедельное или ежемесячное, и ежедневное резервное копирование только тех файлов, которые были изменены со времени последнего полного резервного копирования. Еще лучше создавать резервные копии только тех файлов, которые изменились со времени их последнего резервного копирования.

В-третьих, поскольку обычно резервному копированию подвергается огромный объем данных, может появиться желание сжать их перед записью на ленту. Но у многих алгоритмов сжатия одна сбойная область на ленте может нарушить работу алгоритма распаковки и сделать нечитаемым весь файл или даже всю ленту.

В-четвертых, резервное копирование активной файловой системы существенно затруднено. Если в процессе резервного копирования происходит добавление, удаление и изменение файлов и каталогов, можно получить весьма противоречивый результат.

Но поскольку архивация данных может занять несколько часов, то для выполнения резервного копирования может потребоваться перевести систему в автономный режим на большую часть ночного времени, что не всегда приемлемо. Поэтому были разработаны алгоритмы для создания быстрых копий текущего состояния файловой системы за счет копирования критических структур данных, которые при последующем изменении файлов и каталогов требуют копирования блоков вместо обновления их на месте (Hutchinson et al., 1999).

Для резервного копирования диска можно воспользоваться одной из двух стратегий: физической или логической архивацией. Физическая архивация ведется с нулевого блока диска, при этом все дисковые блоки записываются на лету в порядке их следования и, когда скопирован последний блок, запись останавливается. Эта программа настолько проста, что, возможно, она может быть избавлена от ошибок на все 100 %, чего, наверное, нельзя сказать о любых других полезных программах.

Логическая архивация начинается в одном или нескольких указанных каталогах и рекурсивно архивирует все найденные там файлы и каталоги, в которых произошли изменения со времени какой-нибудь заданной исходной даты (например, даты создания резервной копии при инкрементном архивировании или даты установки системы для полного архивирования). Таким образом, при логической архивации на магнитную ленту записываются последовательности четко идентифицируемых каталогов и файлов, что упрощает восстановление по запросу указанного файла или каталога.

Поскольку наибольшее распространение получила логическая архивация, рассмотрим подробности ее общего алгоритма на основе примера (рис.16). Этот алгоритм используется в большинстве UNIX-систем. На рисунке показана файловая система с каталогами (квадраты) и файлами (окружности). Закрашены те элементы, которые подверглись изменениям со времени исходной даты и поэтому подлежат архивированию. Светлые элементы в архивации не нуждаются.

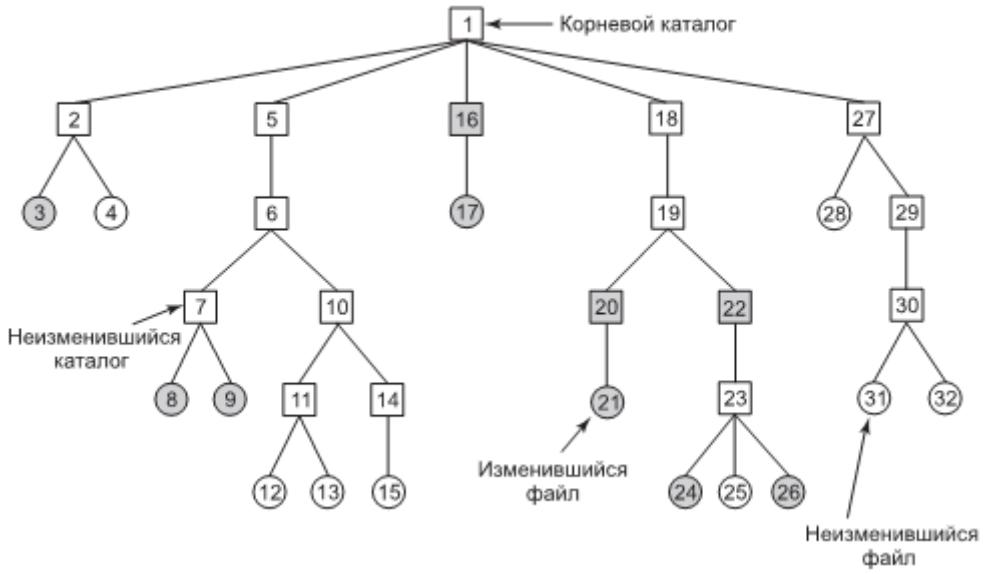

Рисунок 16 Архивируемая файловая система. Квадратами обозначены каталоги, а окружностями — файлы. Закрашены те элементы, которые были изменены со времени последнего архивирования. Каждый каталог и файл помечен номером своего i-узла

Дефрагментация дисков. При начальной установке операционной системы все нужные ей программы и файлы устанавливаются последовательно с самого начала диска, каждый новый каталог следует за предыдущим. За установленными файлами единым непрерывным

участком следует свободное пространство. Но со временем по мере создания и удаления файлов диск обычно приобретает нежелательную фрагментацию, где повсеместно встречаются файлы и области свободного пространства. Вследствие этого при создании нового файла используемые им блоки могут быть разбросаны по всему диску, что ухудшает производительность.

Производительность можно восстановить за счет перемещения файлов с места на место, чтобы они размещались непрерывно, и объединения всего (или, по крайней мере, основной части) свободного дискового пространства в один или в несколько непрерывных участков на диске. В системе Windows имеется программа defrag, которая именно этим и занимается. Пользователи Windows должны регулярно запускать эту программу, делая исключение для SSD-накопителей.

Дефрагментация проводится успешнее на тех файловых системах, которые располагают большим количеством свободного пространства в непрерывном участке в конце раздела. Это пространство позволяет программе дефрагментации выбрать фрагментированные файлы ближе к началу раздела и скопировать их блоки в свободное пространство. В результате этого освободится непрерывный участок ближе к началу раздела, в который могут быть целиком помещены исходные или какие-нибудь другие файлы. Затем процесс может быть повторен для следующего участка дискового пространства и т. д.

Некоторые файлы не могут быть перемещены; к ним относятся файлы, используемые в страничной организации памяти и реализации спящего режима, а также файлы журналирования, поскольку выигрыш от этого не оправдывает затрат, необходимых на администрирование. В некоторых системах такие файлы все равно занимают непрерывные участки фиксированного размера и не нуждаются в дефрагментации. Один из случаев, когда недостаток их подвижности становится проблемой, связан с их размещением близко к концу раздела и желанием пользователя сократить размер этого раздела. Единственный способ решения этой проблемы состоит в их полном удалении, изменении размера раздела, а затем их повторном создании.

Файловые системы Linux (особенно ext2 и ext3) обычно меньше страдают от дефрагментации, чем системы, используемые в Windows, благодаря способу выбора дисковых блоков, поэтому принудительная дефрагментация требуется довольно редко. Кроме того, SSD-накопители вообще не страдают от фрагментации. Фактически дефрагментация SSD-накопителя является контрпродуктивной. Она не только не дает никакого выигрыша в производительности, но и приводит к износу, сокращая время их жизни.

Тема 3.2. Управление памятью Лекция 13. Абстракция памяти.

Память представляет собой очень важный ресурс, требующий четкого управления.

Со временем была разработана концепция иерархии памяти, согласно которой компьютеры обладают несколькими мегабайтами очень быстродействующей, дорогой и энергозависимой кэш-памяти, несколькими гигабайтами памяти, средней как по скорости, так и по цене, а также несколькими терабайтами памяти на довольно медленных, сравнительно дешевых дисковых накопителях, не говоря уже о сменных накопителях, таких как DVD и флеш-устройства USB. Превратить эту иерархию в абстракцию, то есть в удобную модель, а затем управлять этой абстракцией — и есть задача операционной системы.

Та часть операционной системы, которая управляет иерархией памяти (или ее частью), называется менеджером, или диспетчером, памяти. Он предназначен для действенного управления памятью и должен следить за тем, какие части памяти используются, выделять память процессам, которые в ней нуждаются, и освобождать память, когда процессы завершат свою работу.

Даже в условиях, когда в качестве модели памяти выступает сама физическая память, возможны несколько вариантов использования памяти. Три из них показаны на рис. 17. Операционная система может (рис. 17, а) размещаться в нижней части адресов, в оперативном запоминающем устройстве (ОЗУ), или, по-другому, в памяти с произвольным доступом — RAM (Random Access Memory). Она может размещаться также в постоянном

запоминающем устройстве (ПЗУ), или, иначе, в ROM (Read-Only Memory), в верхних адресах памяти (рис. 17, б). Или же драйверы устройств могут быть в верхних адресах памяти, в ПЗУ, а остальная часть системы — в ОЗУ, в самом низу (рис. 17, в). Первая модель прежде использовалась на универсальных машинах и мини-компьютерах, а на других машинах — довольно редко. Вторая модель использовалась на некоторых КПК и встроенных системах. Третья модель использовалась на ранних персональных компьютерах (например, на тех, которые работали под управлением MS-DOS), где часть системы, размещавшаяся в ПЗУ, называлась базовой системой ввода-вывода — BIOS (Basic Input Output System). Недостаток моделей, изображенных на рис. 17, а и в, заключается в том, что ошибка в программе пользователя может затереть операционную систему, и, возможно, с весьма пагубными последствиями.

Рисунок 17 Три простых способа организации памяти при наличии операционной системы и одного пользовательского процесса (существуют и другие варианты). При таком устройстве системы памяти процессы, как правило, могут запускаться только по одному. Как только пользователь наберет команду, операционная система копирует запрошенную программу с диска в память, после чего выполняет эту программу. Когда процесс завершает свою работу, операционная система отображает символ приглашения и ожидает ввода новой команды. По получении команды она загружает в память новую программу, записывая ее поверх первой программы

Рассмотрим для памяти новую абстракцию: адресное пространство. Так же как понятие процесса создает своеобразный абстрактный центральный процессор для запуска программ, понятие адресного пространства создает своеобразную абстрактную память, в которой существуют программы. Адресное пространство — это набор адресов, который может быть использован процессом для обращения к памяти. У каждого процесса имеется собственное адресное пространство, независимое от того адресного пространства, которое принадлежит другим процессам (за исключением тех особых обстоятельств, при которых процессам требуется совместное использование их адресных пространств).

С годами для преодоления перегрузки памяти были выработаны два основных подхода. Самый простой из них, называемый свопингом, заключается в размещении в памяти всего процесса целиком, его запуске на некоторое время, а затем сбросе на диск. Бездействующие процессы большую часть времени хранятся на диске и в нерабочем состоянии не занимают пространство оперативной памяти (хотя некоторые из них периодически активизируются, чтобы проделать свою работу, после чего опять приостанавливаются). Второй подход называется виртуальной памятью, он позволяет программам запускаться даже в том случае, если они находятся в оперативной памяти лишь частично.

Работа системы с использованием свопинга показана на рис. 18. Изначально в памяти присутствует только процесс A. Затем создаются или появляются в памяти путем свопинга с диска процессы B и C. На рис. 18, г процесс A за счет свопинга выгружается на диск. Затем появляется процесс D и выгружается из памяти процесс B. И наконец, снова появляется в памяти процесс A. Поскольку теперь процесс A находится в другом месте, содержащиеся в нем адреса должны быть перестроены либо программным путем, при загрузке в процессе свопинга, либо (скорее всего) аппаратным путем в процессе выполнения программы. К

примеру, для этого случая хорошо подойдут механизмы базового и ограничительного регистров.

Когда в результате свопинга в памяти создаются несколько свободных областей, их можно объединить в одну большую за счет перемещения при первой же возможности всех процессов в нижние адреса. Эта технология известна как уплотнение памяти. Но зачастую она не выполняется, поскольку отнимает довольно много процессорного времени. К примеру, на машине, оснащенной 16 Гбайт памяти, способной скопировать 8 байт за 8 нс, на уплотнение всего объема памяти может уйти около 16 с.

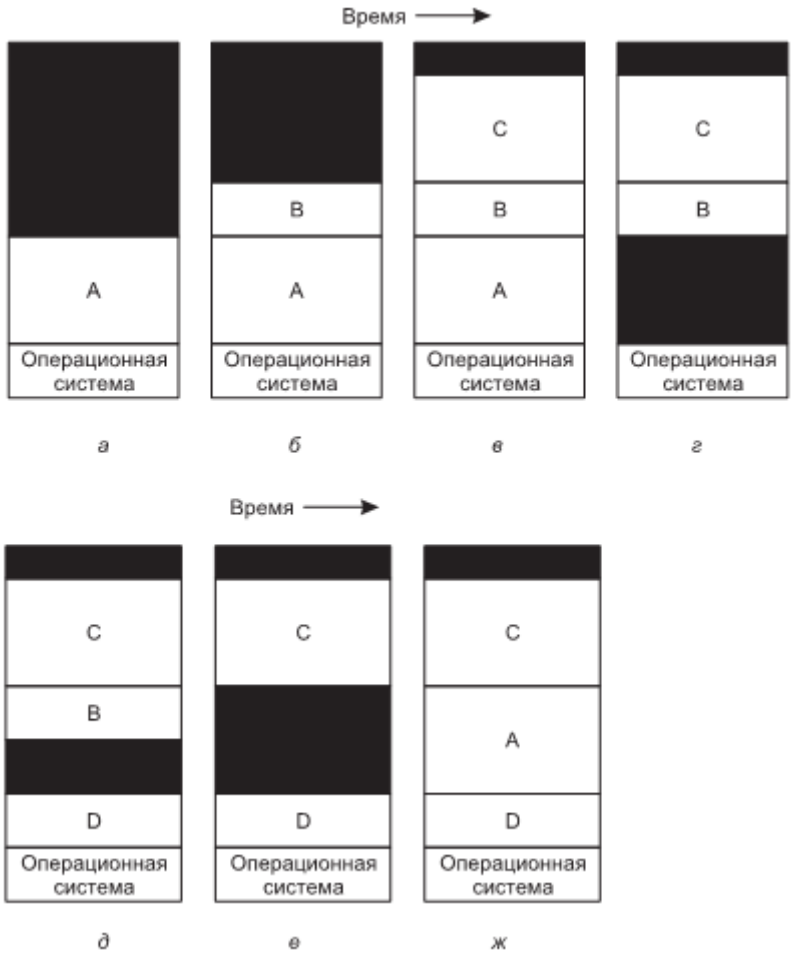

Рисунок 18 Изменения в выделении памяти по мере появления процессов в памяти и выгрузки их из нее (неиспользованные области памяти заштрихованы)

Стоит побеспокоиться о том, какой объем памяти нужно выделить процессу при его создании или загрузке в процессе свопинга. Если создаваемый процесс имеет вполне определенный неизменный объем, то выделение упрощается: операционная система предоставляет процессу строго необходимый объем памяти, ни больше, ни меньше.

Лекция 14. Виртуальная память.

Свопинг — не слишком привлекательный выбор, поскольку обычный диск с интерфейсом SATA обладает пиковой скоростью передачи данных в несколько сотен мегабайт в секунду, а это означает, что свопинг программы объемом 1 Гбайт займет секунды, и еще столько же времени будет потрачено на загрузку другой программы в 1 Гбайт.

Проблемы программ, превышающих по объему размер имеющейся памяти, возникли еще на заре компьютерной эры, правда, проявились они в таких узких областях, как решение научных и прикладных задач (существенные объемы памяти требуются для моделирования возникновения Вселенной или даже для авиасимулятора нового самолета). В 1960-е годы было принято решение разбивать программы на небольшие части, называемые оверлеями. При запуске программы в память загружался только администратор оверлейной загрузки, который тут же загружал и запускал оверлей с порядковым номером 0. Когда этот оверлей завершал свою работу, он мог сообщить администратору загрузки оверлеев о необходимости загрузки оверлея 1 либо выше оверлея 0, находящегося в памяти (если для него было достаточно пространства), либо поверх оверлея 0 (если памяти не хватало). Некоторые оверлейные системы имели довольно сложное устройство, позволяя множеству оверлеев одновременно находиться

в памяти. Оверлеи хранились на диске, и их свопинг с диска в память и обратно осуществлялся администратором загрузки оверлеев.

Хотя сама работа по свопингу оверлеев с диска в память и обратно выполнялась операционной системой, разбиение программ на части выполнялось программистом в ручном режиме. Разбиение больших программ на небольшие модульные части было очень трудоемкой, скучной и не застрахованной от ошибок работой. Преуспеть в этом деле удавалось далеко не всем программистам. Прошло не так много времени, и был придуман способ, позволяющий возложить эту работу на компьютер.

Изобретенный метод (Fotheringham, 1961) стал известен как виртуальная память. В основе виртуальной памяти лежит идея, что у каждой программы имеется собственное адресное пространство, которое разбивается на участки, называемые страницами. Каждая страница представляет собой непрерывный диапазон адресов. Эти страницы отображаются на физическую память, но для запуска программы одновременное присутствие в памяти всех страниц необязательно. Когда программа ссылается на часть своего адресного пространства, находящегося в физической памяти, аппаратное обеспечение осуществляет необходимое отображение на лету. Когда программа ссылается на часть своего адресного пространства, которое не находится в физической памяти, операционная система предупреждается о том, что необходимо получить недостающую часть и повторно выполнить потерпевшую неудачу команду.

Виртуальная память неплохо работает и в многозадачных системах, когда в памяти одновременно содержатся составные части многих программ. Пока программа ждет считывания какой-либо собственной части, центральный процессор может быть отдан другому процессу.

Лекция 15. Страничная организация памяти

Большинство систем виртуальной памяти используют технологию под названием страничная организация памяти (paging), к описанию которой мы сейчас и приступим. На любом компьютере программы ссылаются на набор адресов памяти. Когда программа выполняет команду

MOV REG,1000

она осуществляет копирование содержимого ячейки памяти с адресом 1000 в REG (или наоборот, в зависимости от компьютера). Адреса могут генерироваться с использованием индексной адресации, базовых регистров, сегментных регистров и другими способами.

Такие сгенерированные программным способом адреса называются виртуальными адресами, именно они и формируют виртуальное адресное пространство. На компьютерах, не использующих виртуальную память, виртуальные адреса выставляются непосредственно на шине памяти, что приводит к чтению или записи слова физической памяти с таким же

адресом. При использовании виртуальной памяти виртуальные адреса не выставляются напрямую на шине памяти. Вместо этого они поступают в диспетчер памяти (Memory Management Unit (MMU)), который отображает виртуальные адреса на адреса физической памяти (рис. 19).

Очень простой пример работы такого отображения показан на рис. 20. В этом примере у нас есть компьютер, генерирующий 16-разрядные адреса, от 0 и до 64K – 1. Это виртуальные адреса. Но у этого компьютера есть только 32 Кбайт физической памяти.

И хотя для него можно написать программы объемом 64 Кбайт, целиком загрузить в память и запустить такие программы не представляется возможным. Но полный дубликат содержимого памяти всей программы, вплоть до 64 Кбайт, может размещаться на диске, позволяя вводить ее по частям по мере надобности.

Виртуальное адресное пространство состоит из блоков фиксированного размера, называемых страницами. Соответствующие блоки в физической памяти называются страничными блоками. Страницы и страничные блоки имеют, как правило, одинаковые размеры. В нашем примере их размер составляет 4 Кбайт, но в реальных системах используются размеры страниц от 512 байт до 1 Гбайт. При наличии 64 Кбайт виртуального адресного пространства и 32 Кбайт физической памяти мы получаем 16 виртуальных страниц и 8 страничных блоков. Перенос информации между оперативной памятью и диском всегда осуществляется целыми страницами. Многие процессоры поддерживают несколько размеров страниц, которые могут быть смешаны и подобраны по усмотрению операционной системы. Например, архитектура x86-64 поддерживает страницы размером 4 Кбайт, 2 Мбайт и 1 Гбайт, поэтому для пользовательских приложений можно использовать страницы размером 4 Кбайт, а для ядра — одну страницу размером 1 Гбайт. Почему иногда лучше использовать одну большую страницу, а не много маленьких, будет объяснено позже.

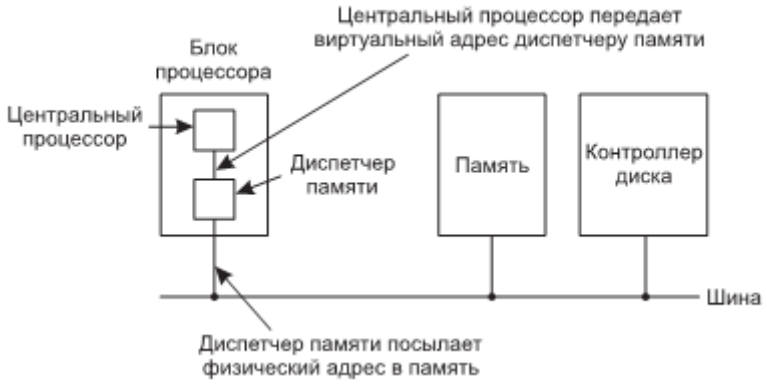

Рисунок 19 Расположение и предназначение диспетчера памяти. Здесь он показан в составе микросхемы центрального процессора, как это чаще всего и бывает в наши дни. Но логически он может размещаться и в отдельной микросхеме, как было в прошлом

На рис. 20 приняты следующие обозначения. Диапазон, помеченный 0K–4K, означает, что виртуальные или физические адреса этой страницы составляют от 0 до 4095. Диапазон 4K–8K ссылается на адреса от 4096 до 8191 и т. д. Каждая страница содержит строго 4096 адресов, которые начинаются с чисел, кратных 4096, и заканчиваются числами на единицу меньше чисел, кратных 4096.

К примеру, когда программа пытается получить доступ к адресу 0, используя команду

MOV REG,0

диспетчеру памяти посылается виртуальный адрес 0. Диспетчер видит, что этот виртуальный адрес попадает в страницу 0 (от 0 до 4095), которая в соответствии со своим отображением представлена страничным блоком 2 (от 8192 до 12 287). Соответственно, он трансформируется в адрес 8192, который и выставляется на шину. Память вообще не знает о существовании диспетчера и видит только запрос на чтение или запись по адресу 8192,

который и выполняет. Таким образом диспетчер памяти эффективно справляется с отображением всех виртуальных адресов между 0 и 4095 на физические адреса от 8192 до 12 287.

Аналогично этому команда

MOV REG,8192

эффективно преобразуется в

MOV REG,24576

поскольку виртуальный адрес 8192 (в виртуальной странице 2) отображается на 24 576 (в физической странице 6). В качестве третьего примера виртуальный адрес 20 500 отстоит на 20 байт от начала виртуальной страницы 5 (виртуальные адреса от 20 480 до 24 575) и отображается на физический адрес 12 288 + 20 = 12308.

Рисунок 20 Связь между виртуальными адресами и адресами физической памяти, получаемая с помощью таблицы страниц. Каждая страница начинается с адресов, кратных 4096, и завершается на 4095 адресов выше, поэтому 4K–8K на самом деле означает 4096–8191, а 8K–12K означает 8192–12 287

Сама по себе возможность отображения 16 виртуальных страниц на 8 страничных блоков за счет соответствующей настройки таблиц диспетчера памяти не решает проблемы превышения объема виртуальной памяти над объемом физической памяти.

Поскольку в нашем распоряжении только 8 физических страничных блоков, то на физическую память могут отображаться только 8 виртуальных страниц (рис. 20).

Остальные, отмеченные на рисунке крестиками, в число отображаемых не попадают. Реальное оборудование отслеживает присутствие конкретных страниц в физической памяти за счет бита присутствия-отсутствия.

А что происходит, если, к примеру, программа ссылается на неотображаемые адреса с помощью команды

MOV REG,32780

которая обращается к байту 12 внутри виртуальной страницы 8 (которая начинается с адреса 32 768)? Диспетчер памяти замечает, что страница не отображена (поскольку она на рисунке

помечена крестиком), и заставляет центральный процессор передать управление операционной системе. Это системное прерывание называется ошибкой отсутствия страницы (page fault). Операционная система выбирает редко используемый страничный блок и сбрасывает его содержимое на диск (если оно еще не там).

Затем она извлекает (также с диска) страницу, на которую была ссылка, и помещает ее в только что освободившийся страничный блок, вносит изменения в таблицы и заново запускает прерванную команду.

К примеру, если операционная система решит выселить содержимое страничного блока 1, она загрузит виртуальную страницу 8 начиная с физического адреса 4096 и внесет два изменения в карту диспетчера памяти. Сначала в запись о виртуальной странице 1 будет внесена пометка о том, что эта страница не отображена, чтобы при любом будущем обращении к виртуальным адресам в диапазоне от 4096 до 8191 вызывалось системное прерывание. Затем крестик в записи, относящейся к виртуальной странице 8, будет изменен на цифру 1, поэтому при повторном выполнении прерванной команды произойдет отображение виртуального адреса 32 780 на физический адрес 4108 (4096 + 12).

Тема 3.3. Общие сведения о процессах и потоках

Лекция 16. Модель процесса. Создание процесса. Завершение процесса. Иерархия процесса. Состояние процесса. Реализация процесса.

Основным понятием в любой операционной системе является процесс: абстракция, описывающая выполняющуюся программу.

Модель процесса. В этой модели все выполняемое на компьютере программное обеспечение, иногда включая операционную систему, сведено к ряду последовательных процессов, или, для краткости, просто процессов. Процесс — это просто экземпляр выполняемой программы, включая текущие значения счетчика команд, регистров и переменных. Концептуально у каждого процесса есть свой, виртуальный, центральный процессор. Разумеется, на самом деле настоящий центральный процессор постоянно переключается между процессами, но чтобы понять систему, куда проще думать о наборе процессов, запущенных в (псевдо) параллельном режиме, чем пытаться отслеживать, как центральный процессор переключается между программами. Это постоянное переключение между процессами, называется мультипрограммированием, или многозадачным режимом работы.

На рис. 21, а показан компьютер, работающий в многозадачном режиме и имеющий в памяти четыре программы. На рис. 21, б показаны четыре процесса, каждый из которых имеет собственный алгоритм управления (то есть собственный логический счетчик команд) и работает независимо от всех остальных. Понятно, что на самом деле имеется только один физический счетчик команд, поэтому при запуске каждого процесса его логический счетчик команд загружается в реальный счетчик. Когда работа с процессом будет на некоторое время прекращена, значение физического счетчика команд сохраняется в логическом счетчике команд, размещаемом процессом в памяти.

На рис. 21, в показано, что за довольно длительный период наблюдения продвинулись вперед все процессы, но в каждый отдельно взятый момент времени реально работает только один процесс.

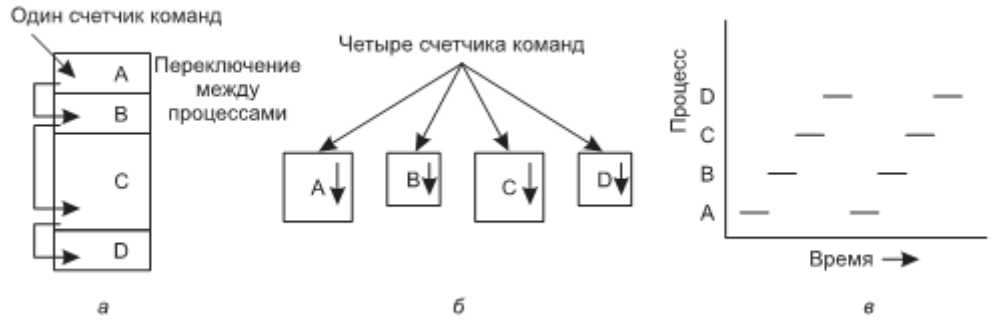

Рисунок 21. Компьютер: а — четыре программы, работающие в многозадачном режиме; б — концептуальная модель четырех независимых друг от друга последовательных процессов; в — в отдельно взятый момент активна только одна программа

Создание процесса. Операционным системам необходим какой-нибудь способ для создания процессов. В самых простых системах или в системах, сконструированных для запуска только одного приложения (например, в контроллере микроволновой печи), появляется возможность присутствия абсолютно всех необходимых процессов при вводе системы в действие. Но в универсальных системах нужны определенные способы создания и прекращения процессов по мере необходимости.

Существуют четыре основных события, приводящих к созданию процессов.

1. Инициализация системы.

2. Выполнение работающим процессом системного вызова, предназначенного для создания процесса.

3. Запрос пользователя на создание нового процесса.

4. Инициация пакетного задания.

Завершение процесса. После создания процесс начинает работать и выполняет свою задачу. Но ничто не длится вечно, даже процессы. Рано или поздно новые процессы будут завершены, обычно в силу следующих обстоятельств:

- обычного выхода (добровольно);
- выхода при возникновении ошибки (добровольно);
- возникновения фатальной ошибки (принудительно);
- уничтожения другим процессом (принудительно).

Иерархии процессов. В некоторых системах, когда процесс порождает другой процесс, родительский и дочерний процессы продолжают оставаться определенным образом связанными друг с другом. Дочерний процесс может и сам создать какие-нибудь процессы, формируя иерархию процессов.

В отличие от этого в Windows не существует понятия иерархии процессов, и все процессы являются равнозначными. Единственным намеком на иерархию процессов можно считать присвоение родительскому процессу, создающему новый процесс, специального маркера (называемого дескриптором), который может им использоваться для управления дочерним процессом. Но он может свободно передавать этот маркер какому-нибудь другому процессу, нарушая тем самым иерархию. А в UNIX процессы не могут лишаться наследственной связи со своими дочерними процессами.

Состояния процессов. Несмотря на самостоятельность каждого процесса, наличие собственного счетчика команд и внутреннего состояния, процессам зачастую необходимо взаимодействовать с другими процессами.

На рис. 22 показана диаграмма, отображающая три состояния, в которых может находиться процесс:

\ выполняемый (в данный момент использующий центральный процессор);

\ готовый (работоспособный, но временно приостановленный, чтобы дать возможность выполнения другому процессу);

\ заблокированный (неспособный выполняться, пока не возникнет какое-нибудь внешнее событие).

Логически первые два состояния похожи друг на друга. В обоих случаях процесс желает выполняться, но во втором состоянии временно отсутствует доступный для этого процессор. Третье состояние коренным образом отличается от первых двух тем, что процесс не может выполняться, даже если процессору кроме него больше нечем заняться.

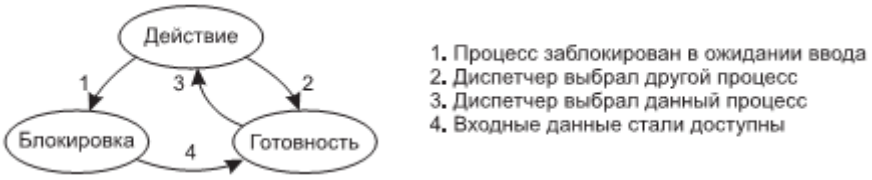

Рисунок 22. Процесс может быть в выполняемом, заблокированном или готовом состоянии. Стрелками показаны переходы между этими состояниями

Как показано на рисунке, между этими тремя состояниями могут быть четыре перехода. Переход 1 происходит в том случае, если операционная система определит, что процесс в данный момент выполняться не может. В некоторых системах для перехода в заблокированное состояние процесс может осуществить такой системный вызов, как pause. В других системах, включая UNIX, когда процесс осуществляет чтение из канала или специального файла (например, с терминала) и доступные входные данные отсутствуют, процесс блокируется автоматически.

Переходы 2 и 3 вызываются планировщиком процессов, который является частью операционной системы, без какого-либо оповещения самого процесса. Переход 2 происходит, когда планировщик решит, что выполняемый процесс продвинулся достаточно далеко и настало время позволить другому процессу получить долю рабочего времени центрального процессора. Переход 3 происходит, когда все другие процессы получили причитающуюся им долю времени и настал момент предоставить центральный процессор первому процессу для возобновления его выполнения.

Переход 4 осуществляется в том случае, если происходит внешнее событие, ожидавшееся процессом (к примеру, поступление входных данных). Если к этому моменту нет других выполняемых процессов, будет вызван переход 3 и процесс возобновится.

В противном случае ему придется немного подождать в состоянии готовности, пока не станет доступен центральный процессор и не придет его очередь.

Использование модели процесса облегчает представление о том, что происходит внутри системы.

Реализация процессов. Для реализации модели процессов операционная система ведет таблицу (состоящую из массива структур), называемую таблицей процессов, в которой каждая запись соответствует какому-нибудь процессу. Эти записи содержат важную информацию о состоянии процесса, включая счетчик команд, указатель стека, распределение памяти, состояние открытых им файлов, его учетную и планировочную информацию и все остальное, касающееся процесса, что должно быть сохранено, когда процесс переключается из состояния выполнения в состояние готовности или блокировки, чтобы позже он мог возобновить выполнение, как будто никогда не останавливался.

Лекция 17. Применение потоков. Классическая модель потоков. Реализация потоков.

Применение потоков. Необходимость в подобных мини-процессах, называемых потоками, обусловливается, целым рядом причин. Основная причина использования потоков заключается в том, что во многих приложениях одновременно происходит насколько действий, часть которых может периодически быть заблокированной. Модель программирования упрощается за счет разделения такого приложения на несколько последовательных потоков, выполняемых в квазипараллельном режиме.

Предположим, что текстовый процессор написан как двухпоточная программа. Один из потоков взаимодействует с пользователем, а другой занимается переформатированием в фоновом режиме. Как только предложение с первой страницы будет удалено, поток, отвечающий за взаимодействие с пользователем, приказывает потоку, отвечающему за формат, переформатировать всю книгу. Пока взаимодействующий поток продолжает отслеживать события клавиатуры и мыши, реагируя на простые команды вроде прокрутки первой страницы, второй поток с большой скоростью выполняет вычисления. Если немного повезет, то переформатирование закончится как раз перед тем, как пользователь запросит просмотр 600-й страницы, которая тут же сможет быть отображена.

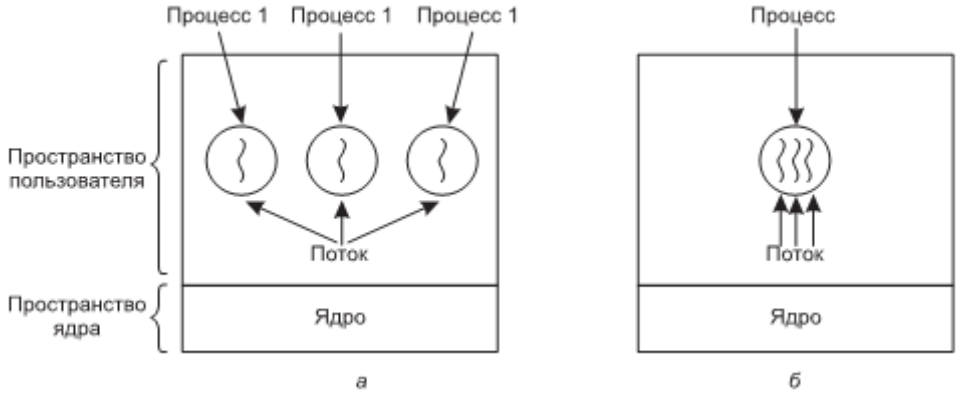

Рисунок 23. а — три процесса, у каждого из которых по одному потоку; б — один процесс с тремя потоками

Классическая модель потоков. На рис. 23, а показаны три традиционных процесса. У каждого из них имеется собственное адресное пространство и единственный поток управления. В отличие от этого, на рис. 23, б показан один процесс, имеющий три потока управления. Хотя в обоих случаях у нас имеется три потока, на рис. 23, а каждый из них работает в собственном адресном пространстве, а на рис. 23, б все три потока используют общее адресное пространство.

Различные потоки в процессе не обладают той независимостью, которая есть у различных процессов. У всех потоков абсолютно одно и то же адресное пространство, а значит, они так же совместно используют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому адресу памяти в пределах адресного пространства процесса, один поток может считывать данные из стека другого потока, записывать туда свои данные и даже стирать оттуда данные. Защита между потоками отсутствует, потому что ее невозможно осуществить и в ней нет необходимости. В отличие от различных процессов, которые могут принадлежать различным пользователям и которые могут враждовать друг с другом, один процесс всегда принадлежит одному и тому же пользователю, который, по-видимому, и создал несколько потоков для их совместной работы, а не для вражды. В дополнение к использованию общего адресного пространства все потоки, как показано в рис. 24, могут совместно использовать одни и те же открытые файлы, дочерние процессы, ожидаемые и обычные сигналы и т. п.

Поэтому структура, показанная на рис. 23, а, может использоваться, когда все три процесса фактически не зависят друг от друга, а структура, показанная на рис. 23, б, может применяться, когда три потока фактически являются частью одного и того же задания и активно и тесно сотрудничают друг с другом.

Элементы, присущие каждому процессу	Элементы, присущие каждому потоку
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные аварийные сигналы	
Сигналы и обработчики сигналов	
Учетная информация	

Рисунок 24. Использование объектов потоками

Следует учесть, что каждый поток имеет собственный стек (рис. 25). Стек каждого потока содержит по одному фрейму для каждой уже вызванной, но еще не возвратившей управление процедуры. Такой фрейм содержит локальные переменные процедуры и адрес возврата управления по завершении ее вызова. Например, если процедура X вызывает процедуру Y, а Y вызывает процедуру Z, то при выполнении Z в стеке будут фреймы для X, Y и Z. Каждый поток будет, как правило, вызывать различные процедуры и, следовательно, иметь среду выполнения, отличающуюся от среды выполнения других потоков. Поэтому каждому потоку нужен собственный стек.

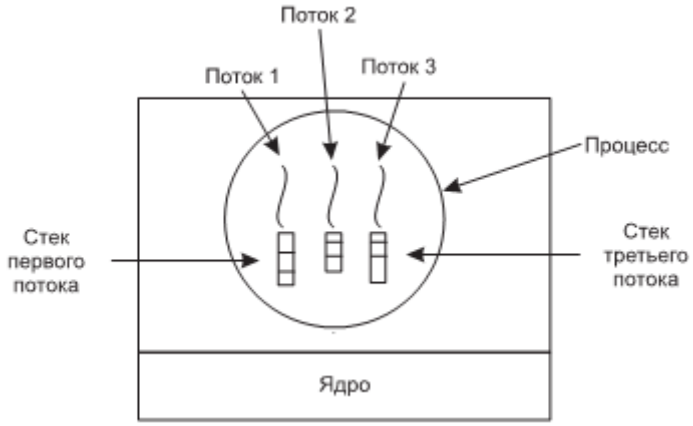

Рисунок 25. У каждого потока имеется собственный стек

Реализация потоков.

Есть два основных места реализации набора потоков: в пользовательском пространстве и в ядре.

У всех этих реализаций одна и та же общая структура (рис. 26, а). Потоки запускаются поверх системы поддержки исполнения программ (run-time system), которая представляет собой набор процедур, управляющих потоками.

Когда потоки управляются в пользовательском пространстве, каждому процессу необходимо иметь собственную таблицу потоков, чтобы отслеживать потоки, имеющиеся в этом процессе. Эта таблица является аналогом таблицы процессов, имеющейся в ядре, за исключением того, что в ней содержатся лишь свойства, принадлежащие каждому потоку, такие как счетчик команд потока, указатель стека, регистры, состояние и т. д.

Таблица потоков управляется системой поддержки исполнения программ. Когда поток переводится в состояние готовности или блокируется, информация, необходимая для возобновления его выполнения, сохраняется в таблице потоков, точно так же, как ядро хранит информацию о процессах в таблице процессов.

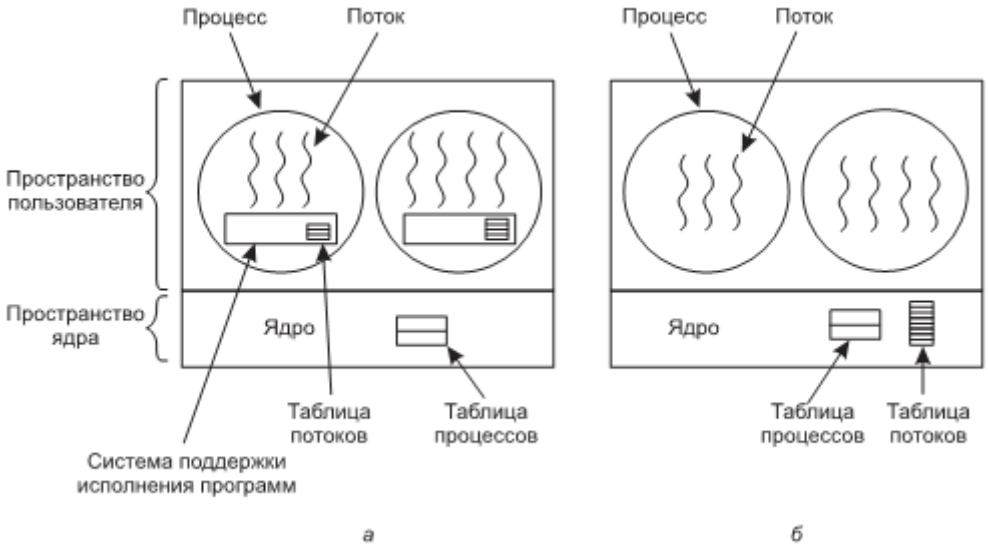

Рисунок 26 Набор потоков: а — на пользовательском уровне; б — управляемый ядром

Теперь давайте рассмотрим, что произойдет, если ядро будет знать о потоках и управлять ими. Как показано на рис. 26, б, здесь уже не нужна система поддержки исполнения программ. Также здесь нет и таблицы процессов в каждом потоке. Вместо этого у ядра есть таблица потоков, в которой отслеживаются все потоки, имеющиеся в системе. Когда потоку необходимо создать новый или уничтожить существующий поток, он обращается к ядру, которое и создает или разрушает путем обновления таблицы потоков в ядре.

В таблице потоков, находящейся в ядре, содержатся регистры каждого потока, состояние и другая информация. Вся информация аналогична той, которая использовалась для потоков, создаваемых на пользовательском уровне, но теперь она содержится в ядре, а не в пространстве пользователя (внутри системы поддержки исполнения программ). Эта информация является подмножеством той информации, которую поддерживают традиционные ядра в отношении своих однопоточных процессов, то есть подмножеством состояния процесса. Вдобавок к этому ядро поддерживает также традиционную таблицу процессов с целью их отслеживания.

Тема 3.4. Взаимодействие и планирование процессов. Лекция 18. Взаимодействие процессов

Довольно часто процессам необходимо взаимодействовать с другими процессами. Например, в канале оболочки выходные данные одного процесса могут передаваться другому процессу, и так далее вниз по цепочке. Поэтому возникает необходимость во взаимодействии процессов, и желательно по хорошо продуманной структуре без использования прерываний. В следующих разделах мы рассмотрим некоторые вопросы, связанные со взаимодействием процессов, или межпроцессным взаимодействием (InterProcess Communication (IPC)).

Критические области

Как же избежать состязательной ситуации? Ключом к предупреждению проблемы в этой и во многих других ситуациях использования общей памяти, общих файлов и вообще чего-нибудь общего может послужить определение способа, при котором в каждый конкретный момент времени доступ к общим данным для чтения и записи может получить только один процесс. Иными словами, нам нужен способ взаимного исключения, то есть некий способ, обеспечивающий правило, при котором если общие данные или файл используются одним процессом, возможность их использования всеми другими процессами исключается. Описанные выше трудности произошли благодаря тому, что процесс Б стал использовать общие переменные еще до того, как процесс А завершил работу с ними. Выбор подходящих

элементарных операций для достижения взаимного исключения является основной проблемой конструирования любой операционной системы, и именно ее мы будем подробно рассматривать в следующих разделах.

Проблемы обхода состязательных ситуаций могут быть сформулированы также в абстрактной форме. Какую-то часть времени процесс занят внутренними вычислениями и чем-нибудь другим, не создающим состязательных ситуаций. Но иногда он вынужден обращаться к общей памяти или файлам либо совершать какие-нибудь другие значимые действия, приводящие к состязаниям. Та часть программы, в которой используется доступ к общей памяти, называется критической областью или критической секцией. Если бы удалось все выстроить таким образом, чтобы никакие два процесса не находились одновременно в своих критических областях, это позволило бы избежать состязаний.

Хотя выполнение этого требования позволяет избежать состязательных ситуаций, его недостаточно для того, чтобы параллельные процессы правильно выстраивали совместную работу и эффективно использовали общие данные. Для приемлемого решения необходимо соблюдение четырех условий:

1. Два процесса не могут одновременно находиться в своих критических областях.

2. Не должны выстраиваться никакие предположения по поводу скорости или количества центральных процессоров.

3. Никакие процессы, выполняемые за пределами своих критических областей, не могут блокироваться любым другим процессом.

4. Процессы не должны находиться в вечном ожидании входа в свои критические области.

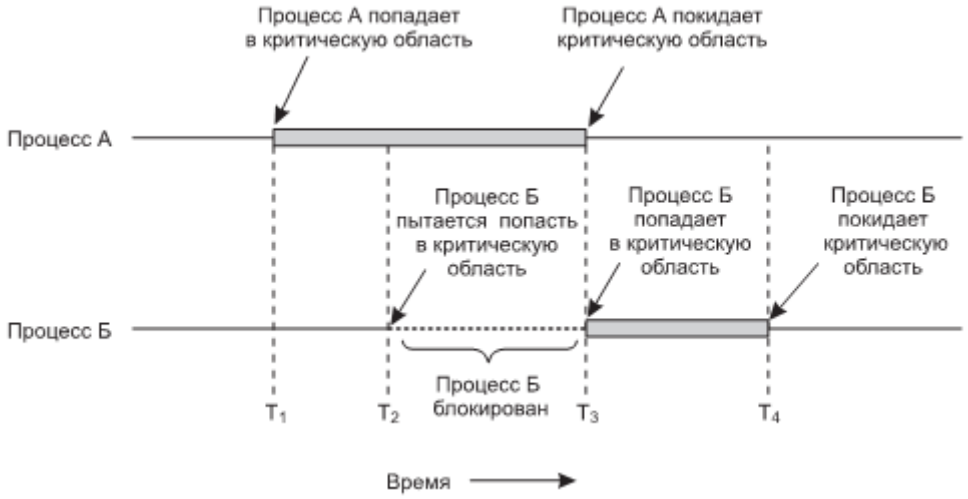

Рисунок 27. Взаимное исключение использования критических областей

В абстрактном смысле необходимое нам поведение показано на рис. 27. Мы видим, что процесс А входит в свою критическую область во время Т1. Чуть позже, когда наступает время Т2, процесс Б пытается войти в свою критическую область, но терпит неудачу, поскольку другой процесс уже находится в своей критической области, а мы допускаем это в каждый момент времени только для одного процесса. Следовательно,Б временно приостанавливается до наступления времени Т3, когда А покинет свою критическую область, позволяя Б тут же войти в свою критическую область. Со временем (в момент Т4) Б покидает свою критическую область, и мы возвращаемся в исходную ситуацию, когда ни один из процессов не находится в своей критической области.

Лекция 19. Семафоры. Мьютексы.

Семафоры

Ситуация изменилась в 1965 году, когда Дейкстра предложил использовать целочисленную переменную для подсчета количества активизаций, отложенных на будущее.

Он предложил учредить новый тип переменной — семафор (semaphore). Значение семафора может быть равно 0, что будет свидетельствовать об отсутствии сохраненных активизаций, или иметь какое-нибудь положительное значение, если ожидается не менее одной активизации. Дейкстра предложил использовать две операции с семафорами, которые сейчас обычно называют down и up (обобщения sleep и wakeup соответственно). Операция down выясняет, отличается ли значение семафора от 0. Если отличается, она уменьшает это значение на 1 (то есть использует одну сохраненную активизацию) и продолжает свою работу.

Если значение равно 0, процесс приостанавливается, не завершая в этот раз операцию down. И проверка значения, и его изменение, и, возможно, приостановка процесса осуществляются как единое и неделимое атомарное действие. Тем самым гарантируется, что с началом семафорной операции никакой другой процесс не может получить доступ к семафору до тех пор, пока операция не будет завершена или заблокирована. Атомарность является абсолютно необходимым условием для решения проблем синхронизации и исключения состязательных ситуаций. Атомарные действия, в которых группа взаимосвязанных операций либо выполняется без каких-либо прерываний, либо вообще не выполняется, приобрели особую важность и во многих других областях информатики.

Операция up увеличивает значение, адресуемое семафором, на 1. Если с этим семафором связаны один или более приостановленных процессов, способных завершить ранее начатые операции down, система выбирает один из них (к примеру, произвольным образом) и позволяет ему завершить его операцию down. Таким образом, после применения операции up в отношении семафора, с которым были связаны приостановленные процессы, значение семафора так и останется нулевым, но количество приостановленных процессов уменьшится на 1. Операция увеличения значения семафора на 1 и активизации одного из процессов также является неделимой. Ни один из процессов не может быть заблокирован при выполнении операции up, равно как ни один из процессов не может быть заблокирован при выполнении wakeup в предыдущей модели. Между прочим, в первоначальном варианте своей работы Дейкстра вместо down и up использовал имена P и V соответственно. Но в них не было никакого мнемонического смысла для тех, кто не говорит по-голландски, да и для тех, кто говорит, смысл был едва уловим — Proberen (пытаться) и Verhogen (поднимать выше), поэтому вместо них мы будет употреблять down и up. Впервые они были представлены в языке программирования Algol 68.

Мьютексы

Иногда при невостребованности возможностей семафоров в качестве счетчиков используется их упрощенная версия, называемая мьютексом. Мьютексы справляются лишь с управлением взаимным исключением доступа к общим ресурсам или фрагментам кода. Простота и эффективность реализации мьютексов делает их особенно полезными для совокупности потоков, целиком реализованных в пользовательском пространстве.

Мьютекс — это совместно используемая переменная, которая может находиться в одном из двух состояний: заблокированном или незаблокированном. Следовательно, для их представления нужен только один бит, но на практике зачастую используется целое число, при этом нуль означает незаблокированное, а все остальные значения — заблокированное состояние. Для работы с мьютексами используются две процедуры.

Когда потоку (или процессу) необходим доступ к критической области, он вызывает процедуру mutex_lock. Если мьютекс находится в незаблокированном состоянии (означающем доступность входа в критическую область), вызов проходит удачно и вызывающий поток может свободно войти в критическую область.

В то же время, если мьютекс уже заблокирован, вызывающий поток блокируется до тех пор, пока поток, находящийся в критической области, не завершит свою работу и не вызовет процедуру mutex_unlock. Если на мьютексе заблокировано несколько потоков, то произвольно

выбирается один из них, которому разрешается воспользоваться заблокированностью других потоков.

Лекция 20. Планирование процессов

Планирование в пакетных системах

Теперь давайте перейдем от общих вопросов планирования к специализированным алгоритмам. В этом разделе будут рассмотрены алгоритмы, используемые в пакетных системах, а в следующих разделах мы рассмотрим алгоритмы, используемые в интерактивных системах и системах реального времени. Следует заметить, что некоторые алгоритмы используются как в пакетных, так и в интерактивных системах. Мы рассмотрим их чуть позже.

«Первым пришел — первым обслужен».

Наверное, наипростейшим из всех алгоритмов планирования будет неприоритетный алгоритм, следующий принципу «первым пришел — первым обслужен». При использовании этого алгоритма центральный процессор выделяется процессам в порядке поступления их запросов. По сути, используется одна очередь процессов, находящихся в состоянии готовности. Когда рано утром в систему извне попадает первое задание, оно тут же запускается на выполнение и получает возможность выполняться как угодно долго. Оно не прерывается по причине слишком продолжительного выполнения.

Другие задания по мере поступления помещаются в конец очереди. При блокировке выполняемого процесса следующим запускается первый процесс, стоящий в очереди. Когда заблокированный процесс переходит в состояние готовности, он, подобно только что поступившему заданию, помещается в конец очереди, после всех ожидающих процессов.

Сильной стороной этого алгоритма является простота его понимания и такая же простота его программирования. Его справедливость сродни справедливости распределения дефицитных билетов на спортивные или концертные зрелища или мест в очереди на новые айфоны тем людям, которые заняли очередь с двух часов ночи.

При использовании этого алгоритма отслеживание готовых процессов осуществляется с помощью единого связанного списка. Выбор следующего выполняемого процесса сводится к извлечению одного процесса из начала очереди. Добавление нового задания или разблокированного процесса сводится к присоединению его к концу очереди. Что может быть проще для восприятия и реализации?

К сожалению, принцип «первым пришел — первым обслужен» страдает и существенными недостатками. Предположим, что используются один процесс, ограниченный скоростью вычислений, который всякий раз запускается на 1 с, и множество процессов, ограниченных скоростью работы устройств ввода-вывода, незначительно использующих время центрального процессора, но каждый из которых должен осуществить 1000 считываний с диска, прежде чем завершить свою работу. Процесс, ограниченный скоростью вычислений, работает в течение 1 с, а затем переходит к чтению блока данных с диска. Теперь запускаются все процессы ввода-вывода и приступают к чтению данных с диска. Когда процесс, ограниченный скоростью вычислений, получает свой блок данных с диска, он запускается еще на 1 с, а за ним непрерывной чередой следуют все процессы, ограниченные скоростью работы устройств ввода-вывода.

В итоге каждый процесс, ограниченный скоростью работы устройств ввода-вывода, считывает один блок в секунду, и завершение его работы займет 1000 с. Если используется алгоритм планирования, выгружающий процесс, ограниченный скоростью вычислений, каждые 10 мс, то процессы, ограниченные скоростью работы устройств ввода-вывода, завершаются за 10 с вместо 1000 с, при этом особо не замедляя работу процесса, ограниченного скоростью вычислений.

Планирование в интерактивных системах

Теперь давайте рассмотрим некоторые алгоритмы, которые могут быть использованы в интерактивных системах. Они часто применяются на персональных компьютерах, серверах и в других разновидностях систем.

«Приоритетное планирование»

В циклическом планировании явно прослеживается предположение о равнозначности всех процессов. Зачастую люди, обладающие многопользовательскими компьютерами и работающие на них, имеют на этот счет совершенно иное мнение. К примеру, в университете иерархия приоритетности должна нисходить от декана к факультетам, затем к профессорам, секретарям, техническим работникам, а уже потом к студентам.

Необходимость учета внешних факторов приводит к приоритетному планированию. Основная идея проста: каждому процессу присваивается значение приоритетности и запускается тот процесс, который находится в состоянии готовности и имеет наивысший приоритет.

Даже если у персонального компьютера один владелец, на нем могут выполняться несколько процессов разной степени важности. Например, фоновому процессу, отправляющему электронную почту, должен быть назначен более низкий приоритет, чем процессу, воспроизводящему на экране видеофильм в реальном времени.

Чтобы предотвратить бесконечное выполнение высокоприоритетных процессов, планировщик должен понижать уровень приоритета текущего выполняемого процесса с каждым сигналом таймера (то есть с каждым его прерыванием). Если это действие приведет к тому, что его приоритет упадет ниже приоритета следующего по этому показателю процесса, произойдет переключение процессов. Можно выбрать и другую альтернативу: каждому процессу может быть выделен максимальный квант допустимого времени выполнения. Когда квант времени будет исчерпан, шанс запуска будет предоставлен другому процессу, имеющему наивысший приоритет.

Приоритеты могут присваиваться процессам в статическом или в динамическом режиме. Например, на военных компьютерах процессы, инициированные генералами, могут начинать свою работу с приоритетом, равным 100, процессы, инициированными полковниками, — с приоритетом, равным 90, майорами — с приоритетом 80, капитанами — 70, лейтенантами — 60 и так далее вниз по табели о рангах. А в коммерческом компьютерном центре высокоприоритетные задания могут стоить 100 долларов в час, задания со средним приоритетом — 75, а задания с низким приоритетом — 50. В UNIX-системах есть команда nice, позволяющая пользователю добровольно снизить приоритет своего процесса, чтобы угодить другим пользователям, но ею никто никогда не пользуется.

Приоритеты также могут присваиваться системой в динамическом режиме с целью достижения определенных системных задач. К примеру, некоторые процессы испытывают существенные ограничения по скорости работы устройств ввода-вывода и проводят большую часть своего времени в ожидании завершения операций ввода-вывода. Как только такому процессу понадобится центральный процессор, он должен быть предоставлен немедленно, чтобы процесс мог приступить к обработке следующего запроса на ввод-вывод данных, который затем может выполняться параллельно с другим процессом, занятым вычислениями. Если заставить процесс, ограниченный скоростью работы устройств ввода-вывода, долго ждать предоставления центрального процессора, это будет означать, что он занимает оперативную память неоправданно долго. Простой алгоритм успешного обслуживания процессов, ограниченных скоростью работы устройств ввода-вывода, предусматривает установку значения приоритета в 1/f, где f — это часть последнего кванта времени, использованного этим процессом. Процесс, использовавший только 1 мс из отпущенных ему 50 мс кванта времени, должен получить приоритет 50, в то время как процесс, проработавший до блокировки 25 мс, получит приоритет, равный 2, а процесс, использовавший весь квант времени, получит приоритет, равный 1.

Зачастую бывает удобно группировать процессы по классам приоритетности и использовать приоритетное планирование применительно к этим классам, а внутри каждого класса использовать циклическое планирование. На рис. 28 показана система с четырьмя классами приоритетности. Алгоритм планирования выглядит следующим образом: если есть готовые к запуску процессы с классом приоритетности 4, следует запустить каждый из них на один квант времени по принципу циклического планирования, при этом вовсе не беспокоясь о классах с

более низким приоритетом. Когда класс с уровнем приоритета 4 опустеет, в циклическом режиме запускаются процессы с классом приоритетности 3. Если опустеют оба класса, 4 и 3, в циклическом режиме запускаются процессы с классом приоритетности 2 и т. д. Если приоритеты каким-то образом не будут уточняться, то все классы с более низким уровнем приоритета могут «умереть голодной смертью».

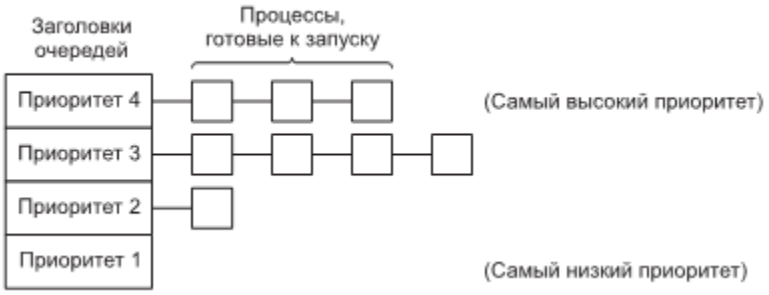

Рисунок 28. Алгоритм планирования для четырех классов приоритетности

Тема 3.5. Работа в операционных системах и средах Лекция 21. Управление безопасностью

Домены защиты. Компьютерная система содержит множество ресурсов или объектов, нуждающихся в защите. Этими объектами могут быть оборудование (например, центральные процессоры, страницы памяти, дисковые приводы или принтеры) или программное обеспечение (например, процессы, файлы, базы данных или семафоры).

У каждого объекта есть уникальное имя, по которому к нему можно обращаться, и конечный набор операций, которые процессы могут выполнять в отношении этого объекта. Файлу свойственны операции read и write, а семафору — операции up и down.

Совершенно очевидно, что нужен способ запрещения процессам доступа к тем объектам, к которым у них нет прав доступа. Более того, этот механизм должен также предоставлять возможность при необходимости ограничивать процессы поднабором разрешенных операций. Например, процессу A может быть дано право проводить чтение данных из файла F, но не разрешено вести запись в этот файл.

Чтобы рассмотреть различные механизмы защиты, полезно ввести понятие домена.

Домен (domain) представляет собой множество пар (объект, права доступа). Каждая пара определяет объект и некоторое подмножество операций, которые могут быть выполнены в отношении этого объекта. Права доступа (rights) означают в данном контексте разрешение на выполнение той или иной операции. Зачастую домен соотносится с отдельным пользователем, сообщая о том, что может, а что не может сделать этот пользователь, но он может также иметь и более общий характер, распространяясь не только на отдельного пользователя. К примеру, сотрудники одной группы программистов, работающие над одним и тем же проектом, могут целиком принадлежать к одному и тому же домену и иметь доступ к файлам проекта.

Распределение объектов по доменам зависит от особенностей того, кому и о чем нужно знать. Тем не менее одним из фундаментальных понятий является принцип минимальных полномочий (Principle of Least Authority (POLA)), или принцип необходимого знания. В общем, безопасность проще соблюсти, когда у каждого домена имеется минимум объектов и привилегий для работы с ними и нет ничего лишнего.

На рис. 29 показаны три домена с объектами в каждом из них и правами на чтение, запись и выполнение (Read, Write, eXecute) применительно к каждому объекту. Заметьте, что объект Printer1 одновременно находится в двух доменах и обладает одинаковыми правами в каждом из них. Объект File1 также присутствует в двух доменах, но имеет разные права в каждом из них.

В любой момент времени каждый процесс работает в каком-нибудь домене защиты. Иными словами, существует некая коллекция объектов, к которым он может иметь доступ, и для каждого объекта у него имеется некий набор прав. Во время работы процессы могут также переключаться с одного домена на другой. Правила переключения между доменами сильно зависят от конкретной системы.

Рисунок 29 Три домена защиты

Матрица для рис.29 показана на рис. 30. Располагая этой матрицей и номером текущего домена, операционная система может определить, разрешен ли из конкретного домена определенный вид доступа к заданному объекту.

Само по себе переключение между доменами также может быть легко включено в ту же табличную модель, если в качестве объекта представить сам домен, в отношении которого может быть разрешена операция входа — enter. На рис. 31 снова показана матрица с рис. 30, но теперь на ней в качестве объектов фигурируют и сами три домена. Процессы в домене 1 могут переключаться на домен 2, но обратно вернуться уже не могут.

Эта ситуация моделирует в UNIX выполнение программы с установленным битом SETUID. Другие переключения доменов в данном примере не разрешены.

	Объект							
Домен	Файл 1	Файл 2	Файл 3	Файл 4	Файл 5	Файл 6	Принтер 1	Плоттер 2
1	Чтение	Чтение Запись						
2			Чтение	Чтение Запись Исполнение	Чтение Запись		Запись	
3						Чтение Запись Исполнение	Запись	Запись

Рисунок 30. Матрица защиты

Домен	Файл 1	Файл 2	Файл 3	Файл 4	Файл 5	Файл 6	Принтер 1	Плоттер 2	Домен 1	Домен 2	Домен 3
1	Чтение	Чтение Запись								Enter	
2			Чтение	Чтение Запись Исполнение	Чтение Запись		Запись				
3						Чтение Запись Исполнение	Запись	Запись			

Рисунок 31. Матрица защиты с доменами в качестве объектов

Лекция 22. Планирование и разработка операционной системы

Цели планирования

Чтобы проект операционной системы был успешным, разработчики должны иметь четкое представление о том, чего они хотят. При отсутствии цели очень трудно принимать последующие решения. Чтобы этот вопрос стал понятнее, полезно взглянуть на два языка программирования, PL/1 и C. Язык PL/1 был разработан корпорацией IBM в 1960-е годы, так как поддерживать одновременно FORTRAN и COBOL и слышатьпри этом за спиной ворчание ученых о том, что Algol лучше, чем FORTRAN и COBOL вместе взятые, было невыносимо. Поэтому был образован комитет для создания нового языка, удовлетворяющего запросам всех программистов: PL/1. Этот язык обладал некоторыми чертами языка FORTRAN, некоторыми особенностями языка COBOL и некоторыми свойствами языка Algol. Проект потерпел неудачу, потому что ему недоставало единой концепции. Проект представлял собой набор свойств, конфликтующих друг с другом, к тому же язык PL/1 был слишком громоздким и неуклюжим, чтобы программы на нем можно было эффективно компилировать.

Теперь взглянем на язык C. Он был спроектирован всего одним человеком, Деннисом Ритчи, для единственной цели — системного программирования. Успех его был колоссален, и это не в последнюю очередь объяснялось тем, что Ритчи знал, чего хотел, а чего не хотел. В результате спустя десятилетия после своего появления этот язык все еще широко распространен. Наличие четкого представления о своих целях является решающим.

Чего же хотят разработчики операционных систем? Очевидно, ответ варьируется от системы к системе и будет разным для встроенных **и серверных систем.** Для универсальных операционных систем основными являются следующие четыре пункта:

1. Определение абстракций.

2. Предоставление примитивных операций.

3. Обеспечение изоляции.

4. Управление аппаратурой.

Рассмотрим каждый из этих пунктов.

Наиболее важная, но, вероятно, наиболее сложная задача операционной системы заключается в определении правильных абстракций. Некоторые из них, такие как процессы и файлы, используются уже так давно, что могут показаться очевидными. Другие, такие как потоки исполнения, представляют собой более новые и потому не столь устоявшиеся понятия. Например, если состоящий из нескольких потоков процесс, один из потоков которого блокирован вводом с клавиатуры, клонируется, то должен ли поток в новом процессе также ожидать ввода с клавиатуры? Другие абстракции относятся к синхронизации, сигналам, модели памяти, моделированию ввода-вывода и иным областям.

Каждая абстракция может быть реализована в виде конкретных структур данных. Пользователи могут создавать процессы, файлы, каналы и т. д. Управляют этими структурами данных при помощи примитивных операций. Например, пользователи могут читать и писать файлы. Примитивные операции реализуются в виде системных вызовов. С точки зрения пользователя, сердце операционной системы формируется абстракциями, а операции над ними возможны при помощи системных вызовов.

Поскольку на одном компьютере могут одновременно зарегистрироваться несколько пользователей, операционная система должна предоставлять механизмы для отделения их друг от друга. Один пользователь не должен вмешиваться в работу другого. Концепция процессов широко используется для группирования ресурсов с целью их защиты.

Как правило, защищаются файлы и другие структуры данных. Еще одним местом, где разделение играет весьма важную роль, является виртуализация: гипервизор должен обеспечить обособленность виртуальных машин от всех сложностей работы других виртуальных машин. Ключевая цель проектирования операционной системы заключается в том, чтобы гарантировать, что каждый пользователь может выполнять только разрешенные ему действия с данными, к которым у него есть право доступа. Однако пользователям бывает необходимо совместное использование данных и ресурсов, поэтому изоляция должна быть избирательной и контролироваться пользователями.

Все это существенно усложняет устройство операционной системы. Почтовые программы не должны наносить вред веб-браузерам. Даже если существует только один пользователь, разные процессы должны быть разделены. Чтобы защитить процессы друг от друга, на некоторых системах, таких как Android, каждый процесс, принадлежащий одному и тому же пользователю, будет запускаться с иным пользовательским идентификатором.

С этим вопросом тесно связана проблема изолирования отказов. Если какая-либо часть системы выйдет из строя (чаще всего это один из пользовательских процессов), то сбойный процесс не должен нарушить работу всей операционной системы. Устройство операционной системы должно гарантировать изоляцию различных частей операционной системы друг от друга, чтобы их сбои были независимыми. Заходя еще дальше, можно задаться вопросом: а не должна ли операционная система быть устойчивой к сбоям и самоисцеляющейся?

Наконец, операционная система должна управлять аппаратурой. В частности, она должна заботиться обо всех низкоуровневых микросхемах, таких как контроллеры прерываний и контроллеры шин. Она также должна обеспечивать каркас для того, чтобы драйверы устройств могли управлять крупными устройствами ввода-вывода, такими как диски, принтеры и дисплей.

Цели разработки операционной системы

Наличие опытных разработчиков играет важнейшую роль для любого проекта по разработке программного обеспечения. Брукс указывает, что большинство ошибок допускают не при программировании, а на стадии проекта. Программисты правильно делают то, что им велят делать. Но если то, что им велели, было неверно, то никакое тестовое программное обеспечение не сможет поймать ошибку неверно составленной спецификации.

Решение, предложенное Бруксом, заключается в отказе от классической модели разработки (рис. 32, а) и использовании модели, показанной на рис. 32, б. Принцип состоит в том, чтобы сначала написать главный модуль программы, который просто вызывает процедуры верхнего уровня. Вначале эти процедуры представляют собой заглушки. Начиная уже с первого дня система будет транслироваться и запускаться, хотя делать она ничего не будет. Постепенно заглушки заменяются модулями. Результат применения такого метода заключается в том, что сборка системы проверяется постоянно, поэтому ошибки в проекте обнаруживаются значительно раньше. Таким образом, процесс обучения на собственных ошибках также начинается значительно раньше.

Неполные знания опасны. В своей книге Брукс описывает явление, названное им эффектом второй системы. Часто первый продукт, созданный группой разработчиков, является минимальным, так как они опасаются, что он не будет работать вообще. Поэтому они не помещают в первый выпуск программного обеспечения много функций. Если проект оказывается удачным, они создают следующую версию программного обеспечения. Воодушевленные собственным успехом, во второй раз разработчики включают в систему все погремушки и побрякушки, намеренно не включенные в первый выпуск. В результате система раздувается и ее производительность снижается.

От этой неудачи команда разработчиков трезвеет и при выпуске третьей версии снова соблюдает осторожность.

Это наблюдение отчетливо видно на примере пары систем CTSS — MULTICS. Операционная система CTSS была первой универсальной системой разделения времени, и ее успех был огромен, несмотря на минимальную функциональность системы. Создатели операционной системы MULTICS, преемницы CTSS, были слишком амбициозны, за что и поплатились. Сами идеи были неплохи, но новых функций добавилось слишком много, что привело к низкой

производительности системы, страдавшей этим недугом в течение долгих лет и так и не получившей коммерческого успеха. Третьей в этой линейке была операционная система UNIX, разработчики которой проявили значительно большую осторожность и в результате добились существенно больших успехов.

Рисунок 32. Проектирование программного обеспечения: а — традиционное поэтапное; б — альтернативный метод создания работающей уже с первого дня системы