**государственное бюджетное образовательное учреждение
среднего профессионального образования
Владимирской области
«Владимирский авиамеханический колледж»**

МЕТОДИЧЕСКОЕ ПОСОБИЕ
по дисциплине

ОПЕРАЦИОННЫЕ СИСТЕМЫ
(профессиональный цикл)

по специальностям среднего профессионального образования

230113 Компьютерные системы и комплексы
230115 Программирование в компьютерных системах

по программам базовой подготовки

Владимир 2012

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ....4
Введение....5
Раздел 1. основные понятия операционных систем....7
Предмет и задачи курса....7
Краткий очерк истории ОС....7
Классификация операционных систем....12
Критерии оценки ОС....14
Основные функции и структура ОС....16
Операционные системы, используемые в дальнейшем изложении....19
РАЗДЕЛ 2. Управление устройствами....23
Основные задачи управления устройствами....23
Классификация периферийных устройств и их архитектура....23
Прерывания....26
Способы организации ввода/вывода....27
Активное и пассивное ожидание....29
Буферизация и кэширование....30
Драйверы устройств....32
Управление устройствами в MS-DOS....34
Управление устройствами в Windows....35
Управление устройствами в UNIX....35
РАЗДЕЛ 3. Управление данными....36
Основные задачи управления данными....36
Характеристики файлов и архитектура файловых систем....36
Размещение файлов....38
Защита данных....42
Разделение файлов между процессами....43
Файловая система FAT и управление данными в MS-DOS....44
Файловые системы и управление данными в UNIX....47
Файловая система NTFS и управление данными в Windows....52
РАЗДЕЛ 4. Управление процессами....58
Основные задачи управления процессами....58
Понятия процесса и ресурса....58
Квазипараллельное выполнение процессов....59
Состояния процесса....59
Вытесняющая и невытесняющая многозадачность....61
Дескриптор и контекст процесса....62
Дисциплины диспетчеризации и приоритеты процессов....63
Изоляция процессов и их взаимодействие....64
Проблема взаимного исключения процессов....65
Проблема тупиков....66
РАЗДЕЛ 5. Управление памятью....68

Основные задачи управления памятью 68
Виртуальные и физические адреса 68
Распределение памяти без использования виртуальных адресов 69
Сегментная организация памяти 70
Страничная организация памяти 72
СПИСОК ЛИТЕРТУРЫ 75

ПРЕДИСЛОВИЕ

Дисциплина «Операционные системы» является одной из важнейших дисциплин общепрофессионального цикла при подготовке кадров в области информатики, вычислительной техники и компьютерных технологий и имеет своей целью приобретение знаний и навыков, необходимых для профессиональной деятельности будущего специалиста.

Целью изучения дисциплины «Операционные системы» является изучение основ построения операционных систем, общих принципов их построения, выполняемых функций, детальное изучения операционных систем современных ПЭВМ, их команд, приобретение практических навыков работы в среде операционных систем WINDOWS XP, LINUX (UNIX).

Курс базируется на знаниях, полученных при изучении дисциплины «Информатика и ИКТ». Изучение дисциплины «Операционные системы» осуществляется в предметной взаимосвязи с дисциплинами:

- «Основы программирования»;
- «Архитектура компьютерных систем»;
- «Информационные технологии»;
- «Технические средства информатизации».

Данное методическое пособие содержит теоретический материал по основополагающим принципам построения операционных систем. В качестве примера операционных систем изучаются системы MD-DOS, Windows XT и Linux (UNUX). Курс построен на сравнении этих трех операционных систем.

Методическое пособие содержит подробное изложение учебного материла по дисциплине «Операционные системы», что позволит студентам освоить материал самостоятельно, если занятие было пропущено по каким-либо причинам.

Методическое пособие предназначено для студентов специальностей 230113 «Компьютерные системы и комплексы» и 230115 «Программирование в компьютерных системах», изучающих дисциплину «Операционные системы».

ВВЕДЕНИЕ

Современный персональный компьютер – универсальное, многофункциональное автоматическое устройство обработки информации. Сегодня компьютеры принимают на себя основную часть функций по обработке данных.

Программное обеспечение компьютера разделяется на *общесистемное* и *прикладное*.

Операционная система, являясь основой общесистемного программного обеспечения, обеспечивает функционирование и взаимосвязь всех компонентов компьютера и предоставляет пользователю доступ к его аппаратным возможностям.

Прикладное программное обеспечение в свою очередь делится на средства разработки и приложения. *Средства разработки* – это инструменты программиста, включающие алгоритмические языки программирования, а также трансляторы (компиляторы и интерпретаторы). *Приложения* – программные продукты, предназначенные для решения задач в конкретной предметной области.

Операционная система является первичной программой для всякой. При включении электропитания ЭВМ автоматически осуществляется считывание с магнитного носителя, запись в оперативную память и запуск резидентных программ операционной системы (загрузка операционной системы).

Операционные системы предназначены для выполнения следующих основных функций:

- управление устройствами;
- управление данными;
- управление задачами (процессами).
- управление памятью.

Структурно операционная система представляет собой совокупность программ, управляющих ходом работы вычислительной машины.

В данном пособии рассматриваются общие принципы функционирования операционных систем, а также основные алгоритмы и структуры данных, используемые при разработке отдельных подсистем и модулей операционных систем. Описываются подсистемы управления устройствами, данными, процессами и памятью. В качестве примеров операционных систем рассмотрены Windows, UNIX и MS-DOS.

В разделе 1 «Основные понятия операционных систем» рассматриваются краткая история развития операционных систем, их структура и состав.

В разделе 2 «Управление устройствами» рассматриваются основные функции операционной системы по управлению устройствами, организация системы ввода/вывода по прерываниям, назначение драйвера устройств.

В разделе 3 «Управление данными» рассматриваются понятия файла и файловой системы. В качестве примеров файловых систем приводятся системы FAT, NTFS, S5. Рассматриваются организация контроля доступа к хранимым данным.

В разделе 4 «Управление процессами» изучаются понятие процесса, различные дисциплины планирования и диспетчеризации процессов.

В разделе 5 «Управление памятью» рассматриваются методы управления памятью от самых простых схем с фиксированными разделами до современной странично-сегментной организации памяти.

Знания, полученные при изучении дисциплины «Операционные системы», применяются в решении практических задач при работе в системах WINDOWS XP, LINUX (UNIX).

РАЗДЕЛ 1. ОСНОВНЫЕ ПОНЯТИЯ ОПЕРАЦИОННЫХ СИСТЕМ

Предмет и задачи курса

Предметом изучения в данном курсе являются *операционные системы* (ОС) современных компьютеров.

В первом приближении ОС можно определить как комплекс программ, обеспечивающих интерфейс между аппаратурой компьютера, прикладными программами и пользователем компьютера. Соответственно этому определению, все функции, выполняемые ОС, подчинены решению двух основных задач:

- организации эффективной работы аппаратуры компьютера;
- обеспечению удобного использования ресурсов компьютера как прикладными программами, так и пользователем, работающим с компьютером.

Основной целью курса является изучение устройства и функционирования современных ОС. При этом будут рассматриваться два круга вопросов:

- основные принципы построения ОС, наиболее распространенные алгоритмы выполнения различных функций ОС, типовые структуры данных, используемые для обеспечения работы ОС;
- практическое воплощение этих принципов, алгоритмов, структур в наиболее распространенных современных ОС.

Краткий очерк истории ОС

Изучение истории развития ОС показывает, что все существенные продвижения в области архитектуры ОС связаны с влиянием двух основных факторов:

- прогресс технологии, приводящий к быстрому возрастанию характеристик аппаратуры ЭВМ и к появлению принципиально новых типов аппаратуры;
- принципиально новые идеи, возникающие у проектировщиков.

Предыстория ОС

Вскоре после того, как в конце 40-х годов XX века были созданы первые электронные компьютеры, очень остро встала проблема повышения эффективности использования оборудования, и прежде всего центрального процессора.

Типичный компьютер первого – второго поколений представлял собой большую комнату, уставленную шкафами и увитую кабелями. Каждое из основных устройств – центральный процессор, оперативная память, накопители на магнитных лентах, устройства

ввода с перфокарт, принтер – занимало один или несколько «шкафов» или «тумб», наполненных радиолампами и механическими частями.

Все это стоило больших денег, потребляло бешеное количество электроэнергии и регулярно ломалось.

В таких условиях машинное время стоило очень дорого. Тем не менее, обычная практика использования ЭВМ не способствовала экономии. Как правило, программист, разрабатывающий программу, заказывал ежедневно несколько часов машинного времени и в течение этого времени монопольно использовал машину. Выполнив очередной запуск отлаживаемой программы (которую надо было каждый раз вводить либо с клавиатуры, либо, в лучшем случае, с перфокарт), пользователь получал распечатку (чаще всего в виде массива цифр), анализировал результаты, вносил изменения в программу и снова запускал ее. Таким образом, в ходе сеанса отладки дорогостоящее оборудование простаивало 99% времени, пока программист осмысливал результаты и работал с устройствами ввода/вывода. Кроме того, сбой при вводе одной перфокарты мог потребовать начать сначала всю работу программы.

Возникла великая идея – использовать сам компьютер для повышения эффективности работы с ним же.

Одно из ответвлений этой идеи – создание языков и систем программирования – рассматривается в отдельных курсах. Другим важным шагом стало возложение на специальную компьютерную программу части тех функций, которые до этого выполнял оператор или сам программист.

Программы такого рода назывались обычно ***мониторами*** (не путать с монитором как устройством вывода, который в то время был редчайшей экзотикой!). Монитор принимал команды, состоящие, как правило, из 1-2 букв названия и 1-3 аргументов, заданных 8-ричными или 16-ричными числами. Типичными командами были, например:

- загрузка данных с перфокарт по указанному адресу памяти;
- просмотр и корректировка (с пишущей машинки) значений в указанном диапазоне адресов;
- пошаговое выполнение программы с выдачей результатов каждой команды на пишущую машинку;
- запуск программы с указанного адреса с заданием адресов контрольных точек остановки.

Несмотря на убогость, по нынешним меркам, подобных средств, они в свое время значительно повысили производительность работы программистов. Однако кардинального повышения загрузки процессора не произошло.

Временем широкого распространения мониторов в мире были 50-е годы прошлого века (в СССР – 60-е годы). В настоящее время нечто подобное можно встретить на самых примитивных микропроцессорных контроллерах.

Пакетные ОС

Историю собственно ОС можно начать с появления в конце 50-х годов первых систем, организующих работу по пакетному принципу.

Важнейшим организационным изменением, происшедшим на этом этапе развития, стало массовое изгнание программистов из машинных залов, как фактора, лишь вносящего сумятицу в работу.

Теперь от программиста требовалось собрать пакет перфокарт, содержащий его программу, данные к ней, а также управляющие перфокарты. Эти карты на специально разработанном *языке управления заданиями* (JCL, Job Control Language) объясняли операционной системе, чье это задание, что нужно сделать с программой (например, передать ее транслятору с Фортрана), что предпринять в случае успешной трансляции (вероятно, пустить на решение), что – при наличии ошибок (например, перейти к другой программе), откуда взять исходные данные (например, с такого-то цилиндра магнитного диска). Кроме того, там могли быть даже указания на то, сколько метров бумаги можно выделить на распечатку и какое максимальное время может занять работа программы.

Обойтись без столь подробных инструкций было нельзя, потому что программист не присутствовал при запуске задания и не мог вмешаться лично.

Подготовленный пакет передавался, вместе с другими подобными пакетами, оператору ЭВМ, перед которым стояли две основные задачи: чтобы в устройстве ввода не переводились пакеты заданий и чтобы в принтере не кончилась бумага. Когда процессор заканчивал обработку задания и печать его результатов, он вводил следующий пакет и приступал к его обработке. Так достигалась основная цель пакетного режима – исключить простои процессора из-за нерасторопности людей.

В скором времени разработчики ОС осознали, что вычерпаны далеко не все резервы повышения загрузки процессора. Операции ввода и печати требовали лишь очень небольшой доли от полной производительности процессора. Кроме того, в ходе работы программы случались обращения к периферийным устройствам (например, к магнитным лентам и,

позднее, дискам), при выполнении которых процессор опять простаивал. Целесообразно было найти способ, чтобы в эти периоды ожидания загрузить процессор другой работой. Но для этого необходимо, чтобы в памяти процессора находились сразу несколько программ, тогда ОС смогла бы переключать процессор на выполнение той программы, которая в данный момент может работать.

Такая организация работы, когда в памяти находятся несколько программ и система в определенные моменты переключает выполнение с одной программы на другую, была названа *мультипрограммированием.* Эта важная идея в разных воплощениях пережила те пакетные системы, в которых она впервые была реализована, и является основой для функционирования практически всех современных ОС.

Среди наиболее развитых пакетных ОС с мультипрограммированием нельзя не назвать OS/360, основную ОС знаменитого в 60-70 гг. семейства ЭВМ IBM 360/370.

ОС с разделением времени

На рубеже 60-70 гг. распространенным и не слишком дорогим периферийным устройством становятся мониторы (сначала монохромные и работающие только в текстовом режиме). При этом процессор и ОЗУ остаются самыми дорогими и громоздкими устройствами вычислительной системы. В этих условиях возникает и быстро приобретает популярность принципиально новый тип ОС – *системы с разделением времени*.

К одной ЭВМ подключается несколько десятков рабочих мест, оборудованных дисплеем (монитор + клавиатура) и совместно использующих вычислительные ресурсы ЭВМ. Процессорное время делится на кванты длительностью в несколько десятков миллисекунд и по истечении каждого кванта процессор может быть переключен на обслуживание другого процесса, другого дисплея. Поскольку теперь подготовку текстов программ выполняют сами программисты за дисплеями, а работа по редактированию текста требует очень малых затрат процессорного времени, процессор успевает обслужить все рабочие места практически без ощутимой задержки. Большая часть времени процессора уделяется небольшому числу рабочих мест, где в данный момент запущены на выполнение программы. При этом, разумеется, средняя скорость работы каждой программы уменьшается, по крайней мере во столько раз, сколько программ выполняется одновременно.

Режим разделения времени стал огромным облегчением для программистов, которые вновь смогли в некоторой степени почувствовать себя «хозяевами» ЭВМ и получили возможность запускать программы на трансляцию и отладку хоть каждые 5 минут. Это позволило сократить сроки разработки и отладки программ.

Для трудоемких вычислительных заданий, предусматривающих счет по ранее отлаженным программам, режим разделения времени менее эффективен, чем пакетный, поскольку частое переключение процессора между выполняемыми программами требует дополнительных затрат времени.

Первоначально в качестве аппаратной основы систем разделения времени должны были использоваться «большие» ЭВМ, которые позднее стало принято называть «мейнфреймами» (mainframes). Позднее, по мере прогресса вычислительной техники, это стало по плечу даже миниЭВМ (так назывался в те годы класс компьютеров, занимавших всего лишь один-два небольших шкафчика). Следует особо упомянуть серию миниЭВМ PDP-11, имевшую широчайшее распространение во всем мире в течение полутора десятков лет.

Этот период (70-е годы в мире, 80-е в СССР) характерен глубоким развитием теории и практики создания мощных ОС, содержащих развитые средства управления процессами и памятью, реализующих многопользовательский режим работы. Из большого числа подобных систем особого упоминания заслуживает UNIX – единственная система, благополучно дожившая до нашего времени.

Однозадачные ОС для ПЭВМ

В середине 70-х годов был изобретен микропроцессор, а к началу 80-х микропроцессоры стали догонять по функциональным характеристикам ранее использовавшиеся «большие» процессоры. Эта ситуация сделала почти бесполезным режим разделения времени: зачем делить один процессор между многими задачами и многими пользователями, если проще и дешевле дать отдельный микропроцессор каждому пользователю? Разделение времени осталось целесообразным разве что в отношении суперкомпьютеров.

Появление и бурное распространение персональных компьютеров (ПК) вызвало к жизни новое поколение ОС, которые оказались во много раз проще своих предшественниц. Ненужной оказалась многопользовательская защита. На первых порах показалась ненужной и многозадачность. Все это можно было расценить как явный регресс в развитии ОС.

Наиболее популярной ОС для ранних восьмиразрядных ПК была система CP/M известной тогда фирмы Digital Research, однако с появлением в начале 80-х знаменитой машины IBM PC лидерство было прочно перехвачено системой MS-DOS фирмы Microsoft.

Многозадачные ОС для ПК с графическим интерфейсом

Быстрое развитие технологии привело к тому, что к концу 80-х годов ПК оказались в состоянии решать значительно более сложные и трудоемкие задачи, чем раньше. При этом

многие из достижений прежних этапов развития ОС оказались вновь востребованными, но теперь уже в новых условиях, среди которых надо назвать резкое повышение мощности процессоров и объема памяти, появление высококачественных графических мониторов и развитие сетевых технологий.

На смену ОС, которые выполняли текстовые команды, вводимые пользователем с клавиатуры, пришли системы, в которых взаимодействие с пользователем основано на использовании GUI (Graphical User Interface, графический интерфейс пользователя).

Значительная часть ПК работает в составе локальных вычислительных сетей. Это привело к тому, что вопросы защиты данных пользователя вновь приобрели первостепенное значение.

Классификация операционных систем

Существуют различные виды классификации ОС по тем или иным признакам, отражающие разные существенные характеристики систем.

- По назначению.

– Системы общего назначения. Это ОС, предназначенные для решения широкого круга задач, включая запуск различных приложений, разработку и отладку программ, работу с сетью и с мультимедиа.

– *Системы реального времени*. Этот класс систем предназначен для работы для управления объектами (такими, как летательные аппараты, технологические установки, автомобили, сложная бытовая техника и т.п.). Из подобного назначения вытекают жесткие требования к надежности и эффективности системы. Должно быть обеспечено точное планирование действий системы во времени (управляющие сигналы должны выдаваться в заданные моменты времени, а не просто «по возможности быстро»). Особый подкласс составляют системы, *встроенные* в оборудование. Такие системы годами могут выполнять фиксированный набор программ, не требуя вмешательства человека-оператора на более глубоком уровне, чем нажатие кнопки «Вкл.».

– Прочие специализированные системы. Это различные ОС, ориентированные прежде всего на эффективное решение задач определенного класса, с большим или меньшим ущербом для прочих задач. Можно выделить, например, сетевые системы (такие, как Novell Netware), обеспечивающие надежное и высокоэффективное функционирование локальных сетей.

- По характеру взаимодействия с пользователем.

– Пакетные ОС, обрабатывающие заранее подготовленные задания.

– Диалоговые ОС, выполняющие команды пользователя в интерактивном режиме. Красивое слово «интерактивный» означает постоянное взаимодействие системы с пользователем.

– ОС с графическим интерфейсом. В принципе, их также можно отнести к диалоговым системам, однако использование мыши и всего, что с ней связано (меню, кнопки и т.п.) вносит свою специфику.

– Встроенные ОС, не взаимодействующие с пользователем.

- По числу одновременно выполняемых задач.

– Однозадачные ОС. В таких системах в каждый момент времени может существовать не более чем один активный пользовательский процесс. Следует заметить, что одновременно с ним могут работать системные процессы (например, выполняющие запросы на ввод/вывод).

– Многозадачные ОС. Они обеспечивают параллельное выполнение нескольких пользовательских процессов. Реализация многозадачности требует значительного усложнения алгоритмов и структур данных, используемых в системе.

- По числу пользователей.

– Однопользовательские ОС. Для них характерен полный доступ пользователя к ресурсам системы. Подобные системы приемлемы в основном для изолированных компьютеров, не допускающих доступа к ресурсам данного компьютера по сети или с удаленных терминалов.

– Многопользовательские ОС. Их важной компонентой являются средства защиты данных и процессов каждого пользователя, основанные на понятии владельца ресурса и на точном указании прав доступа, предоставленных каждому пользователю системы.

- По аппаратурной основе.

– Однопроцессорные ОС. В данном курсе будут рассматриваться только они.

– Многопроцессорные ОС. В задачи такой системы входит, помимо прочего, эффективное распределение выполняемых заданий по процессорам и организация согласованной работы всех процессоров.

– Сетевые ОС. Они включают возможность доступа к другим компьютерам локальной сети, работы с файловыми и другими серверами.

– Распределенные ОС. Их отличие от сетевых заключается в том, что распределенная система, используя ресурсы локальной сети, представляет их пользователю как единую систему, не разделенную на отдельные машины.

Критерии оценки ОС

Надежность

Этот критерий вообще принято считать самым важным при оценке программного обеспечения, и в отношении ОС его действительно принимают во внимание в первую очередь.

Что понимают под надежностью?

Во-первых, ее *живучесть*, т.е. способность сохранять хотя бы минимальную работоспособность в условиях аппаратных сбоев и программных ошибок. Высокая живучесть особенно важна для ОС компьютеров, встроенных в аппаратуру, когда вмешательство человека затруднено, а отказ компьютерной системы может иметь тяжелые последствия.

Во-вторых, способность, как минимум, диагностировать, а как максимум, компенсировать хотя бы некоторые типы аппаратных сбоев. Для этого обычно вводится избыточность хранения наиболее важных данных системы.

В-третьих, ОС не должна содержать собственных (внутренних) ошибок. Это требование редко бывает выполнимо в полном объеме (программисты давно сумели доказать своим заказчикам, что в любой большой программе всегда есть ошибки, и это в порядке вещей), однако следует хотя бы добиться, чтобы основные, часто используемые или наиболее ответственные части ОС были свободны от ошибок.

Наконец, к надежности системы следует отнести ее способность противодействовать явно неразумным действиям пользователя. Обычный пользователь должен иметь доступ только к тем возможностям системы, которые необходимы для его работы. Если же пользователь, даже действуя в рамках своих полномочий, пытается сделать что-то очень странное (например, отформатировать системный диск), то самое малое, что должна сделать ОС, это переспросить пользователя, уверен ли он в правильности своих действий.

Эффективность

Как известно, эффективность любой программы определяется двумя группами показателей, которые можно обобщенно назвать «время» и «память». При разработке

системы приходится принимать много непростых решений, связанных с оптимальным балансом этих показателей.

Важнейшим показателем временно́й эффективности является *производительность* системы, т.е. усредненное количество полезной вычислительной работы, выполняемой в единицу времени. С другой стороны, для диалоговых ОС не менее важно *время реакции* системы на действия пользователя. Эти показатели могут в некоторой степени противоречить друг другу. Например, в системах разделения времени увеличение кванта времени повышает производительность (за счет сокращения числа переключений процессов), но ухудшает время реакции.

В программировании известна аксиома: выигрыш во времени достигается за счет проигрыша в памяти, и наоборот. Это в полной мере относится к ОС, разработчикам которых постоянно приходится искать баланс между затратами времени и памяти.

Удобство

Этот критерий наиболее субъективен. Можно предложить, например, такой подход: система или ее часть удобна, если она позволяет легко и просто решать те задачи, которые встречаются наиболее часто, но в то же время содержит средства для решения широкого круга менее стандартных задач (пусть даже эти средства не столь просты). Пример: такое частое действие, как копирование файла, должно выполняться при помощи одной простой команды или легкого движения мыши; в то же время для изменения разделов диска не грех почитать руководство, поскольку это может понадобиться даже не каждый год.

Масштабируемость

Термин «масштабируемость» (scalability) означает возможность настройки системы для использования в разных вариантах, в зависимости от мощности вычислительной системы, от набора конкретных периферийных устройств, от роли, которую играет конкретный компьютер (сервер, рабочая станция или изолированный компьютер) от назначения компьютера (домашний, офисный, исследовательский и т.п.).

Гарантией масштабируемости служит продуманная модульная структура системы, позволяющая в ходе установки системы собирать и настраивать нужную конфигурацию. Возможен и другой подход, когда под общим названием объединяются, по сути, разные системы, обеспечивающие в разумных пределах программную совместимость. Примером могут служить версии Windows NT/2000/XP, Windows 95/98 и Windows CE.

Способность к развитию

Чтобы сложная программа имела шансы просуществовать долго, в нее изначально должны быть заложены возможности для будущего развития.

Одним из главных условий способности системы к развитию является хорошо продуманная модульная структура, в которой четко определены функции каждого модуля и его взаимосвязи с другими модулями. При этом создается возможность совершенствования отдельных модулей с минимальным риском вызвать нежелательные последствия для других частей системы.

Важным требованием к развитию ОС является ***совместимость версий снизу вверх***, означающая возможность безболезненного перехода от старой версии к новой, без потери ранее наработанных прикладных программ и без необходимости резкой смены всех навыков пользователя. Обратная совместимость – сверху вниз – как правило, не гарантируется, поскольку в ходе развития система приобретает новые возможности, не реализованные в старых версиях. Программа из Windows 3.1 будет нормально работать и в Windows XP; наоборот – вряд ли.

Мобильность

Под *мобильностью* (portability) понимается возможность переноса программы (в данном случае ОС) на другую аппаратную платформу, т.е. на другой тип процессора и другую архитектуру компьютера. Здесь имеется в виду перенос с умеренными трудозатратами, не требующий полной переработки системы.

Свойство мобильности не столь однозначно положительно, как может показаться. Чтобы программа была мобильна, при ее разработке следует отказаться от глубокого использования особенностей конкретной архитектуры (таких, как количество и функциональные возможности регистров процессора, нестандартные команды и т.п.). Мобильная программа должна быть написана на языке достаточно высокого уровня (часто используется язык C), который можно реализовать на компьютерах любой архитектуры. Платой за мобильность всегда является некоторая потеря эффективности, поэтому немобильные системы распространены достаточно широко.

Основные функции и структура ОС

При рассмотрении основ функционирования ОС принято выделять четыре основных группы функций, выполняемых системой.

- *Управление устройствами*. Имеются в виду все периферийные устройства, подключаемые к компьютеру, – клавиатура, монитор, принтеры, диски и т.п.

- *Управление данными*. Под этим старинным термином сейчас понимается работа с файлами, хотя были времена, когда обращение к данным на магнитных носителях

выполнялось путем указания адреса размещения данных на устройстве, а понятия файла не существовало.

- *Управление процессами*. Эта сторона работы ОС связана с запуском и завершением работы программ, обработкой ошибок, обеспечением параллельной работы нескольких программ на одном компьютере.

- *Управление памятью*. Оперативная память компьютера – это такой ресурс, которого всегда не хватает. В этих условиях разумное планирование использования памяти является важнейшим фактором эффективной работы.

Для понимания работы ОС необходимо уметь выделять основные части системы и их связи, т.е. описывать структуру системы. Для разных ОС их структурное деление может быть весьма различным. Наиболее общими видами структуризации можно считать два. С одной стороны, можно считать, что ОС разделена на подсистемы, соответствующие перечисленным выше группам функций. Такое деление достаточно обосновано, программные модули ОС действительно в основном можно отнести к одной из этих подсистем. Другое важное структурное деление связано с понятием ***ядра*** системы.

Ядро, как можно понять из названия, это основная, «самая системная» часть операционной системы. Имеются разные определения ядра. Согласно одному из них, ядро – это *резидентная* часть системы, т.е. к ядру относится тот программный код, который постоянно находится в памяти в течение всей работы системы. Остальные модули ОС являются *транзитными*, т.е. подгружаются в память с диска по мере необходимости на время своей работы. К транзитным частям системы относятся:

- *утилиты* (utilities) – отдельные системные программы, решающие частные задачи, такие как форматирование и проверку диска, поиск данных в файлах, мониторинг (отслеживание) работы системы и многое другое;

- *системные библиотеки подпрограмм*, позволяющие прикладным программам использовать различные специальные возможности, поддерживаемые системой (например, библиотеки для графического вывода, для работы с мультимедиа и т.п.);

- *интерпретатор команд* – программа, выполняющая ввод команд пользователя, их анализ и вызов других модулей для выполнения команд;

- *системный загрузчик* – программа, которая при запуске ОС (например, при включении питания) обеспечивает загрузку системы с диска, ее инициализацию и старт;

- другие виды программ, в зависимости от конкретной системы.

Не менее важным является определение ядра, основанное на различении режимов работы компьютера. Все современные процессоры поддерживают, как минимум, два режима: *привилегированный* режим (он же режим ядра, kernel mode) и *непривилегированный* (режим задачи, режим пользователя, user mode). Программы, работающие в режиме ядра, имеют полный, неограниченный доступ ко всем ресурсам компьютера: его командам, адресам, портам ввода/вывода и т.п. В режиме задачи возможности программы ограничены, она, в частности, не может выполнить некоторые специальные команды. Аппаратное разграничение возможностей является абсолютно необходимым условием реализации надежной защиты данных в многопользовательской системе. Отсюда вытекает и определение ядра как части ОС, работающей в режиме ядра. Все остальные программы, как системные утилиты, так и программы пользователей, работают в режиме пользователя и должны обращаться к ядру для выполнения многих системных действий.

Следует сказать, что переходы из режима пользователя в режим ядра и обратно – это действия, требующие определенного времени, и слишком частое их выполнение может привести к заметному снижению скорости работы программ. В связи с этим определение того, какие функции должны поддерживаться ядром, а какие лучше выполнять в режиме пользователя – это непростая и важная задача, которую должны решить разработчики ОС.

В качестве программного интерфейса системы, т.е. средств для обращения прикладных программ к услугам ОС, используется документированный набор *системных вызовов* или *функций API* (Applied Programming Interface). Между этими двумя терминами есть некоторая разница. Под системными вызовами понимаются функции, реализуемые непосредственно программами ядра системы. При их выполнении происходит переход из режима пользователя в режим ядра, а затем обратно. В отличие от этого, API-функции определяются как функции, описанные в документации ОС, независимо от того, выполняются ли они ядром или же системными библиотеками, работающими в режиме пользователя. В Windows часто несколько разных API-функций обращаются к одному и тому же недокументированному системному вызову, но имеют различные обрамляющие части, работающие в режиме пользователя.

Там, где различие между двумя этими понятиями несущественно, можно использовать нейтральный термин «*системные функции*».

Операционные системы, используемые в дальнейшем изложении

В следующих разделах курса будут рассматриваться основные функции ОС и способы их реализации. Изложение общих подходов будет дополняться примерами, относящимися главным образом к трем широко известным ОС:

- MS-DOS – пример простой однозадачной системы;
- Windows – сложная современная система, выросшая на базе MS-DOS;
- UNIX – система, по возможностям сопоставимая с Windows, однако разительно отличающая по набору основных концепций и методам реализации.

MS-DOS

Система MS-DOS была разработана в 1981 г. специально для только что появившейся первой 16-разрядной ПЭВМ IBM PC на базе процессора i86. Первая версия системы была ужасна, но работоспособна. В последующие годы фирме Microsoft удалось значительно улучшить свою систему, хотя некоторые пережитки первой версии оказались неистребимы. Альянс с фирмой IBM позволил Microsoft добиться фантастического финансового успеха.

MS-DOS представляет собой однозадачную, однопользовательскую, диалоговую ОС. Она ведет диалог с пользователем в текстовом режиме и в большей степени рассчитана на обслуживание прикладных программ текстового режима, хотя допускает и графику. Работа с мышью должна обеспечиваться самими прикладными программами при минимальной поддержке со стороны ОС. Для размещения программы пользователя и для своих собственных нужд MS-DOS позволяет использовать 640 Кбайт памяти, что казалось огромной величиной в те незапамятные времена аккуратного программирования и полного отсутствия файлов AVI и MP3. Позднее были добавлены средства, позволяющие с некоторым усилием использовать до 4 Мб памяти.

Интерфейс MS-DOS с прикладными программами основан на вызовах программных прерываний, обрабатываемых системой. Бóльшую часть этих прерываний принято называть *функциями DOS*.

Система MS-DOS явилась стартовой площадкой для создания Windows. В настоящее время MS-DOS тихо отмирает, хотя все версии Windows стараются обеспечить выполнение большей части программ, разработанных для их предшественницы.

В данном курсе MS-DOS рассматривается как наиболее жизненный пример простой и хорошо изученной однозадачной системы для сравнения с более мощными многозадачными системами.

Windows

Система Windows была первоначально разработана фирмой Microsoft как графическая оболочка, загружаемая поверх MS-DOS. Идеи GUI (Graphic User Interface – графический интерфейс пользователя) были впервые разработаны для экспериментальной машины Xerox PARC еще в 70-х гг., затем подхвачены в MacOS – операционной системе компьютера Macintosh, откуда и были с некоторыми ухудшениями позаимствованы в Windows. Версию Windows 1.0, вышедшую в 1985 г. и работавшую на 1 Мб памяти с неперекрывающимися окнами, принято рассматривать как интересную игрушку. Версия 2.0 (1987 г.) была более серьезна, а версии 3.0 и 3.1 (1990-1992 гг.), предназначенные для процессоров i386 и использующие до 16 Мб памяти, уже имели большой успех.

Все перечисленные версии продолжали оставаться надстройками над MS-DOS, использующими имеющуюся файловую систему, но добавляющие свое собственное управление процессами, памятью и устройствами. За счет этого комбинацию DOS + Windows можно было назвать многозадачной однопользовательской ОС с графическим интерфейсом пользователя.

В 1993 г. Microsoft выпустила Windows NT – полноценную многозадачную и многопользовательскую ОС, уже не основанную на MS-DOS. Однако, поскольку NT предъявляла повышенные требования к мощности процессора и объему памяти, в 1995 г. была выпущена компромиссная система Windows 95, предназначавшаяся для замены Windows 3.x у массового пользователя. Повышение скорости работы по сравнению с версией NT было достигнуто ценой отказа от многопользовательской защиты и ослабления надежности системы. В Windows 95 неаккуратно написанная прикладная программа может привести к краху системы, а в Windows NT система лучше изолирована от программ пользователя. В то же время, практически все корректно написанные программы могут переноситься из Windows 95 в Windows NT и наоборот.

Некоторое время две линии Windows развивались параллельно. Очередные версии Windows NT получили название Windows 2000, Windows XP, Windows 2003. Линия Windows 95 была продолжена непринципиально отличающимися от нее версиями Windows 98 и Windows ME, но дальше, видимо, развиваться не будет. Microsoft считает, что современный уровень производительности ПЭВМ снимает необходимость в облегченной версии системы.

Windows предоставляет в распоряжение прикладных программ несколько тысяч документированных API-функций на все случаи жизни.

В дальнейшем изложении описание возможностей Windows будет в основном ориентировано на линию Windows NT/2000/XP.

UNIX

ОС UNIX была первоначально разработана в 1969 г. сотрудниками фирмы Bell Laboratories Кеном Томпсоном и Деннисом Ритчи. В 1971 г. система была перенесена на машины чрезвычайно распространенной в 70-е годы серии PDP-11, а в 1973 г. Ритчи переписал систему на языке С, оставив лишь минимум текста на языке ассемблера. В первое десятилетие существования UNIX и сама система, и ее исходные тексты распространялись свободно, что привело к чрезвычайной популярности системы в научных кругах и университетах. Усовершенствования системы могли вноситься каждым желающим и обсуждались «всем миром». Оборотной стороной такой открытости стала трудность стандартизации UNIX. Однако в 1988-1990 гг. был разработан набор стандартов, получивший название POSIX (Portable OS, а окончание IX – как намек на UNIX). Эти стандарты фиксировали современные требования к системам типа UNIX с учетом теоретических и практических достижений за прошедшие годы.

Начиная с первых версий, UNIX представляет собой многозадачную, многопользовательскую систему разделения времени. Основными достоинствами UNIX являются ее высокая мобильность, хорошо продуманный программный и пользовательский интерфейс.

К недостаткам UNIX можно отнести более низкую эффективность и надежность работы, что в значительной мере является платой за мобильность. Традиционная модель безопасности UNIX не соответствует современным требованиям, поэтому в различные коммерческие версии приходится включать дополнительные средства защиты данных. Широкому распространению UNIX мешает также то, что процедуры установки и настройки системы не так просты, как у Windows, и при их выполнении желательно участие программиста.

В 80-е годы были попытки превратить UNIX в коммерческую систему. Однако в 1991-1994 гг. Линус Торвальдс, в то время студент-программист из Хельсинки, заново написал систему, соответствующую стандартам POSIX, но отличающуюся от традиционной UNIX большей надежностью и эффективностью. Эта система получила название Linux. Исходные тексты Linux свободно распространяются, что позволяет, как во времена молодости UNIX, развивать систему общими усилиями огромного сообщества заинтересованных программистов. Эффективной координации этих усилий очень способствует Интернет. Несколько позднее был открыт свободный доступ к текстам известной версии UNIX FreeBSD.

Архитектура UNIX, первоначально предназначенная для систем разделения времени с одним процессором, впоследствии оказалась вполне подходящей для поддержки сетевых систем. Значительная часть серверов Интернета работает под управлением той или иной версии UNIX.

В настоящее время происходит ощутимое сближение разных типов ОС, предназначенных для поддержки одних и тех же типов вычислительных систем. Современные версии UNIX и Windows предоставляют весьма близкие функциональные возможности, хотя зачастую в совершенно разной форме. Выравниваются также характеристики надежности и производительности систем.

Существенным отличием UNIX от Windows остается место, занимаемое в системе средствами графического интерфейса. Если в Windows окна и все, что с ними связано, являются неотъемлемой частью архитектуры системы, то для UNIX по традиции основным средством интерфейса с пользователем является текстовая консоль. Те или иные средства оконного интерфейса, конечно, присутствуют в современных UNIX-системах, но как дополнительная, необязательная надстройка скорее прикладного, чем системного характера.

Очень интересной особенностью UNIX является развитый язык команд ***shell***, который позволяет не только вести элементарный диалог с системой, но и писать своеобразные программы (скрипты), с помощью которых часто удается решить требуемую задачу, не прибегая к разработке новой программы на одном из традиционных языков программирования.

РАЗДЕЛ 2. УПРАВЛЕНИЕ УСТРОЙСТВАМИ

Основные задачи управления устройствами

Понятие периферийных устройств (ПУ) объединяет, по сути, все основные аппаратурные блоки компьютера, за исключением процессора и основной памяти.

Важнейшими задачами любой ОС являются обеспечение надежной работы ПУ, эффективное использование всех возможностей устройств.

Для повышения надежности процедуры выполнения операций с устройствами должны обеспечивать, как минимум, выявление аппаратных ошибок и сбоев, а как максимум – их компенсацию за счет избыточности данных и повторного выполнения операций.

Эффективность использования устройств означает прежде всего сокращение времени, затрачиваемого на обмен данными. Помимо повышения скорости обмена, сокращение времени может достигаться за счет распараллеливания работы ПУ и процессора. Мощным фактором повышения производительности системы является сокращение количества операций ввода/вывода за счет сохранения данных в памяти для последующего использования.

Большое разнообразие используемых устройств и постоянное появление новых моделей диктуют необходимость такой структуры системы, которая позволяла бы легкое подключение новых устройств. Широкое распространение получает технология «Plug & Play», т.е. возможность оперативного подсоединения устройств без выключения компьютера.

При всем разнообразии ОС должна обеспечивать максимально возможную стандартизацию работы с ними, чтобы изменения аппаратуры не приводили к необходимости постоянно модифицировать прикладное программное обеспечение.

К числу дополнительных задач, решаемых подсистемами управления устройствами современных ОС, можно отнести хранение данных в сжатом виде, шифрование данных и т.п.

Классификация периферийных устройств и их архитектура

Под *программной архитектурой* (или просто – *архитектурой*) устройства мы будем понимать совокупность тех структурных особенностей, которые влияют на работу программ с устройством. Например, форма разъема для подключения устройства не входит в его архитектуру, но количество и назначение линий в этом разъеме может в нее входить (если эти линии могут программно управляться).

Как правило, вместе с устройством поставляется его *контроллер* (адаптер), содержащий электронные схемы управления устройством. Конструктивно контроллер может представлять собой плату, вставляемую в разъем шины компьютера, либо может быть расположен в корпусе устройства. В любом случае программы работают с устройством через посредство его контроллера, а поэтому с точки зрения архитектуры нет различия между понятиями «устройство» и «контроллер устройства».

Классификация периферийных устройств может быть выполнена по различным признакам.

- Устройства *последовательного доступа* (sequential access) и устройства *произвольного доступа* (random access). Для последовательных устройств характерно наличие определенного естественного порядка данных, при этом обработка данных в ином порядке либо невозможна, либо крайне затруднена. Классическим примером являются магнитные ленты, для которых чтение и запись данных ведутся от начала ленты к концу, а попытка доступа в ином порядке потребует постоянной перемотки ленты, резко снижающей скорость работы. К устройствам последовательного доступа можно отнести также клавиатуру, мышь, принтер, модем.

Для устройств произвольного доступа возможно обращение к различным порциям данных в любом порядке, причем эффективность работы не зависит (или слабо зависит) от порядка обращения. Для таких устройств характерно наличие адресации данных и операции поиска нужного адреса. Наиболее известный пример – магнитные диски и другие дисковые устройства. Кроме того, к устройствам произвольного доступа можно отнести монитор ПК (там есть адресация точек-пикселов, хотя операция поиска не нужна).

- *Символьные* (*байтовые*) и *блочные* устройства. Для символьных устройств наименьшей порцией вводимых и выводимых данных является один байт. Для некоторых символьных устройств можно за одну операцию выполнить ввод или вывод любого (в разумных пределах) требуемого количества байт.

Для блочных устройств наименьшей порцией ввода/вывода, выполняемого за одно обращение к устройству, является один блок, равный, как правило, 2^k байт. Типичным размером блока может быть 512 байт, 1K байт, 4K байт и т.п., в зависимости от конкретного устройства. Наиболее известные примеры блочных устройств – магнитные диски и магнитные ленты. Для диска понятие блока обычно совпадает с понятием сектора. В частности, для IBM-совместимых ПК сектор (блок) диска равен 512 байт.

- *Физические*, *логические* и *виртуальные* устройства. Под физическим устройством обычно понимается некоторый реально существующий прибор, «железка». На самом деле, с точки зрения программной архитектуры для наличия физического устройства достаточно знать набор адресов, команд, прерываний и других сигналов, позволяющих выполнять операции с данными. Куда идут или откуда приходят эти сигналы – это вопрос, не касающийся программиста.

Логическое устройство – это понятие, характеризующее специальное назначение устройства в данной ОС. Например, «загрузочный диск» (т.е. тот, с которого была выполнена загрузка ОС). Наиболее важными логическими устройствами во многих ОС являются *устройство стандартного ввода* и *устройство стандартного вывода*. Их можно упрощенно определить как устройства, используемые для ввода и, соответственно, вывода «по умолчанию», т.е. когда в программе явно не указано другое устройство или файл для ввода/вывода. Как правило, для современных компьютеров устройству стандартного ввода соответствует физическое устройство – клавиатура, а устройству стандартного вывода – монитор. Важно, однако, понимать, что это соответствие может быть изменено: стандартный вывод может быть переназначен, например, на принтер или в файл, стандартный ввод – на удаленный терминал, на файл и т.п.

Понятие «виртуальный» в программировании, вообще говоря, означает примерно следующее: «нечто, на самом деле не существующее, но ведущее себя так, как если бы оно существовало». С этой точки зрения, виртуальное устройство – это программно реализованный объект, который ведет себя подобно некоторому физическому устройству, хотя на самом деле использует ресурсы совсем других устройств (или даже никаких устройств). Примеры виртуальных устройств весьма разнообразны:

- виртуальные диски, расположенные на самом деле в оперативной памяти (такие устройства были популярны в конце 80-х годов);
- виртуальная память, расположенная на самом деле на диске;
- виртуальные CD и DVD – программы, имитирующие поведение соответствующих устройств;
- виртуальный экран, предоставляемый DOS-программе, работающей в режиме окна Windows (программа работает так, как если бы ей был предоставлен весь экран, но на самом деле система направляет вывод программы в отведенное ей окно).

Прерывания

Прерывания (аппаратные) – это сигналы, при поступлении которых нормальная последовательность выполнения программы может быть прервана, при этом система запоминает информацию, необходимую для возобновления работы прерванной программы, и передает управление *подпрограмме обработки прерывания* (ISR, Interrupt Service Routine). По завершению обработки, как правило, управление возвращается прерванной программе.

Все прерывания можно разделить на три основных типа:

- аппаратные прерывания от периферийных устройств;
- внутренние аппаратные прерывания (называемые также *исключениями*, exceptions);
- программные прерывания.

В подавляющем большинстве ОС обработку всех прерываний берет на себя сама система. Поскольку типы и разновидности прерываний весьма многообразны и каждый из них требует особой обработки, большинство процессоров поддерживает *векторные прерывания*. Это означает, что каждая разновидность прерывания имеет свой номер, и этот номер используется как индекс в массиве, хранящем адреса ISR для всех прерываний. При возникновении прерывания аппаратура компьютера по номеру прерывания определяет адрес подпрограммы обработки и вызывает ее.

Программные прерывания вызываются выполнением специальной команды, но обрабатываются точно так же, как остальные типы прерываний. По сути, команда программного прерывания представляет собой особый случай вызова подпрограммы, но при этом вместо адреса подпрограммы указывается номер прерывания, обработчик которого должен быть вызван. В большинстве современных ОС программные прерывания используются для перехода из режима пользователя в режим ядра при вызове системных функций из прикладной программы.

Одним из важнейших источников прерываний являются периферийные устройства. Как правило, устройство генерирует сигнал прерывания в одном из двух случаев:

- при переходе в состояние готовности;
- при возникновении ошибки выполнения операции.

Состояние готовности – это такое состояние устройства, в котором оно готово принять и выполнить команды от процессора. Для устройства ввода готовность означает наличие в устройстве данных, которые могут быть переданы в процессор (например,

клавиатура переходит в состояние «Готово» при нажатии клавиши и возвращается в состояние «Не готово», когда код нажатой клавиши считан в процессор). Для устройства вывода готовность – это возможность принять от процессора данные, которые следует вывести. Например, матричный принтер принимает символы, которые нужно напечатать, в свой внутренний буфер. Если буфер полон, принтер переходит в состояние «Не готово» до тех пор, пока часть символов будет напечатана и в буфере освободится место. Дисковый накопитель при начале выполнения новой операции чтения или записи на диск переходит в состояние «Не готово», а после завершения операции возвращается в состояние «Готово». В любом из этих случаев переход в состояние «Готово» – это повод для устройства напомнить о себе процессору: обратите на меня внимание, я к вашим услугам! Для этого и служит сигнал прерывания.

Ошибка операции также требует вмешательства системы или пользователя. Например, при ошибке отсутствия бумаги в лотке принтера система должна оповестить об этом пользователя; при ошибке чтения с диска либо система, либо пользователь должен решить, что делать: повторить операцию, завершить программу или продолжить выполнение.

Не каждое устройство генерирует прерывания. Например, монитор ПК не выдает прерываний: он «всегда готов», т.е. всегда может принять данные для отображения, и он «никогда не ошибается», точнее сказать, его неисправность обнаруживается «на глаз».

Способы организации ввода/вывода

Ввод/вывод по опросу и по прерываниям

Рассмотрим более подробно работу программы, непосредственно выполняющей ввод или вывод данных на конкретное устройство. (На самом деле, этой работой обычно занимается драйвер устройства, так что мы фактически рассматриваем логику работы драйвера.)

Для определенности положим, что программа должна выдать **N** байт данных из массива **A** на символьное устройство **X**. Для операции ввода могут использоваться те же подходы, которые будут рассмотрены здесь для операции вывода.

Пусть архитектура устройства представлена регистром данных **X.DATA** и флагом готовности **X.READY**. Когда **X.READY** = **TRUE**, в регистр **X.DATA** можно выдавать очередной байт данных. Запишем на псевдокоде, близком к языку Паскаль, варианты организации соответствующей программы.

а) Ввод/вывод без проверки готовности

```
i := 1;
while i <= N do begin
X.DATA := A[i];
  i := i + 1;
end;
```

Этот «наглый» способ вывода вполне работоспособен, если используется «всегда готовое» устройство (например, монитор), т.е. флаг **X.READY** всегда истинен и потому вообще не нужен. При попытке использовать тот же подход для вывода на принтер мы убедились бы, что напечатаны будут лишь некоторые символы, которым посчастливилось быть выданными в редкие моменты готовности принтера.

б) Ввод/вывод по опросу готовности

```
i := 1;
while i <= N do begin
  while not X.READY do
    ;
  X.DATA := A[i];
  i := i + 1;
end;
```

Здесь добавлен цикл ожидания, в котором не делается ничего, кроме постоянной циклической проверки готовности устройства. Передача данных происходит только тогда, когда устройство готово. Поскольку после выдачи одного байта устройство вполне может опять перейти в состояние неготовности, следует опять выполнять цикл ожидания, пока выданный символ не будет обработан устройством.

Такая организация ввода/вывода позволяет корректно работать с любыми устройствами. Этот способ действительно применяется в некоторых однозадачных системах. Недостатком данного способа является непроизводительная трата времени на постоянную проверку флага готовности. При современном соотношении скоростей работы процессора и периферии, цикл ожидания может повторяться миллионы раз перед выдачей каждого байта. Более того, если по каким-то причинам устройство вообще не перейдет в состояние готовности, то работа всей системы может быть парализована бесконечным циклом ожидания.

в) Ввод/вывод по прерываниям

```
i := 1;
```

```
while i <= N do begin
X_INT: if not X.READY
  return;
  X.DATA := A[i];
  i := i + 1;
end;
```

Здесь исчез цикл ожидания, вместо него – однократная проверка готовности и оператор возврата, если не готово.

Куда происходит возврат? Чтобы это понять, надо вспомнить, что данный фрагмент – явно не единственная программа, работающая в данный момент на ЭВМ. Очевидно, операция вывода была начата операционной системой по запросу какой-то программы. Данный фрагмент был вызван как подпрограмма ОС, и возврат означает передачу управления ОС. Как система распорядится полученным временем? Это уже совсем другой вопрос, не связанный с вводом/выводом. Например, ОС может переключиться на другой процесс. Или, от нечего делать, запустить экранную заставку либо программу самотестирования.

Но как же быть с брошенной на полпути операцией вывода? Для ее возобновления будет использовано аппаратное прерывание, которое должно выдать устройство **X** при переходе в состояние готовности. Системный обработчик прерывания должен будет передать управление по адресу, обозначенному меткой **X_INT**. После нелишней дополнительной проверки готовности программа вывода передаст очередной байт на устройство, затем снова проверит готовность и, возможно, вновь вернет управление системе. Таким образом, выполнение ввода/вывода разбивается на отдельные интервалы работы при готовности устройства, перемежающиеся работой системы, пока устройство не готово.

Для устройств, использующих контроллер ПДП, возможные варианты организации работы остаются, по сути, теми же, но только используются гораздо более крупные операции: вместо ввода или вывода одного элемента данных выполняется ввод/вывод целого блока данных, и только после этого контроллер переходит в состояние готовности и генерирует прерывание.

Активное и пассивное ожидание

Основной особенностью ввода/вывода по опросу готовности является цикл ожидания. Если выполняется ввод или вывод на медленное устройство (например, матричный принтер), то этот скромно выглядящий цикл будет занимать 99% всего времени

работы процессора. Если происходит ожидание ввода с клавиатуры, то процессор вообще не будет делать ничего полезного, пока пользователь не нажмет клавишу.

Такое использование процессора может быть оправдано разве что в том случае, если для него нет никакой более полезной работы. Это возможно в случае однозадачной ОС, когда работающая прикладная программа не может продвигаться дальше, пока не завершена операция ввода/вывода. В этом случае ввод/вывод по опросу не лишен определенных достоинств: он не связан с обработкой прерываний, которая требует некоторого времени, а потому замедляет реакцию на переход устройства в состояние готовности.

Способ ожидания программой некоторого события, основанный на постоянной циклической проверке ожидаемого условия, называется *активным ожиданием* (busy waiting). Это понятие применяется не только по отношению к вводу/выводу, но и во многих других ситуациях, возникающих при работе системных и прикладных программ.

Если рассматривается многозадачная ОС, в которой может быть несколько активных задач одновременно, то активное ожидание становится совершенно неприемлемым. В этом случае расход процессорного времени на выполнение циклического опроса наносит прямой ущерб другим программам, которые могли бы использовать это время более осмысленно. Поэтому при разработке многозадачных систем, как при вводе/выводе, так и в некоторых других ситуациях, обязательно реализуется *пассивное ожидание*, т.е. такая реализация ожидания, при которой ожидающая программа не затрачивает процессорного времени. Для реализации пассивного ожидания всегда в той или иной форме используются аппаратные прерывания. Частным примером пассивного ожидания является рассмотренный выше ввод/вывод по прерываниям.

Буферизация и кэширование

Буферизацию в самом широком смысле можно определить как такую организацию ввода/вывода, при которой данные не передаются непосредственно с устройства в заданную область памяти (или из области памяти на устройство), а предварительно направляются во вспомогательную область памяти, называемую *буфером*. через несколько буферов разного назначения.

Существует несколько причин для использования буферизации, важнейшие из которых рассмотрены ниже.

Сглаживание неравномерности скоростей процессов

Достаточно часто в работе ОС встречается ситуация, когда один процесс порождает данные, которые должны оперативно обрабатываться другим процессом. В качестве примера

можно привести прием по сети данных, которые должны обрабатываться браузером или другой прикладной программой.

Скорость приема данных очень неравномерна: интервалы времени интенсивного поступления данных перемежаются с интервалами простоя. Обработка данных прикладной программой тоже не обязательно идет с постоянной скоростью. В результате, хотя средняя скорость обработки может быть вполне достаточной, не исключено, что в некоторые моменты обрабатывающая программа будет «захлебываться» данными. Это может привести к потере части данных, не успевших пройти обработку.

Стандартным решением в этой ситуации является использование буфера, размер которого достаточно велик, чтобы вместить все данные, ожидающие обработки. Чем больше буфер, тем меньше вероятность потери данных из-за его переполнения.

Распараллеливание ввода и обработки

Во многих вычислительных системах имеются аппаратные возможности совместить во времени выполнение операций ввода/вывода и обработку данных процессором. Чтобы использовать эти возможности, данные при вводе направляются в буфер. После заполнения буфера его данные пересылаются в обрабатывающую программу, а их обработка выполняется параллельно с накоплением следующей порции данных в буфере.

Согласование размеров логической и физической записи

Логической записью называют порцию данных, указанную в операторе ввода/вывода. Размер логической записи определяется логикой работы программы или, например, логической структурой базы данных.

При фактическом выполнении чтения или записи на блочное устройство обрабатывается порция данных, называемая физической записью или блоком. Размер физической записи определяется особенностями устройства (для диска это один сектор) и никак не связан с логикой программы.

На рисунке 1 показана ситуация, когда логическая запись содержит 100 байт, а физическая – 512 байт.

Предположим, что логические записи в последовательном порядке записываются в файл на диске. Если каждый оператор вывода логической записи будет вызывать немедленную запись на диск, то выдача первых пяти логических записей потребует пять раз выполнить последовательность операций: «чтение физической записи с диска – изменение 100 байт – запись измененной записи на диск». На шестой раз получится еще хуже, поскольку придется читать – изменять – записывать не одну, а две физических записи.

Использование буфера для накопления данных до размера физической записи позволяет резко сократить количество операций записи на диск и почти полностью исключить чтение с диска.

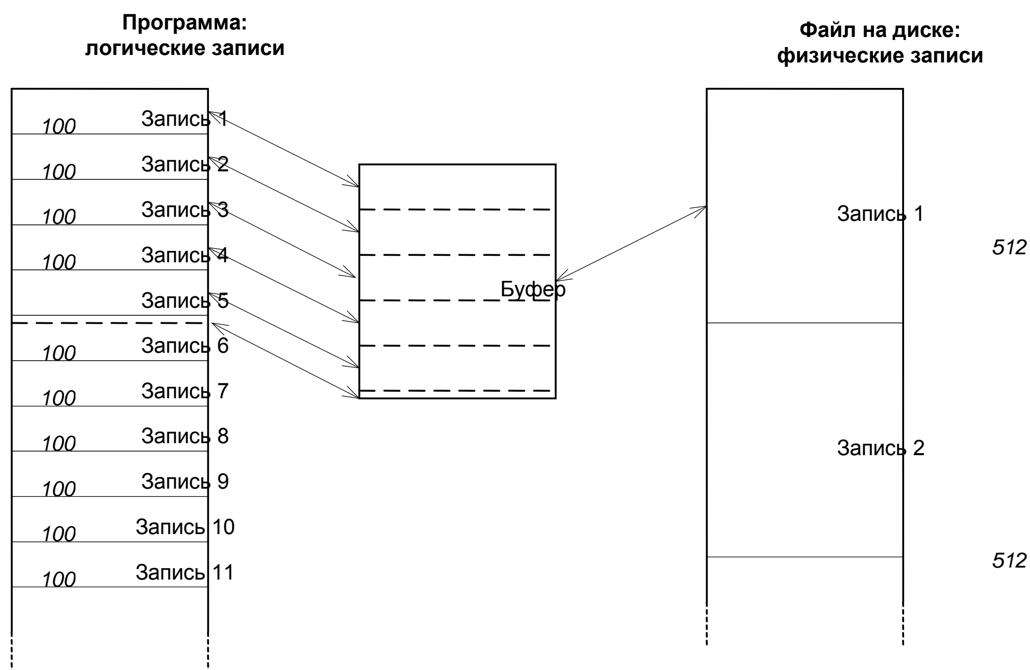

Рисунок - 1 Логические и физические записи

Драйверы устройств

Драйвер устройства – это системная программа, которая под управлением ОС выполняет все операции с конкретным периферийным устройством. Драйвер является как бы посредником между ОС и устройством. Перед драйверами стоят две одинаково важные, но трудно совместимые задачи:

- обеспечить возможность стандартного обращения к любому устройству, скрывая от остальных частей ОС специфические особенности отдельных устройств;
- добиться максимально эффективного использования всех функциональных возможностей и особенностей конкретных устройств.

В большинстве ОС различаются, как минимум, два разных типа драйверов: для символьных и для блочных устройств.

Обращаясь к драйверу, ОС указывает функцию, которую требуется выполнить. Список этих функций общий для драйверов различных устройств, при этом каждый драйвер может реализовать только те функции, которые имеют смысл для данного устройства. Наиболее общими являются функции чтения данных, записи данных, инициализации устройства (эта функция вызывается системой один раз, сразу после загрузки), открытия и закрытия устройства (используются, когда символьное устройство открывается как файл). Для блочных устройств имеют смысл функции форматирования, поиска сектора. Для символьных устройств ввода – функция «неразрушающего ввода», т.е. проверки очередного символа без его изъятия из входного потока.

Типичный драйвер устройства содержит, как минимум, три основных блока:

- заголовок драйвера;
- блок стратегии;
- блок прерываний.

Заголовок содержит различную информацию о данном драйвере и об управляемом устройстве. Сюда может включаться имя устройства, тип устройства, число однотипных устройств, обслуживаемых одним драйвером, объем памяти на устройстве и т.п. Заголовок содержит также адреса блока стратегии и блока прерываний.

В обязанность *блока стратегии* входит прием заявок на выполнение операции, ведение очереди заявок (в многозадачных системах, а также при асинхронных операциях, выполнения могут дожидаться несколько заявок), а также запуск операции и ее завершение.

Заявка на выполнение операции представляет собой стандартную запись, формируемую системой перед обращением к драйверу. Заявка содержит код требуемой функции драйвера и сведения об адресе данных в памяти и на устройстве, о количестве передаваемых данных. Заявка также содержит поле, в которое драйвер должен будет записать код завершения операции (обычно 0 – нормально выполненная операция, другие значения – коды ошибок).

Блок прерываний выполняет примерно тот алгоритм, который ранее назывался вводом/выводом по прерываниям. Система вызывает этот блок, когда получает сигнал прерывания от устройства, обслуживаемого драйвером. Закончив выполнение заявки, блок прерываний возвращает управление блоку стратегии для завершения операции.

Помимо трех основных блоков, в разных ОС драйверы могут содержать, например, блок инициализации (он используется один раз при загрузке ОС, а затем может быть выгружен из памяти), блок изменения параметров драйвера и др.

Управление устройствами в MS-DOS

Система MS-DOS предоставляет пользователю возможности доступа к устройствам на нескольких уровнях, отличающихся степенью близости к аппаратуре. Нижние уровни позволяют более полно использовать тонкие особенности устройств, но за это приходится платить сложностью программирования. Верхние уровни более удобны для решения стандартных задач ввода/вывода.

Уровни доступа к устройствам показаны на рисунке 2.

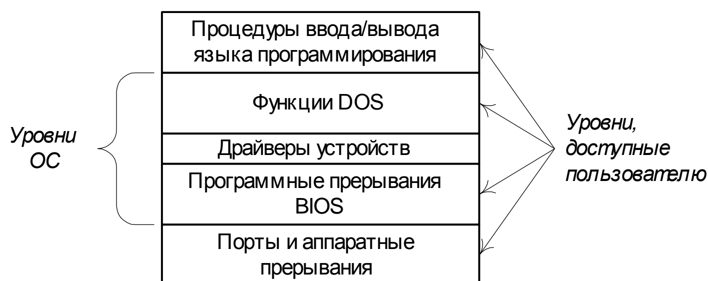

Рисунок 2 - Уровни доступа к устройствам в MS-DOS

Самый нижний уровень предполагает, что программа пользователя работает непосредственно с портами ввода/вывода, а также, если это необходимо, частично или полностью берет на себя обработку аппаратных прерываний, поступающих от устройства.

Программные прерывания BIOS представляют собой подпрограммы, выполняющие операции ввода/вывода и управления конкретными устройствами. Для каждого из стандартных устройств зарезервирован свой номер прерывания, а для указания требуемой операции используется номер функции, который заносится в один из регистров процессора перед вызовом прерывания.

Драйверы устройств не вызываются непосредственно из программы пользователя. Они вызываются из функций DOS и выполняют заданную операцию, обычно используя для этого программные прерывания BIOS, хотя возможна и прямая работа драйвера с портами и аппаратными прерываниями.

Функции DOS, в отличие от прерываний BIOS, работают не с конкретной аппаратурой, а с именованными устройствами. Имена устройств задаются в заголовках соответствующих драйверов. Например, для DOS клавиатура и экран объединяются драйвером консольного устройства CON.

Процедуры ввода/вывода, используемые в языках программирования (например, *Read* и *Write* в Паскале, *scanf* и *printf* в C), при компиляции реализуются как подходящие вызовы функций DOS или, в особых случаях, программных прерываний BIOS.

Управление устройствами в Windows

Поскольку Windows – многозадачная система, она исключает для прикладных программ такие вольности, как прямое обращение к портам ввода/вывода или обработка аппаратных прерываний. Взаимодействие с аппаратурой на низком уровне может выполняться только системными программами, работающими в привилегированном режиме. Основную роль здесь играют драйверы устройств.

В Windows используется многоуровневая структура драйверов, в которой высокоуровневые драйверы могут играть роль фильтров, выполняющих специальную обработку данных, полученных от драйвера низкого уровня или передаваемых такому драйверу. В качестве примера можно привести отделение драйвера, управляющего шиной, от драйверов конкретных устройств, подключенных к шине. Еще один пример – драйвер, выполняющий шифрацию/дешифрацию данных при работе с файловой системой NTFS. Структура драйверов всех уровней подчинена единым стандартам, известным как WDM (Windows Diver Model), однако высокоуровневые драйверы, в отличие от низкоуровневых, не занимаются обработкой аппаратных прерываний.

Управление устройствами в UNIX

Интересной отличительной особенностью UNIX является то, что для работы с периферийными устройствами прикладные программы могут и должны использовать те же средства, что для работы с файлами. Вообще, устройства в UNIX представлены как *специальные файлы*, вписанные в каталог файловой системы наравне с обычными файлами. Каждому драйверу устройства соответствует отдельный специальный файл, символьный или блочный, в зависимости от типа драйвера. Как правило, все специальные файлы размещаются в каталоге */dev*. Чтобы начать работу с устройством, программа должна вызвать функцию открытия файла, указав ей имя специального файла. При этом происходит обращение к функции открытия из драйвера соответствующего устройства.

С каждым специальным файлом связаны два числа, называемые старшим и младшим номерами устройства. Старший номер определяет номер строки в таблице символьных либо блочных драйверов. Младший номер передается драйверу как дополнительный параметр. Он может означать, например, номер конкретного дискового устройства.

РАЗДЕЛ 3. УПРАВЛЕНИЕ ДАННЫМИ

Основные задачи управления данными

Старинный термин «управление данными» в настоящее время всегда понимается как управление файлами.

Файл есть набор данных, хранящийся на периферийном устройстве и доступный по имени. При этом конкретное расположение данных на устройстве не интересует пользователя и полностью передоверяется системе. До изобретения файлов пользователь должен был обращаться к своим данным, указывая их адреса на диске или на магнитной ленте.

Понятие «*файловая система*» означает стандартизованную совокупность структур данных, алгоритмов и программ, обеспечивающих хранение файлов и выполнение операций с ними. Мощная современная ОС обычно поддерживает возможность использования нескольких разных файловых систем. И наоборот, одна и та же файловая система может поддерживаться различными ОС.

Среди задач, решаемых подсистемой управления данными, можно назвать следующие:

- выполнение операций создания, удаления, переименования, поиска файлов, чтения и записи данных в файлы, а также ряда вспомогательных операций;
- обеспечение эффективного использования дискового пространства и высокой скорости доступа к данным;
- обеспечение надежности хранения данных и их восстановления в случае сбоев;
- защита данных пользователя от несанкционированного доступа;
- управление одновременным совместным использованием данных со стороны нескольких процессов.

Характеристики файлов и архитектура файловых систем

С каждым файлом связан набор *атрибутов* (*характеристик*), т.е. набор сведений о файле. Состав атрибутов может сильно различаться для разных файловых систем. Приведем примерный список возможных атрибутов, не привязываясь к какой-либо конкретной системе.

- *Имя файла*. В старых ОС длина имени была жестко ограничена 6 – 8 символами с целью экономии места для хранения имени и ускорения работы. В настоящее

время максимальная длина имени составляет обычно около 250 символов, что позволяет при желании включить в имя файла подробное описание его содержимого.

- *Расширение имени*. По традиции, так принято называть правую часть имени, отделенную точкой. В MS-DOS, как и в некоторых более ранних системах, этот атрибут не является частью имени, он хранится отдельно и ограничивается по длине 3 символами. Однако сейчас возобладал подход, принятый в UNIX, где расширение – это чисто условно выделяемая часть имени после последней точки. Расширение обычно указывает тип данных в файле.

- *Тип файла*. Некоторые ОС выделяют несколько существенно различных типов файлов, например, символьные и двоичные, файлы данных и файлы программ и т.п. Ниже будут рассмотрены типы файлов, различаемые UNIX.

- *Размер файла*. Обычно указывается в байтах, хотя раньше часто задавался в блоках.

- *Временные штампы*. Под этим термином понимаются различные отметки даты и, может быть, времени дня. Важнейшим из временных штампов является время последней модификации, позволяющее определить наиболее свежую версию файла. Полезными могут быть также время последнего доступа (т.е. открытия файла), время последней модификации атрибутов.

- *Номер версии*. В некоторых ОС при всяком изменении файла создавалась его новая версия, причем система могла хранить либо все версии, либо только несколько последних. Это давало немаловажное преимущество – возможность вернуться к старой версии файла, если изменения оказались неудачными. Тем не менее, этот атрибут не привился из-за большой избыточной траты дисковой памяти. При необходимости разработчики могут использовать специальные программные системы управления проектами, обеспечивающие в том числе и хранение старых версий файлов.

- *Владелец файла*. Этот атрибут необходим в многопользовательских системах для организации защиты данных. Как правило, владельцем является пользователь, который создал файл. Иногда, кроме индивидуального владельца, указывается еще и группа пользователей как коллективный владелец файла.

- *Атрибуты защиты*. Они указывают, какие именно права доступа к файлу имеют различные пользователи, в том числе и владелец файла.

- *Тип доступа*. В некоторых ОС (например, в OS/360) для каждого файла должен был храниться допустимый тип доступа: последовательный, произвольный или один из

индексных типов, обеспечивающих быстрый поиск данных в файле. В настоящее время более распространен подход, при котором для всех файлов поддерживаются одни и те же типы доступа (последовательный и произвольный), а ускорение поиска должно обеспечиваться, например, системой управления базами данных.

- *Размер записи*. Если эта величина указана, то адресация нужных данных выполняется с помощью номера записи. Другой подход заключается в том, что данные адресуются их смещением (в байтах) от начала файла, а разбиение файла на записи возлагается на прикладные программы, работающие с файлом.

- *Флаги (битовые атрибуты*). Их разнообразие ограничивается лишь фантазией разработчиков системы, но наиболее распространенным и важным является флаг «только для чтения» (read only), защищающий файл от случайного изменения или удаления. В зависимости от возможностей конкретной файловой системы, файл может быть отмечен как «сжатый», «шифрованный» и т.п.

- *Данные о размещении файла на диске*. Пользователь, как правило, не знает и не хочет ничего знать о размещении файла (именно для этого и существует понятие файла). Для системы эти данные необходимы, чтобы найти файл.

Записи, в которых содержатся атрибуты каждого файла, собраны в *каталоги* (они же папки, директории). В ранних ОС (и даже в первой версии MS-DOS) на каждом дисковом томе имелся единственный каталог, содержащий полный список всех файлов этого тома. Такое решение было вполне естественным, пока количество файлов не превышало двух – трех десятков. Однако при увеличении объема дисков и, как следствие, числа файлов на них такой одноуровневый каталог становился все менее удобным. В некоторых ОС использовалась двухуровневая организация каталогов. При этом главный каталог содержал список каталогов второго уровня, закрепленных за отдельными пользователями или проектами. Однако позднее стала общепринятой *иерархическая структура каталогов*, при которой каждый каталог может, помимо файлов, содержать вложенные подкаталоги, причем глубина вложения не ограничивается.

Все хранящиеся в файловой системе служебные данные, описывающие атрибуты и размещение файлов, структуру каталогов, общую структуру дискового тома и т.п., принято называть *метаданными*, в отличие от «просто данных», хранящихся в файлах.

Размещение файлов

Область данных диска, отведенную для хранения файлов, можно представить как линейную последовательность адресуемых блоков (секторов). Размещая файлы в этой

области, ОС должна отвести для каждого файла необходимое количество блоков и сохранить информацию о том, в каких именно блоках размещен данный файл. Существуют два основных способа использования дискового пространства для размещения файлов.

- *Непрерывное размещение* характеризуется тем, что каждый файл занимает непрерывную последовательность блоков.
- *Сегментированное размещение* означает, что файлы могут размещаться «по кусочкам», т.е. один файл может занимать несколько несмежных сегментов разной длины. Оба способа размещения показаны на рисунке 3.

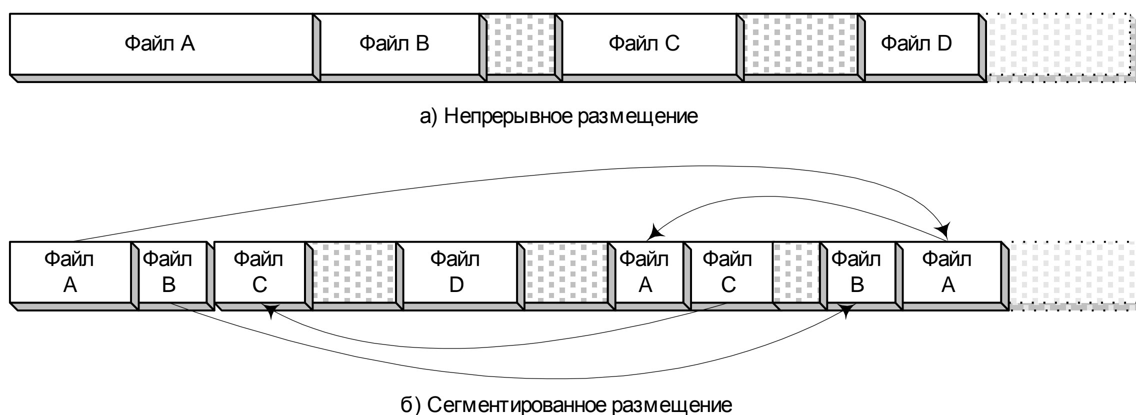

Рисунок 3 - Способы размещения файлов

Непрерывное размещение имеет два серьезных достоинства.

- Информация о размещении файла очень проста и занимает мало места. Фактически достаточно хранить два числа: номер начального блока файла и число занимаемых блоков (или размер файла в байтах, по которому легко вычислить число блоков).
- Доступ к любой позиции в файле выполняется быстро, поскольку, зная смещение от начала файла, легко можно вычислить номер требуемого блока и прочитать сразу этот блок, не читая предыдущие блоки.

Недостатки непрерывного распределения еще более весомы.

- При создании файла требуется заранее знать его размер, чтобы найти и зарезервировать на диске область достаточной величины. Последующее возможное увеличение файла весьма затруднено, т.к. после конца файла может не оказаться достаточно свободного места. Фактически вместо увеличения файла обычно приходится заново создавать файл большего размера в другом месте, переписывать в него данные и удалять старый файл. Но такое решение требует много времени на чтение и запись данных и, кроме

того, снижает надежность хранения данных, поскольку ошибка при чтении или записи гораздо более вероятна, чем порча данных, «спокойно лежащих» на диске.

- В ходе обычной эксплуатации файловой системы, после многократного создания и удаления файлов разной длины, свободное пространство на диске оказывается разбитым на небольшие кусочки. Суммарный объем свободного места на диске может быть достаточно большим, но создать файл приличного размера не удается, для него нет непрерывной области нужной длины. Это явление носит название *фрагментации диска*. Для борьбы с ним приходится использовать специальную процедуру дефрагментации, которая перемещает все файлы, размещая их впритык друг к другу от начала области данных диска. Но такая процедура требует много времени, снижает, как сказано выше, надежность и усугубляет проблемы в случае, если позднее потребуется увеличить файл.

Сегментированное размещение лишено первого из недостатков непрерывного: при создании файла ему обычно вообще не выделяют память, а потом, по мере возрастания размера файла, ему могут быть выделены любые свободные сегменты на диске, независимо от их длины.

Еще одной важной характеристикой размещения файлов является степень его «дробности». До сих пор мы предполагали, что файл может занимать любое целое число блоков, а под блоком фактически понимали сектор диска. Проблема в том, что для дисков большого объема число блоков может быть слишком большим. Допустим, в некоторой файловой системе размер блока равен 512 байт, а для хранения номеров блоков файла используются 16-разрядные числа. В этом случае размер области данных диска не сможет превысить $512 * 2^{16} = 32$ Мб, что нынче смешно. Конечно, можно перейти к использованию 32-разрядных номеров блоков, но тогда суммарный размер информации о размещении всех файлов на диске становится чересчур большим. Обычный выход из этого затруднения заключается в том, что минимальной единицей размещения файлов считают *кластер* (называемый в некоторых системах *блоком* или *логическим блоком*), который принимается равным 2^k секторов, т.е., например, 1, 2, 4, 8, 16, 32 сектора, редко больше. Каждому файлу отводится целое число кластеров, и в информации о размещении файла хранятся номера кластеров, а не секторов. Увеличение размера кластеров позволяет сократить количество данных о размещении файлов «и в длину и в ширину»: во-первых, для каждого файла нужно хранить информацию о меньшем числе кластеров, а во-вторых, уменьшается число двоичных разрядов, используемых для задания номера кластера (либо при той же разрядности можно использовать больший диск). Так, при кластере размером 32 сектора и 16-разрядных номерах можно адресовать до 1 Гб дисковой памяти.

Использование больших кластеров имеет свою плохую сторону. Поскольку размер файла можно считать случайной величиной (по крайней мере, этот размер никак не связан с размером кластера), то можно приближенно считать, что в среднем половина последнего кластера каждого файла остается незанятой. Это явление иногда называют *внутренней фрагментацией* (в отличие от описанной выше фрагментации свободного пространства диска, которую называют также *внешней* фрагментацией). Кроме того, если хотя бы один из секторов, входящих в кластер, отмечен как дефектный, то и весь кластер считается дефектным, т.е. не может быть использован. Очевидно, что при увеличении размера кластера возрастает и число неиспользуемых секторов диска.

Оптимальный размер кластера либо вычисляется автоматически при форматировании диска, либо задается вручную.

Для нормальной работы файловой системы требуется, чтобы, кроме информации о размещении файлов, система хранила в удобном для использования виде информацию об имеющихся свободных кластерах диска. Эта информация необходима при создании новых или увеличении существующих файлов. Используются различные способы представления информации о свободном месте, некоторые из них перечислены ниже.

- Можно хранить все свободные кластеры как связанный линейный список, т.е. в начале каждого свободного кластера хранить номер следующего по списку. Недостаток такого способа в том, что затрудняется поиск свободного непрерывного фрагмента нужного размера, поэтому сложнее оптимизировать размещение файлов.

- Названный недостаток можно преодолеть, если хранить список не из отдельных кластеров, а из непрерывных свободных фрагментов диска. Правда, работать с таким списком несколько сложнее.

- В системах с непрерывным размещением часто каждый непрерывный фрагмент диска описывают так же, как файл, но отмечают его флажком «свободен».

- Удобный и простой способ заключается в использовании *битовой карты* (bitmap) свободных кластеров. Она представляет собой массив, содержащий по одному биту на каждый кластер, причем значение 1 означает «кластер занят», а 0 – «кластер свободен». Для поиска свободного непрерывного фрагмента нужного размера система должна будет просмотреть весь массив.

Защита данных

В многопользовательских ОС первостепенное значение приобретает задача защиты данных пользователя от случайного или намеренного доступа со стороны других пользователей. Вопросы защиты данных и стандартизации требований к безопасности ОС заслуживают изучения в отдельном курсе, поэтому здесь они будут рассмотрены очень кратко.

Как отмечалось в п. 0, для реализации многопользовательской защиты данных необходимо наличие аппаратных средств, таких как привилегированный режим работы процессора. В противном случае любая чисто программная система защиты могла бы быть нарушена с помощью достаточно изощренной программы взлома.

Для любой системы защиты характерно наличие, по крайней мере, трех компонент.

- Список пользователей системы, содержащий имена, пароли и привилегии, присвоенные пользователям.
- Наличие атрибутов защиты у файлов и других защищаемых объектов. Эти атрибуты указывают, кто из пользователей имеет право доступа к данному объекту и какие именно операции ему разрешены.
- Процедура *аутентификации пользователя*, т.е. установление его личности при входе в систему. Такие процедуры чаще всего основаны на вводе пароля, хотя могут использоваться и более экзотические средства (отпечатки пальцев, специальные карточки и т.п.).

Помимо отдельных пользователей, определенными правами доступа к объектам могут обладать *группы пользователей*. Понятие группы облегчает администрирование прав доступа. Вместо того, чтобы индивидуально указывать набор прав для каждого пользователя, достаточно зачислить его в одну или несколько групп, права которых определены заранее.

Нормальное обслуживание системы защиты невозможно без наличия *администратора системы* (он же в различных системах именуется *привилегированным пользователем* или *суперпользователем*) или же группы пользователей, обладающих правами администратора. Администратор назначает права прочим пользователям, а также имеет возможность в чрезвычайных случаях получить доступ к объектам любого владельца. Однако при этом желательно, чтобы действия администратора, как минимум, фиксировались системой с целью выявления возможных злоупотреблений с его стороны.

Разделение файлов между процессами

В многозадачных ОС, а также в сетевых системах, возможна ситуация, когда два или более процессов пытаются одновременно использовать один и тот же файл. Например, два пользователя могут одновременно работать с одной базой данных. Будем предполагать, что с правами доступа все в порядке, каждый процесс в отдельности имеет право читать и записывать файл. Вопрос в том, можно ли разрешить одновременную работу, не приведет ли это к нарушению целостности данных.

Может привести. Если один процесс обновляет данные в файле, а другой в это время пытается читать эти же данные, то он может прочесть частично обновленные данные. Еще опаснее, когда два процесса пытаются одновременно изменить одни и те же данные. В этом случае трудно даже предсказать, что в результате будет сохранено в файле.

В принципе, всегда безопасными являются лишь два крайних случая:

- только один процесс работает с файлом, выполняя чтение и запись;
- с файлом работает произвольное число процессов, но все они выполняют только чтение.

ОС могла бы обеспечить безопасный доступ, разрешая процессу открывать файл только в этих двух случаях, т.е. если файл не открыт еще ни одним другим процессом либо если файл открыт кем-то только для чтения и данный процесс тоже открывает его для чтения. Однако такая суровость в ряде случаев существенно снизила бы производительность системы. Скажем, много пользователей хотели бы одновременно работать с одной базой данных. И в этом нет ничего плохого, пока они работают с разными записями базы. Опасность возникает только при одновременной работе с одной и той же записью. Но ОС не может сама отследить ситуацию так подробно, это может сделать программа, управляющая базой данных. Ввиду подобных ситуаций, большинство ОС позволяют программам процессов самим определять, допустим ли совместный доступ в различных конкретных ситуациях.

Типичное решение заключается в следующем. Прикладная программа, вызывая системную функцию открытия файла, указывает два дополнительных параметра: режим доступа и режим разделения.

Режим доступа определяет, какие операции сам процесс собирается выполнять с файлом. Обычно различают доступ «только для чтения», «только для записи», «для чтения и записи».

Режим разделения определяет, какие операции данный процесс готов разрешить другим процессам, которые захотят открыть тот же файл. Примерный набор режимов разделения – «запрет записи», «запрет чтения», «запрет чтения и записи» и «без запретов».

Первый процесс, открывающий файл, устанавливает по своему усмотрению режимы доступа и разделения. Когда второй процесс пытается открыть тот же файл, ОС проверяет два условия:

- режим доступа второго процесса не должен противоречить режиму разделения, установленному первым процессом;
- режим разделения, запрашиваемый вторым процессом, не должен запрещать тот режим доступа, который уже установил для себя первый процесс.

В случае нарушения одного из этих условий система не открывает файл для второго процесса, функция открытия файла возвращает ошибку. Если же условия соблюдены, система открывает файл для второго процесса, как бы снимая с себя ответственность за последствия: вы этого хотели – получите.

Файловая система FAT и управление данными в MS-DOS

Общая характеристика системы FAT

Система FAT была разработана для ОС MS-DOS. Это простая файловая система с сегментированным размещением, без многопользовательской защиты. Структура каталогов – древовидная, причем на каждом дисковом томе создается отдельное дерево. Для указания местоположения файла может использоваться его *полное имя*, содержащее букву диска, путь по дереву каталогов и собственно имя файла, например: «***C***:*UTILS**ARCH**RAR.EXE*».

В ОС Windows также возможно использование FAT, особенно оправданное для дискет. Для жестких дисков большого объема система FAT становится малоэффективной и постепенно вытесняется более мощной системой NTFS.

Структуры данных на диске

При форматировании дискеты или раздела жесткого диска в системе FAT все дисковое пространство разбивается на следующие области, показанные на рисунке 4.

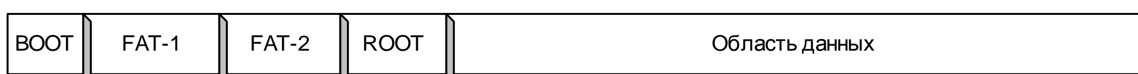

Рисунок 4 - Структура диска в файловой системе FAT

- *BOOT-сектор* содержит основные количественные параметры дискового тома и файловой системы, а также может содержать программу начальной загрузки ОС.

- *таблица FAT* (File Allocation Table) – содержит информацию о размещении файлов и свободного места на диске. Ввиду критической важности этой таблицы она всегда хранится в двух экземплярах, которые должны быть идентичны. Каждая операция, изменяющая содержимое FAT, должна одинаковым образом изменять оба экземпляра.
- *ROOT* – корневой каталог системы, содержащий данные о файлах и о подкаталогах верхнего уровня, каждый из которых в свою очередь может содержать файлы и подкаталоги.
- *Область данных* – массив кластеров, содержащий все файлы и все каталоги (кроме корневого).

Рассмотрим подробно, как хранится вся информация о файле, имеющаяся в системе FAT.

При создании файла в одном из каталогов файловой системы создается запись, хранящая основной объем информации об этом файле. Каждый каталог, кроме корневого, также является файлом особого вида, и запись о нем содержится в родительском каталоге. Каталожная запись всегда занимает 32 байта, ее структура показана в табл. 1.

Таблица 1 – Структура записи каталога файловой системы AT

Поле записи	**Размер поля (в байтах)**
Имя файла	8
Расширение имени (тип файла)	3
Атрибуты (флаги)	1
Размер файла (в байтах)	4
Дата последнего изменения	2
Время последнего изменения	2
Резерв (не используется)	10
Номер первого кластера файла	2

Как видно из таблицы, имя файла может занимать не более 8 символов плюс еще 3 символа расширения. В начале 80-х годов казалось, что этого вполне достаточно. Позднее

это ограничение окрестили «проклятием 8 + 3», и избавить от него файловую систему FAT удалось только в Windows 95.

Байт атрибутов содержит набор битов, характеризующих свойства файла. Наряду с практически бесполезными атрибутами «скрытый», «системный» и «архивный», там содержатся и важные: «только для чтения», «каталог» и «метка тома». Атрибут «только для чтения» запрещает системе удалять файл или открывать его для записи. Атрибут «каталог» означает, что данная запись описывает не обычный файл, а каталог. Атрибут «метка тома» может содержаться только в корневом каталоге, такая запись не описывает никакой файл, а вместо этого содержит в полях имени и расширения 11-символьную метку (имя), присвоенную данному дисковому тому.

В целом, запись каталога содержит почти все, что системе известно о файле, а если размер файла не превышает одного кластера, то полностью все. Если же файл содержит более одного кластера, то номера остальных можно найти в таблице FAT.

Таблица FAT состоит из записей, количество которых равно количеству кластеров в области данных, а размер одной записи может быть равен 12, 16 или 32 битам. Соответственно говорят о разновидностях файловой системы FAT-12, FAT-16 или FAT-32. Размер записи должен быть таким, чтобы в ней можно было записать максимальный номер кластера. Например, для стандартной трехдюймовой дискеты емкостью 1.44 Мб достаточно использовать FAT-12, поскольку это позволяет иметь $2^{12} = 4096$ кластеров (на самом деле, чуть меньше), и даже при кластерах размером в 1 сектор (512 байт) этого более чем достаточно: $4096 \times 512 = 2$ Мб.

Записи FAT «по историческим причинам» нумеруются, начиная с 2 и кончая максимальным номером кластера, каждая запись FAT описывает соответствующий кластер с тем же номером. Запись может принимать следующие значения:

- если кластер принадлежит некоторому файлу (или каталогу) и является последним (или единственным) в этом файле, то запись FAT содержит специальное значение – все единицы (FFF_{16} для FAT-12 или $FFFF_{16}$ для FAT-16);
- если кластер принадлежит некоторому файлу (или каталогу), но не является последним в файле, то запись FAT содержит номер следующего кластера того же файла;
- если кластер свободен, то запись содержит все нули;
- если кластер дефектный (т.е. при проверке диска выяснилось, что данный кластер содержит хотя бы один дефектный сектор), то запись содержит специальное значение $FF7_{16}$ для FAT-12 или $FFF7_{16}$ для FAT-16.

Номер первого кластера файла хранится в записи каталога, а остальные кластеры можно последовательно определить по записям таблицы FAT.

В каждом каталоге, кроме корневого, первые две записи содержат специальные имена: имя «**.**» означает сам данный каталог, имя «**..**» – родительский каталог.

Файловые системы и управление данными в UNIX

Архитектура файловой системы UNIX

Здесь рассматривается классическая файловая система UNIX, называемая иногда системой ***s5fs*** и поддерживаемая всеми версиями UNIX.

1.1.1.1. Жесткие и символические связи

Структуру каталогов файловой системы UNIX называют иногда сетевой, чтобы подчеркнуть ее отличие от строго иерархической (древесной) структуры каталогов таких систем, как, например, FAT. Отличие это заключается в понятиях *жестких и символических связей* файла.

Жесткая связь означает связь между именем файла и самим файлом. Особенность UNIX в том, что любой файл может иметь несколько (точнее, неограниченное количество) жестких связей, т.е. неограниченное количество имен. Это могут быть разные имена в одном каталоге или даже имена, хранящиеся в разных каталогах одного дискового тома.

Есть ли какая-нибудь польза от нескольких имен одного файла? Безусловно, есть. Предположим, пользователь часто использует какую-либо системную программу или файл данных, лежащий где-то глубоко в одной из ветвей дерева каталогов. Вместо того чтобы каждый раз указывать длинный путь к нужному файлу, пользователь может просто создать новую жесткую связь, т.е. дать файлу удобное имя и поместить это имя в свой личный каталог. UNIX предоставляет для этого команду ***link***, которая создает новое имя для указанного файла.

Что произойдет, если один из пользователей удалит имя файла из каталога? Произойдет только обрыв одной из жестких связей. Пока у данного файла остаются другие имена, файл продолжает существовать. Только после того, как удалены все имена файла, система понимает, что файл перестал быть доступен кому-либо, и удаляет сам файл.

Все жесткие связи (имена) одного файла абсолютно равноправны, среди них нельзя выделить какое-то «основное» имя.

Символическая связь представляет собой файл, который содержит только полное имя другого файла. Важно при этом то, что файл помечен в системе именно как символическая связь, а не просто текстовый файл, случайно хранящий имя файла. Когда

файл символической связи используется как аргумент системной команды или функции, UNIX автоматически подставляет вместо него тот файл, на который указывает связь.

Можно кратко сказать, что жесткая связь указывает на сам файл, а символическая – на имя файла. Оба типа связей проиллюстрированы на рисунке 5.

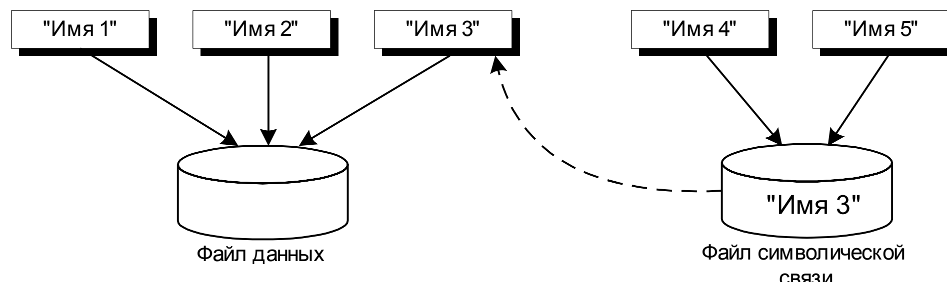

Рисунок 5 - Жесткие и символические связи в файловой системе UNIX

В примере показан файл данных, для которого имеются три жесткие связи, т.е. три имени в каталогах системы, обозначенные как «Имя 1», «Имя 2» и «Имя 3». Кроме того, в системе имеется файл типа «символическая связь», который содержит одно из имен файла данных. Файл символической связи, как и любой другой файл, доступен по имени и в данном случае имеет два имени (две жестких связи): «Имя 4» и «Имя 5». Таким образом, использование любого из пяти имен в качестве, например, имени открываемого файла приведет к открытию одного и того же файла.

Предположим, администратор системы решил заменить некоторый файл его более свежей версией, оставив то же самое имя файла. Если некоторые пользователи хранили жесткие связи на прежнюю версию, то они так и будут ею пользоваться, пока явно не удалят ее имя и не создадут связи на новую версию. Если же пользователь хранил символическую связь, то она теперь будет указывать на новую версию.

Монтируемые тома

В UNIX нет понятия «буква диска», подобно буквам ***A:***, ***C:*** и т.д., используемым в MS-DOS и в Windows. В системе может быть несколько дисковых томов, но, прежде чем получить доступ к файловой системе любого диска, кроме основного, пользователь должен выполнить операцию *монтирования диска*. Она заключается в том, что данный диск отображается на какой-либо из каталогов основного тома. Как правило, для этого используются пустые подкаталоги каталога */mount* или */mnt*.

Если представить файловую систему на дисковом томе в виде дерева, то монтирование тома – это как бы «прививка» одного дерева к какому-либо месту на другом,

основном дереве. В отличие от этого, MS-DOS и Windows допускают использование нескольких отдельных деревьев.

Типы и атрибуты файлов

Для каждого файла в UNIX хранится его тип, который при выдаче каталога обозначается одним из следующих символов:

- \- – обычный файл, т.е. файл, содержащий данные;
- *d* – каталог;
- *c* – символьный специальный файл, т.е., на самом деле, символьное устройство;
- *b* – блочный специальный файл;
- *l* – символическая связь;

Особенностью UNIX является то, что работа с разными типами объектов, перечисленными выше (файлами, устройствами) организуется с использованием одного и того же набора функций файлового ввода/вывода.

К числу атрибутов, описывающих файл, относятся его размер в байтах, число жестких связей и три «временн*ы*х штампа»: дата/время последнего доступа к файлу, последней модификации файла, последней модификации атрибутов файла. Эту последнюю величину часто называют неточно «датой создания файла».

1.1.1.2. Управление доступом

Для каждого файла (в том числе каталога, специального файла) определены такие понятия, как *владелец* (один из пользователей системы) и *группа-владелец*. Их числовые идентификаторы (называемые, соответственно, UID и GID) хранятся вместе с другими атрибутами файла. Полные имена, пароли и другие характеристики пользователей и групп хранятся в отдельном системном файле. Владельцем файла обычно является тот пользователь, который создал этот файл.

Кроме того, для файла задаются *атрибуты защиты*, хранящиеся в виде 9 бит, которые определяют допустимость трех основных видов доступа – на чтение, на запись и на исполнение – для владельца, для членов группы-владельца и для прочих пользователей.

При отображении каталога с помощью команды ***ls –l*** эти атрибуты показываются в виде 9 букв или прочерков, например:

```
r w x  r – x  – – x
```

В приведенном примере показано, что сам владелец файла имеет все права (r – чтение, w – запись, x – исполнение), члены группы-владельца могут читать файл и запускать

на исполнение (если этот файл содержит программу), всем прочим разрешено только исполнение.

Привилегированный пользователь (администратор системы) всегда имеет полный доступ ко всем файлам.

В том случае, если файл является каталогом, права доступа на чтение и на исполнение понимаются несколько иначе. Право на чтение каталога позволяет получить имена файлов, хранящиеся в данном каталоге. Право на исполнение каталога означает возможность читать атрибуты файлов каталога, использовать эти файлы, а также право сделать данный каталог текущим. Возможны интересные ситуации: если текущий пользователь не имеет права на чтение каталога, но имеет право на его «исполнение», то он не может узнать имена файлов, хранящихся в каталоге; однако, если он все же каким-то образом узнал имя одного из файлов, то может открыть этот файл или запустить на исполнение (если этому не препятствуют атрибуты доступа самого файла).

Чтобы удалить файл, не нужно иметь никаких прав доступа к самому файлу, но необходимо право на запись в соответствующий каталог (потому что удаление файла есть изменение не файла, а каталога). Впрочем, в современных версиях UNIX добавлен еще один битовый атрибут, при установке которого удаление разрешено только владельцу файла или пользователю, имеющему право записи в файл.

Изменение атрибутов защиты, а также смена владельца файла, могут быть выполнены только самим владельцем либо привилегированным пользователем.

Структуры данных файловой системы UNIX

Дисковый том UNIX состоит из следующих основных областей, показанных на рисунке 6:

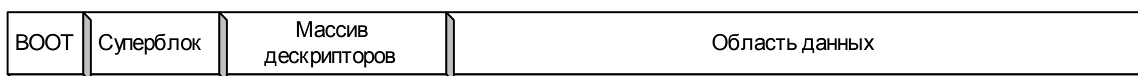

Рисунок 6 - Структура диска в файловой системе UNIX (s5fs)

- блок начальной загрузки (BOOT-сектор); его структура определяется не UNIX, а архитектурой используемого компьютера;
- суперблок – содержит основные сведения о дисковом томе в целом (размер логического блока и количество блоков, размеры основных областей, тип файловой системы, возможные режимы доступа), а также данные о свободном месте на диске;

- массив индексных дескрипторов, каждый из которых содержит полные сведения об одном из файлов, хранящихся на диске (кроме имени этого файла);
- область данных, состоящая из логических блоков (кластеров), которые используются для хранения файлов и каталогов (в UNIX используется сегментированное размещение файлов).

В отличие от системы FAT, где основные сведения о файле содержались в каталожной записи, UNIX использует более изощренную схему.

Запись каталога не содержит *никаких данных о файле, кроме только* имени файла и номера индексного дескриптора этого файла.

В ранних версиях UNIX каждая запись имела фиксированную длину 16 байт, из которых 14 использовались для имени и 2 для номера. В более современных версиях запись имеет переменный размер, что позволяет использовать длинные имена файлов.

Как и в системе FAT, в каждом каталоге первые две записи содержат специальные имена «**..**» (ссылка на родительский каталог) и «**.**» (ссылка на данный каталог).

Нулевое значение номера соответствует удаленной записи каталога.

Все сведения о файле, кроме имени, содержатся в его *индексном дескрипторе* (*inode*). Такая схема делает возможными жесткие связи, описанные выше: любое количество записей из одного каталога или из разных каталогов может относиться к одному и тому же файлу. Для этого надо только, чтобы эти записи содержали один и тот же номер inode.

Индексные дескрипторы хранятся в массиве, занимающем отдельную область диска. Размер этого массива задается при форматировании, этот размер определяет максимальное количество файлов, которое можно разместить на данном томе.

Дескриптор содержит, прежде всего, счетчик жестких связей файла, т.е. число каталожных записей, ссылающихся на данный дескриптор. Этот счетчик изменяется при создании и удалении связей, его нулевое значение говорит о том, что файл перестал быть доступным и должен быть удален.

В дескрипторе содержатся тип и атрибуты файла, описанные выше. Наконец, здесь же содержатся данные о размещении файла, имеющие весьма оригинальную структуру, показанную на рисунке 7.

Размещение блоков файла задается массивом из 13 (в некоторых версиях 14) элементов, каждый из которых может содержать номер блока в области данных. Пусть, для определенности, блок равен 1 Кб, а его номер занимает 4 байта (обе эти величины зависят от версии файловой системы). Первые 10 элементов массива содержат номера первых 10

блоков от начала файла. Если размер файла превышает 10 Кб, то в ход идет 11-й элемент массива. Он содержит номер *косвенного блока* – такого блока в области данных, который содержит номера следующих 256 блоков файла. Таким образом, использование косвенного блока позволяет работать с файлами размером до 266 Кб, используя для этого один дополнительный блок. Если файл превышает 266 Кб, то в 12-ом элементе массива содержится номер *вторичного косвенного блока*, который содержит до 256 номеров косвенных блоков, каждый из которых… Ну, вы поняли. Наконец, для очень больших файлов будет задействован 13-й элемент массива, содержащий номер *третичного косвенного блока*, указывающего на 256 вторичных косвенных.

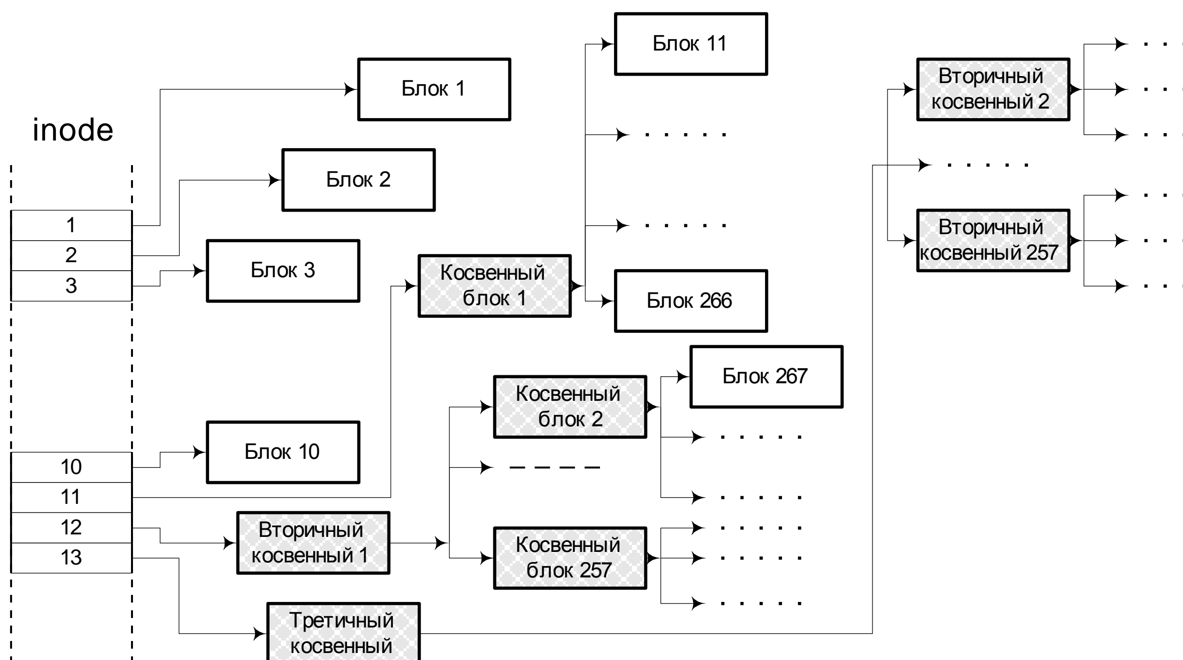

Рисунок 7 - Информация о размещении файла

Недостатком описанной схемы является то, что доступ к большим файлам требует значительно больше времени, чем к маленьким. Если расположение первых 10 Кб данных файла записано непосредственно в индексном дескрипторе, то для того, чтобы прочитать данные, отстоящие, скажем, на 50 Мб от начала файла, придется сначала прочитать третичный, вторичный и обычный косвенные блоки.

Файловая система NTFS и управление данными в Windows

Особенности файловой системы NTFS

Файловая система NTFS была разработана специально для использования в ОС Windows NT как замена для устаревшей системы FAT. NTFS является основной системой и для новых версий – Windows 2000/XP.

Система NTFS спроектирована как очень мощная многопользовательская файловая система с большим количеством возможностей. Тем не менее, как утверждают разработчики, NTFS обеспечивает более быстрый доступ к данным, чем предельно простая система FAT, если объем диска превышает 600 Мб.

Среди возможностей, отсутствующих в FAT, но реализованных в NTFS, можно назвать следующие.

- Развитые средства защиты данных, предотвращающие возможность несанкционированного доступа к данным и при этом позволяющие весьма детально разграничить права доступа для различных пользователей и групп пользователей.
- Быстрый поиск файлов в больших каталогах.
- Обеспечение целостности данных в случае сбоев или отключения питания, основанное на механизме транзакций. Это означает, что любая операция с файлом рассматривается как неделимое действие (транзакция), которое должно быть либо выполнено до конца, либо не выполнено вовсе. В ходе операции система протоколирует в специальном журнале ход выполнения отдельных этапов транзакции: запись данных, внесение изменений в каталог и т.п. Если транзакция будет прервана на промежуточном этапе, то при следующей загрузке системы информация из журнала позволит «откатить» недовыполненную транзакцию, т.е. отменить выполненные этапы.
- Возможность сжатия данных на уровне отдельных файлов (т.е. на одном дисковом томе могут храниться файлы как в сжатом, так и в несжатом формате).
- Возможность хранения файлов в зашифрованном виде.
- Механизм точек повторного анализа (reparse points), позволяющий для отдельных каталогов задать действия, которые должны выполняться всякий раз, когда система обращается к данному каталогу. В частности, этот механизм позволяет реализовать такие UNIX-подобные возможности, как символические связи и монтирование файловых систем.
- Возможность протоколирования всех изменений, происходящих в файловой системе, таких как создание, изменение и удаление файлов и каталогов.
- Расширяемость системы. Учтя трудный опыт, связанный с попытками модернизации FAT, разработчики NTFS заранее заложили в систему возможность добавления новых, не предусмотренных в настоящее время атрибутов файлов.

Некоторые возможности, заложенные в файловую систему NTFS, даже опережают развитие ОС Windows и пока не могут быть использованы в этой системе.

Структуры дисковых данных

Наиболее важной частью файловой системы на диске является *главная таблица файлов* (MFT, Master File Table). Эта таблица содержит записи обо всех файлах и каталогах, расположенных на данном томе. Размер записи составляет один кластер, но не менее 1 Кб. Если метаданные о файле не помещаются в одной записи, то могут быть использованы дополнительные записи (не обязательно соседние).

После форматирования дискового тома, когда на нем еще нет пользовательских файлов, MFT содержит 16 записей, из которых 11 содержат описания файлов метаданных, а 5 зарезервированы как дополнительные. Список файлов метаданных достаточно интересен.

- Первая запись MFT описывает саму MFT, которая тоже считается файлом. Это отнюдь не формальность, поскольку MFT может, как и прочие файлы, состоять из нескольких сегментов, размещение которых задается в этой записи.
- Копия первых 16 записей MFT, которая хранится как файл где-нибудь в середине диска. Это позволяет восстановить метаданные в случае повреждения основного экземпляра MFT.
- Журнал протоколирования транзакций.
- Файл информации о томе: имя тома, серийный номер, дата форматирования и т.п.
- Файл с перечислением всех атрибутов, используемых для описания файлов на данном томе. Таким образом, список атрибутов не является жестко фиксированным и может быть расширен в последующих версиях NTFS.
- Корневой каталог тома.
- Битовая карта занятости кластеров тома.
- BOOT-сектор. Он по-прежнему является первым сектором тома, но тоже считается файлом.
- Файл, состоящий из всех дефектных кластеров на данном томе. Это дает основания пометить в битовой карте все дефектные кластеры как занятые.
- Файл, содержащий все различные дескрипторы защиты, используемые для файлов и каталогов данного тома.
- Файл, задающий пары прописных / строчных букв для всех языков, поддерживаемых Windows. Такие данные необходимы, поскольку имена файлов могут

содержать буквы обоих типов, но в Windows по традиции регистр букв в именах файлов не различается (в отличие, например, от UNIX).

- Каталог, содержащий еще 4 файла метаданных, добавленных в Windows 2000. К ним относятся:

– файл уникальных 16-байтовых идентификаторов, создаваемых Windows для каждого файла, на который имеется ярлык или OLE-связь; это позволяет автоматически исправить ярлык, если исходный файл был перемещен в другой каталог или даже на другой компьютер в пределах домена сети;

– файл квот дискового пространства, выделяемых каждому пользователю;

– файл точек повторного анализа, установленных для каталогов данного тома;

– файл журнала изменений, происходящих на томе.

Далее, начиная с 17-й позиции MFT, хранятся записи метаданных о файлах и каталогах, размещенных на данном томе.

Система пытается сохранить MFT непрерывной, поскольку это ускоряет обращение ко всем описанным в ней файлам. Для этого система старается по возможности не занимать некоторую область в начале диска под размещение файлов, сохраняя свободное место для роста MFT.

Атрибуты файла

Каждая запись MFT содержит набор атрибутов, который может различаться для разных файлов и каталогов.

Рассмотрим наиболее важные типы атрибутов, используемых в записи о файле.

- *Имя файла*. Этот атрибут всегда резидентен. Допускается несколько атрибутов этого типа, например, «длинное» имя (до 255 символов, включая буквы любого языка) и имя «8 + 3» для того же файла.

- *Стандартная информация*. Это примерно та информация о файле, которая хранилась в записи каталога FAT: размер файла, временные штампы и битовые флаги.

- *Дескриптор защиты*. Он служит для задания прав доступа к данному файлу для различных пользователей и групп, подробнее см. п. **Ошибка! Источник ссылки не найден.**. В новых версиях NTFS запись MFT содержит не сам дескриптор, а ссылку на его место в системном файле. Так получается компактнее, поскольку обычно на диске имеется много файлов с одинаковыми дескрипторами защиты и лучше хранить каждый дескриптор один раз, в специально отведенном для этого файле метаданных.

- *Данные*. Это самое неожиданное при первом знакомстве с NTFS: сами данные файла рассматриваются как один из типов атрибутов этого файла. Следующая неожиданность состоит в том, что атрибут данных небольшого файла может храниться резидентно в составе записи MFT. Напомним, что размер этой записи – от 1 Кб и больше, так что место для данных маленького файла может найтись. Безусловно, резидентное хранение данных позволяет ускорить доступ к ним, поскольку запись MFT так или иначе всегда читается при открытии файла.

При сравнении структуры NTFS с ранее рассмотренной структурой s5fs можно найти некоторую аналогию между таблицей MFT и массивом индексных дескрипторов, содержащих всю информацию о файле в s5fs. При этом NTFS имеет значительно более сложную структуру и предоставляет много дополнительных возможностей.

Защита данных

Средства безопасности в Windows NT/2000/XP представляют собой отдельную подсистему, которая обеспечивает защиту не только файлов, но и других типов системных объектов. Файлы и каталоги NTFS представляют собой наиболее типичные примеры защищаемых объектов.

Как известно, Windows позволяет использовать различные файловые системы, при этом возможности защиты данных определяются архитектурой конкретной файловой системы. Например, если на дисковом томе используется система FAT (где, как нам известно, никаких средств защиты не предусмотрено), то Windows может разве что ограничить доступ ко всему тому, но не к отдельным файлам и каталогам.

Аутентификация пользователя

Важным элементом любой системы защиты данных является процедура входа в систему, при которой выполняется аутентификация пользователя. В Windows NT для вызова диалога входа в систему используется известная «комбинация из трех пальцев» – ***Ctrl+Alt+Del***. Как утверждают разработчики, никакая «троянская» программа не может перехватить обработку этой комбинации и использовать ее с целью коллекционирования паролей.

Система ищет введенное имя пользователя сначала в списке пользователей данного компьютера, а затем и на других компьютерах текущего домена локальной сети. В случае, если имя найдено и пароль совпал, система получает доступ к *учетной записи* (account) данного пользователя.

На основании сведений из учетной записи пользователя система формирует структуру данных, которая называется *маркером доступа* (access token). Маркер содержит

идентификатор пользователя (SID, Security IDentifier), идентификаторы всех групп, в которые включен данный пользователь, а также набор *привилегий*, которыми обладает пользователь.

Привилегиями называются права общего характера, не связанные с конкретными объектами. К числу привилегий, доступных только администратору, относятся, например, права на установку системного времени, на создание новых пользователей, на присвоение чужих файлов.

В дальнейшей работе, когда пользователю должен быть предоставлен доступ к каким-либо защищаемым ресурсам, решение о доступе принимается на основании информации из маркера доступа.

РАЗДЕЛ 4. УПРАВЛЕНИЕ ПРОЦЕССАМИ

Основные задачи управления процессами

Под управлением процессами понимаются процедуры ОС, обеспечивающие запуск системных и прикладных программ, их выполнение и завершение.

В однозадачных ОС управление процессами решает следующие задачи:

- загрузка программы в память, подготовка ее к запуску и запуск на выполнение;
- выполнение системных вызовов процесса;
- обработка ошибок, возникших в ходе выполнения;
- нормальное завершение процесса;
- прекращение процесса в случае ошибки или вмешательства пользователя.

Все эти задачи решаются сравнительно просто.

В многозадачном режиме добавляются значительно более серьезные задачи:

- эффективная реализация параллельного выполнения процессов на единственном процессоре, переключение процессора между процессами;
- выбор очередного процесса для выполнения с учетом заданных приоритетов процессов и статистики использования процессора;
- исключение возможности несанкционированного вмешательства одного процесса в выполнение другого;
- предотвращение или устранение тупиковых ситуаций, возникающих при конкуренции процессов за системные ресурсы;
- обеспечение синхронизации процессов и обмена данными между ними.

Понятия процесса и ресурса

Процесс (иногда называемый «задача») есть работа, производимая последовательным процессором при выполнении программы с ее данными.

Проанализируем это определение. Оно подчеркивает последовательный характер процесса, т.е. выполнение команд в определенном порядке. Термин «задача» мы будем понимать как синоним термина «процесс» (в некоторых ОС эти термины различаются). Далее, процесс – понятие динамическое. Программа – это текст, процесс – выполнение этого текста. Конечно, на практике мы часто говорим: «программа вызывает функцию»,

«программа ждет ввода» и т.п., однако, строго говоря, правильнее было бы «процесс, выполняющий программу, вызывает…».

Другим основополагающим понятием, тесно связанным с управлением процессами, является понятие ***ресурса***. Под ресурсом понимается любой аппаратный или программный объект, который может понадобиться для работы процессов и доступ к которому может при этом вызвать конкуренцию процессов. Говоря упрощенно, ресурс – это нечто дефицитное в вычислительной системе. К важнейшим ресурсам любой системы относятся процессор (точнее сказать, процессорное время), основная память, периферийные устройства, файлы.

Многозадачная ОС управляет доступом процессов к ресурсам. В некоторых случаях система временно закрепляет ресурс за одним процессом, отказывая другим процессам в доступе или заставляя их ждать освобождения ресурса. В других случаях оказывается возможным совместный доступ нескольких процессов к одному ресурсу.

Квазипараллельное выполнение процессов

С точки зрения внешнего наблюдателя, в хорошей многозадачной ОС происходит одновременная, параллельная работа нескольких процессов. Однако понятно, что эта одновременность кажущаяся. На самом деле, если в системе работает лишь один процессор, то в каждый момент времени он выполняет команды, относящиеся только к одному из имеющихся процессов. Иллюзия параллельности создается за счет того, что процессы сменяют друг друга через малые интервалы времени, которые человек-наблюдатель не в силах отследить. Подобная организация работы называется ***квазипараллельным*** выполнением процессов.

Состояния процесса

Любой процесс в многозадачной ОС многократно испытывает переход из одного *состояния* в другое.

Основных состояний всего три.

- *Работа* (running) – в этом состоянии находится процесс, программу которого в данный момент выполняет процессор. Работающий процесс иногда удобно называть также *текущим* процессом.
- *Готовность* (ready) – состояние, их которого процесс может быть переведен в состояние работы, как только это сочтет нужным сделать ОС.

- *Блокировка* или, что то же самое, *сон* (sleeping, waiting) – состояние, в котором процесс не может продолжать выполнение, пока не произойдет некоторое *внешнее* по отношению к процессу событие.

Первые два состояния часто объединяют понятием *активного* состояния процесса.

Готовый к выполнению процесс не выполняется только потому, что есть другие не менее готовые процессы, по мнению системы более достойные занимать сейчас процессорное время. В каждый момент времени выбор одного из готовых процессов на роль работающего определяется логикой работы ОС. Этот выбор должен обеспечивать эффективную квазипараллельную работу готовых процессов. Как решается эта задача – будет рассмотрено ниже.

В отличие от этого, спящий процесс – это всегда процесс, ожидающий некоторого конкретного события. Спящий процесс не сможет заработать, даже если процессор вдруг окажется свободным. Такой процесс, в соответствии со своей собственной логикой, ждет чего-то, что должно произойти.

Чего он может ждать? Ну, например:

- завершения начатой операции синхронного ввода/вывода (т.е., например, процесс ждет нажатия клавиши *Enter* или окончания записи на диск);
- освобождения запрошенного у системы ресурса (например, дополнительной области памяти или открытого файла);
- истечения заданного интервала времени («*посплю-ка я минут десять!*») или достижения заданного момента времени («*разбудите меня ровно в полночь!*») (в обоих случаях процесс ждет сигнала от запрограммированного таймера);
- сигнала на продолжение действий от другого, взаимосвязанного процесса;
- сообщения от системы о необходимости выполнить определенные действия (например, перерисовать содержимое окна).

В любом из названных (и многих неназванных) случаев должно произойти некоторое событие, источник которого лежит *вне* данного процесса.

На рисунке 8 показаны основные состояния процесса и переходы между ними. Этот рисунок кочует из книги в книгу, поскольку он действительно наглядно отражает самую суть работы многозадачных систем.

Рассмотрим возможные переходы между состояниями процесса, показанные на рисунке стрелками.

Переход *Работа* → *Сон* представляет собой блокировку процесса, которая может произойти при вызове блокирующей системной функции.

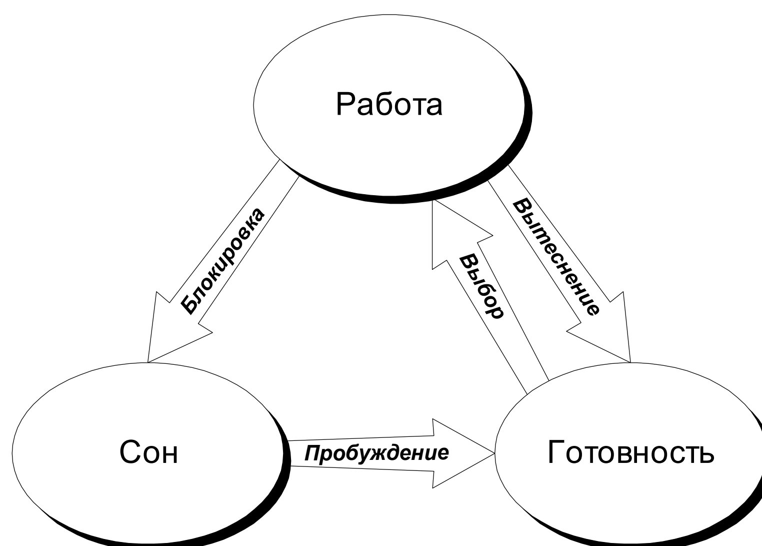

Рисунок 8 - Основные состояния процесса

Переход *Сон* → *Готовность* – это пробуждение процесса, оно выполняется системой при возникновении соответствующего условия.

Переход *Работа* → *Готовность* ранее не рассматривался. Он называется *вытеснением* процесса и выполняется системой, когда она принимает решение о смене текущего процесса.

Для обратного перехода *Готовность* → *Работа* нет общепринятого термина. Будем называть его *выбором* процесса для выполнения. Отметим, что этот переход почти всегда связан либо с блокировкой, либо с вытеснением прежнего текущего процесса.

Вытесняющая и невытесняющая многозадачность

Все многозадачные ОС можно разделить на два класса, различающиеся по способам организации переключения процессов.

В системах *с невытесняющей диспетчеризацией* (non-preemptive multitasking) работа любого процесса может быть прервана только «по инициативе самого процесса».

Использование невытесняющей диспетчеризации позволило разработать достаточно впечатляющие примеры ОС, из которых наиболее известной является Windows версий 1, 2 и 3. Обычно в таких системах большая часть процессов находится в спящем состоянии,

ожидая, пока пользователь обратится к соответствующему приложению. Для пользователя это выглядит совершенно естественно.

Главный недостаток многозадачности невытесняющего типа заключается в том, что любой процесс в принципе имеет возможность полностью и надолго захватить процессор.

Более сложным и совершенным типом многозадачных ОС являются системы с *вытесняющей диспетчеризацией* процессов (preemptive multitasking). Их отличие заключается в том, что планировщик (программ ОС, планирующая очередность выполнения процессов) вступает в работу не только когда текущий процесс передал управление, но и в следующих двух случаях (или хотя бы в одном из них):

- когда активизируется (т.е. пробуждается или запускается) процесс, обладающий более высоким приоритетом, чем текущий;
- когда истекает *квант времени,* выделенный планировщиком для текущего процесса.

Принципиальной чертой систем с вытесняющей диспетчеризацией является то, что текущий процесс может быть прерван и вытеснен практически в любой точке своей программы. Это заметно усложняет реализацию таких систем по сравнению с невытесняющими системами, где смена текущего процесса может произойти только в момент вызова системной функции.

Дескриптор и контекст процесса

С каждым процессом связаны описывающие его данные в основной памяти, необходимые ОС для поддержки выполнения процесса. Все эти данные можно разбить на две большие структуры: дескриптор процесса и контекст процесса.

Дескриптор процесса включает в себя все те данные о процессе, которые могут понадобиться ОС при различных состояниях процесса. В число элементов дескриптора могут входить, например, идентификатор процесса (некое условное число, обозначающее данный процесс); текущее состояние процесса; его приоритет; владелец процесса (т.е. идентификатор пользователя, запустившего процесс); статистика затраченного процессом общего и процессорного времени; указатель местоположения контекста процесса и др. Дескрипторы всех процессов, существующих в системе, собраны в таблицу процессов.

Контекст процесса включает данные, необходимые только для текущего процесса. Суда относятся, прежде всего, значения всех регистров процессора, включая указатель текущей команды; таблица файлов, открытых процессом; указатели на области памяти, которые должен занимать процесс при его выполнении; значения системных переменных,

используемых процессом (например, текущий диск и каталог, информация о последней ошибке при выполнении системных функций); другие системные флаги и режимы, которые могут иметь разные значения для разных процессов.

Точный состав дескриптора и контекста сильно зависят от конкретной ОС.

При переключении текущего процесса система должна каждый раз переключать и текущий контекст, т.е. сохранять в своей памяти или на диске контекст предыдущего выполнявшегося процесса и восстанавливать ранее сохраненный контекст того процесса, который будет выполняться.

Дисциплины диспетчеризации и приоритеты процессов

Когда планировщик процессов получает управление, его основной задачей является выбор следующего процесса, который должен получить управление. Алгоритмы, лежащие в основе этого выбора, определяют *дисциплину диспетчеризации*, принятую в данной ОС.

Одной из самых очевидных дисциплин является *простая круговая диспетчеризация* (round robin scheduling). Ее суть в следующем. Все активные процессы считаются равноправными и образуют круговую очередь. Каждый процесс получает от системы квант времени, по истечении которого планировщик выбирает для выполнения следующий процесс из очереди. Таким образом, если все процессы остаются активными, то система обеспечивает их равномерное продвижение, имитирующее параллельное выполнение всех процессов. Если текущий процесс блокируется, он выпадает из круга и попадает в список спящих процессов. Когда система активизирует один из спящих процессов, он включается в круговую очередь.

В некотором смысле противоположной дисциплиной является *фоново-оперативная диспетчеризация* (foreground/background scheduling) – одна из самых старых форм организации многозадачной работы. В простейшем случае она включает два процесса: фоновый процесс и оперативный процесс (процесс переднего плана). Фоновый процесс выполняется только тогда, когда спит оперативный процесс. При активизации оперативного процесса происходит немедленное вытеснение фонового, т.е. оперативный процесс имеет более высокий абсолютный приоритет. Обычно при такой дисциплине предполагается, что активизация оперативного процесса не потребует много процессорного времени, так что выполнение фонового процесса будет скоро возобновлено. Одним из примеров эффективного использования фоново-оперативной диспетчеризации является так называемая «фоновая печать», которую позволяет выполнить даже однозадачная MS-DOS. При этом процесс вывода файла на принтер рассматривается как процесс переднего плана, а обычная

диалоговая работа с ОС – как фоновый процесс. Поскольку обслуживание прерываний от принтера занимает лишь доли процента процессорного времени, пользователь не ощущает никакого замедления работы.

Между описанными двумя крайностями лежит большое разнообразие дисциплин *приоритетной диспетчеризации*. Все они основаны на приписывании каждому процессу при его создании некоторого числа – *приоритета*. Более высокий приоритет должен давать процессу определенные преимущества перед низкоприоритетными процессами при работе планировщика.

Изоляция процессов и их взаимодействие

Одна из важнейших целей, которые ставятся при разработке многозадачных систем, заключается в том, чтобы разные процессы, одновременно работающие в системе, были как можно лучше изолированы друг от друга. Это означает, что процессы (в идеале) не должны ничего знать даже о существовании друг друга.

Изоляция процессов, во-первых, является необходимым условием надежности и безопасности многозадачной системы. Один процесс не должен иметь возможности вмешаться в работу другого или получить доступ к его данным, ни по случайной ошибке, ни намеренно.

Во-вторых, проектирование и отладка программ чрезвычайно усложнились бы, если бы программист должен был учитывать непредсказуемое влияние других процессов.

С другой стороны, есть ситуации, когда взаимодействие необходимо. Процессы могут совместно обрабатывать общие данные, обмениваться сообщениями, ждать ответа и т.п. Система должна предоставлять в распоряжение процессов средства взаимодействия. Это не противоречит тому, что выше было сказано об изоляции процессов. Чтобы взаимодействие не привело к полному хаосу, оно должно выполняться только с помощью тех хорошо продуманных средств, которые предоставляет процессам ОС. За пределами этих средств действует изоляция процессов.

Понятие взаимодействия процессов включает в себя несколько видов взаимодействия, основными из которых являются:

- синхронизация процессов, т.е., упрощенно говоря, ожидание одним процессом каких-либо событий, связанных с работой других процессов;
- обмен данными между процессами.

Проблема взаимного исключения процессов

Серьезная проблема возникает в ситуации, когда два (или более) процесса одновременно пытаются работать с общими для них данными, причем хотя бы один процесс изменяет значение этих данных.

Проблему часто поясняют на таком, немного условном, примере. Пусть имеется система резервирования авиабилетов, в которой одновременно работают два процесса. Процесс А обеспечивает продажу билетов, процесс В – возврат билетов. Не углубляясь в детали, будем считать, что оба процесса работают с переменной **N** – числом оставшихся билетов, причем соответствующие фрагменты программ на псевдокоде выглядят примерно так:

Процесс А:	***Процесс В:***
. . .	**. . .**
R1 := N;	**R2 := N;**
R1 := R1 - 1;	**R2 := R2 + 1;**
N := R1;	**N := R2;**
. . .	**. . .**

Представим себе теперь, что при квазипараллельной реализации процессов в ходе выполнения этих трех операторов происходит переключение процессов. В результате, в зависимости от непредсказуемых случайностей, порядок выполнения операторов может оказаться различным, например:

1) **R1 := N; R2 := N; R2 := R2 + 1; N := R2; R1 := R1 - 1; N := R1;**
2) **R2 := N; R2 := R2 + 1; R1 := N; R1 := R1 - 1; N := R1; N := R2;**
3) **R1 := N; R1 := R1 - 1; N := R1; R2 := N; R2 := R2 + 1; N := R2;**

Ну и что? А то, что в случае 1 значение **N** в результате окажется уменьшенным на 1, в случае 2 – увеличенным на 1, и только в случае 3 значение **N**, как и положено, не изменится.

Можно привести менее экзотические примеры.

- Если два процесса одновременно пытаются отредактировать одну и ту же запись базы данных, то в результате разные поля одной записи могут оказаться несогласованными.

- Если один процесс добавляет сообщение в очередь, а другой в это время пытается взять сообщение из очереди для обработки, то он может прочесть не полностью сформированное сообщение.

- Если два процесса одновременно пытаются обратиться к диску, каждый со своим запросом, то что именно каждый из них в результате прочтет или запишет на диск – сказать трудно.

Ситуация понятна: нельзя разрешать двум процессам одновременно обращаться к одним и тем же данным, если при этом происходит изменение этих данных.

Для более четкого описания ситуации было введено понятие *критической секции*.

Критической секцией процесса по отношению к некоторому ресурсу называется такой участок программы процесса, при прохождении которого необходимо, чтобы никакой другой процесс не находился в *своей* критической секции по отношению к *тому же* ресурсу.

В примере с билетами приведенные три оператора в каждом из процессов составляют критическую секцию этого процесса по отношению к общей переменной **N**. Алгоритм работы каждого процесса в отдельности правилен, но правильная работа двух процессов в совокупности может быть гарантирована, только если они не затронут одновременно каждый свою критическую секцию.

Для решения этой проблемы удалось найти решение, согласно которому, должны выполняться следующие условия:

- в любой момент времени не более, чем один процесс может находиться в критической секции;

- если критическая секция свободна, то процесс может беспрепятственно войти в нее;

- все процессы равноправны.

Проблема тупиков

Группа процессов находится в тупиковой ситуации, если каждый процесс из группы ожидает события, которое может вызвать только другой процесс из той же группы.

Рассмотрим такой пример. Пусть каждый из процессов A и B собирается работать с двумя файлами, F1 и F2, причем не намерен разделять эти файлы с другим процессом. Программы же процессов слегка различаются, а именно:

Процесс А:	***Процесс В:***
. . .	**. . .**
Открыть(F1);	**Открыть(F2);**
Открыть(F2);	**Открыть(F1);**
(работа процесса А с файлами);	**(работа процесса В с файлами);**
Закрыть(F1);	**Закрыть(F1);**
Закрыть(F2);	**Закрыть(F2);**
. . .	**. . .**

В этой ситуации все может пройти благополучно. Пусть, например, процесс А успеет открыть оба файла к тому моменту, когда процесс В попытается открыть F2. Эта попытка не увенчается успехом: процесс В либо будет заблокирован до освобождения файлов, либо получит сообщение об ошибке при открытии файла и, если процесс умный, через какое-то время попытается еще раз. В конце концов оба процесса получат требуемые ресурсы (в данном случае открытые файлы), хотя и не оба сразу.

Совсем иное будет дело, если А успеет открыть только F1, после чего В откроет F2. Тут-то и получится тупик. Процесс А хочет открыть файл F2, но не сможет этого сделать раньше, чем В закроет этот файл. Но В не закроет F2 до того, как сумеет открыть файл F1, который занят процессом А. Каждый из процессов захватил один из ресурсов и не собирается его отдавать раньше, чем получит другой. Ситуация «двух баранов на мосту».

Подчеркнем: тупик – это не просто блокировка процесса, когда необходимый ему ресурс занят. Занят – ну и что, со временем, авось, освободится. Тупик – это *взаимная блокировка*, из которой нет выхода.

Все известные способы борьбы с тупиками можно разделить на три группы:

- исключение возможности тупиков путем анализа исходного текста программ. Определить по тексту программ процессов, могут ли они зайти в тупик – сложная задача;
- предотвращение возникновения тупиков при работе ОС. Любой запрос процесса на выделение ему дополнительных ресурсов должен удовлетворяться только в том случае, если состояние, в которое перейдет система после этого выделения, будет безопасным;
- ликвидация возникших тупиков через снятия процесса с решения.

РАЗДЕЛ 5. УПРАВЛЕНИЕ ПАМЯТЬЮ

Основные задачи управления памятью

Основная память (она же ОЗУ) является важнейшим ресурсом, эффективное использование которого решающим образом влияет на общую производительность системы.

К основным задачам, которые должна решать подсистема управления памятью многозадачной ОС, относятся.

- выделение памяти для процесса пользователя при его запуске и освобождение этой памяти при завершении процесса;
- обеспечение настройки запускаемой программы на выделенные адреса памяти;
- управление выделенными областями памяти по запросам программы пользователя (например, освобождение части памяти перед запуском порожденного процесса).
- предоставление процессам возможностей получения и освобождения дополнительных областей памяти в ходе работы;
- эффективное использование ограниченного объема основной памяти для удовлетворения нужд всех работающих процессов, в том числе с использованием дисков как расширения памяти;
- изоляция памяти процессов, исключающая случайное или намеренное несанкционированное обращение одного процесса к областям памяти, занимаемым другим процессом;
- предоставление процессам возможности обмена данными через общие области памяти.

Виртуальные и физические адреса

Понятие «адрес памяти» может рассматриваться с двух точек зрения. С одной стороны, при написании любой программы ее автор либо явно указывает, по каким адресам должны размещаться переменные и команды (так бывает при программировании на языке ассемблера), либо присвоение конкретных адресов доверяется системе программирования. Те адреса памяти, которые записаны в программе, принято называть *виртуальными адресами*.

С другой стороны, каждой ячейке памяти компьютера соответствует ее адрес, который должен помещаться на шину адреса при каждом обращении к ячейке. Эти адреса называются *физическими*.

В ЭВМ первого поколения не делалось различия между виртуальными и физическими адресами: в программе требовалось указывать физические адреса. Это означало, что такая программа могла правильно работать, только если сама программа и все ее данные при каждом запуске (и на любом компьютере) должны были размещаться по одним и тем же физическим адресам. Такой подход стал крайне неудобным, как только была поставлена задача передать распределение памяти под управление ОС.

В настоящее время программирование в физических адресах может использоваться лишь в очень специальных случаях. Как правило, ни программист, пишущий программу, ни компилятор, транслирующий ее в машинные коды, не должны рассчитывать на использование конкретных физических адресов.

Но тогда возникает вопрос, когда и каким образом должен происходить переход от виртуальных адресов к физическим.

Есть два принципиально разных ответа на этот вопрос.

Замена виртуальных адресов на физические может быть выполнена только программным путем. Преобразование в физические адреса выполняется при выборке каждой команды из памяти, при обращении к ячейкам данных – т.е. при каждом использовании адреса.

Распределение памяти без использования виртуальных адресов

Распределение с фиксированными разделами

При этом способе распределения памяти администратор системы заранее, при установке ОС, выполняет разбиение всей имеющейся памяти на несколько разделов. Как правило, формируются разделы разных размеров. Допускается также определение большого раздела как суммы нескольких примыкающих друг к другу меньших разделов, как показано на рисунке 9.

Очевидно, что память используется не слишком эффективно. Вполне возможна ситуация, когда к некоторым разделам выстроилась очередь программ, а другие разделы в это время пустуют.

Кроме того, количество одновременно загруженных программ не может превышать числа разделов.

Распределение с динамическими разделами

При такой организации памяти никакого предварительного разбиения не делается. Вся имеющаяся память рассматривается как единое пространство, в котором размещаются загруженные программы. Когда возникает необходимость запустить еще одну программу, система выбирает свободный фрагмент памяти достаточного размера и выделяет его в качестве «динамического раздела» для данной программы. Если не удается найти достаточно большой непрерывный участок памяти, то самым простым решением будет подождать с запуском новой программы, пока не завершится одна из работающих программ.

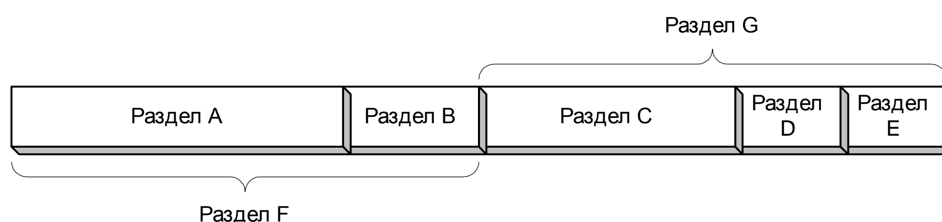

Рисунок 9 - Фиксированные разделы

Вообще говоря, распределение с динамическими разделами позволяет более эффективно использовать память, чем при использовании фиксированных разделов. Однако при этом возникает проблема, которая уже встречалась нам совсем в другой ситуации, в связи с непрерывным размещением файлов на диске. Речь идет о *фрагментации*, т.е. о разбиении свободной памяти на большое число маленьких фрагментов, которые не удается использовать для загрузки крупной программы, хотя суммарный объем свободной памяти остается достаточно большим. Фрагментация является неизбежным следствием того, что память выделяется и освобождается разделами различной длины, причем в произвольном порядке. Но если для файлов можно было время от времени выполнять дефрагментацию, перемещая все файлы ближе к началу диска, то для работающих программ это весьма затруднительно, поскольку перемещение программы нарушило бы настройку адресов, выполненную при ее загрузке.

Сегментная организация памяти

Перейдем теперь к рассмотрению способов организации памяти, которые базируются на использовании виртуальных адресов, аппаратно преобразуемых в физические при выполнении команд. Два основных вида организации виртуальной памяти – *сегментная и страничная организация*.

При сегментной организации вся виртуальная память, используемая программой, разбивается на части, называемые *сегментами*. Это разбиение выполняется либо самим программистом (если он программирует на языке ассемблера), либо компилятором используемого языка программирования. Размеры сегментов могут быть различными, но в пределах максимального размера, используемого в данной архитектуре. Разбиение обычно производится на логически осмысленные части, такие, как сегмент данных, сегмент кода, сегмент стека и т.п. Большая программа может содержать несколько сегментов одного типа, например, несколько сегментов кода или данных.

Таким образом, при сегментной организации у программы нет единого линейного адресного пространства. Виртуальный адрес состоит из двух частей: *селектора сегмента* и *смещения* от начала сегмента.

Селектор сегмента представляет некоторое число, которое обычно является индексом в таблице сегментов данного процесса. Такая таблица содержит для каждого сегмента его размер, режим доступа (только чтение или возможна запись), флаг присутствия сегмента в памяти. Если сегмент находится в памяти, то в таблице хранится его базовый адрес (адрес физической памяти, соответствующий началу сегмента). Отсутствие сегмента означает, что его данные временно вытеснены на диск и хранятся в *файле подкачки* (swap file).

При каждом обращении к виртуальному адресу аппаратными средствами выполняется преобразование пары «сегмент : смещение» в физический адрес. Упрощенная схема такого преобразования показана на рисунке 10.

Селектор сегмента используется для доступа к соответствующей записи таблицы сегментов. Если данный сегмент присутствует в памяти, то его базовый адрес, прочитанный в таблице, складывается со смещением из виртуального адреса. Результат сложения представляет собой физический адрес, по которому и происходит обращение к памяти.

Если сегмент отсутствует в памяти, то происходит прерывание. Обрабатывая его, система должна подгрузить сегмент с диска на свободное место в памяти, записать его базовый адрес в таблицу сегментов и затем повторить команду, вызвавшую прерывание.

Но откуда возьмется свободное место в памяти? По всей вероятности, системе придется для этого убрать из памяти какой-то другой сегмент, принадлежащий либо к этому же, либо к иному процессу. Копия вытесняемого сегмента должна остаться в файле подкачки. Чтобы избежать лишней работы, в каждой записи таблицы хранится флаг, отмечающий, является ли сегмент в памяти «чистым» или «грязным», т.е. совпадает ли его содержимое с дисковой копией или же оно было изменено в памяти после последней

загрузки с диска. «Грязный» сегмент должен быть сохранен на диске, для «чистого» сохранение не требуется. Если сегмент определен как доступный только для чтения, то он заведомо «чистый».

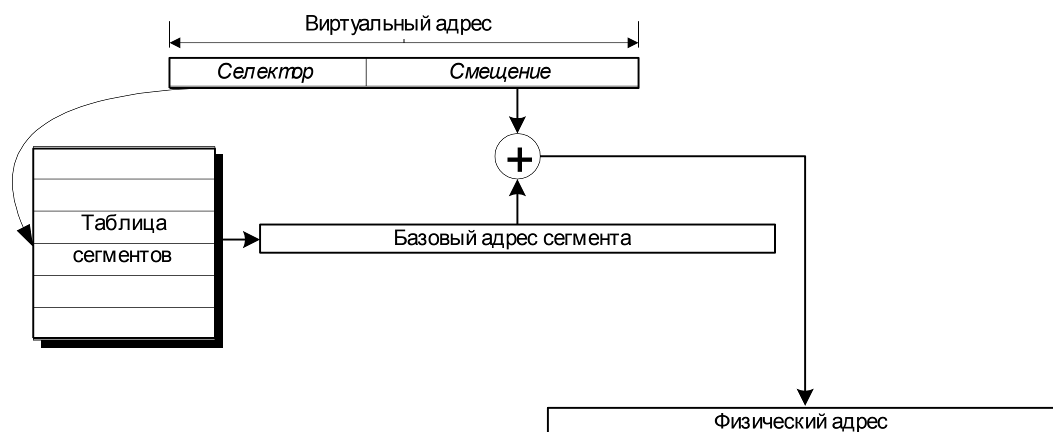

Рисунок 10 - Преобразование сегментного адреса

Поскольку сегменты имеют различные размеры, то в ходе работы системы, сопровождающейся многократной загрузкой и выгрузкой сегментов, возникает эффект фрагментации памяти. Во всех случаях причиной фрагментации является многократное занятие и освобождение областей различного размера.

Для борьбы с фрагментацией можно время от времени производить дефрагментацию, т.е. перемещение всех сегментов, находящихся в памяти, на новые места, без «дырок» в памяти между сегментами.

Страничная организация памяти

Эта форма организации виртуальной памяти во многом похожа на сегментную. Основные различия заключаются в том, что все страницы, в отличие от сегментов, имеют одинаковые размеры, а разбиение виртуального адресного пространства процесса на страницы выполняется системой автоматически. Типичный размер страницы – несколько килобайт. Для процессоров Pentium, например, страница равна 4 Кб.

Все виртуальные адреса одного процесса относятся к единому линейному пространству, Проще сказать, виртуальный адрес выражается одним числом, от 0 до некоторого максимума. Старшие разряды двоичного представления этого адреса определяют номер виртуальной страницы, а младшие разряды – смещение от начала страницы. Например, для страниц по 4 Кб смещение занимает 12 младших разрядов адреса.

Физическая память также считается разбитой на части, размеры которых совпадают с размером виртуальной страницы. Эти части называются *физическими страницами* или

страничными кадрами (page frames). Таблица страниц процесса по структуре похожа на таблицу сегментов. Для каждой виртуальной страницы она содержит режим доступа, флаг присутствия страницы в памяти, номер страничного кадра, флаг чистоты. Если страница отсутствует в памяти, ее данные сохраняются в файле подкачки, который в этом случае чаще называют *страничным файлом* (page file).

Простейший вариант схемы преобразования виртуального страничного адреса в физический адрес показан на рисунке 11.

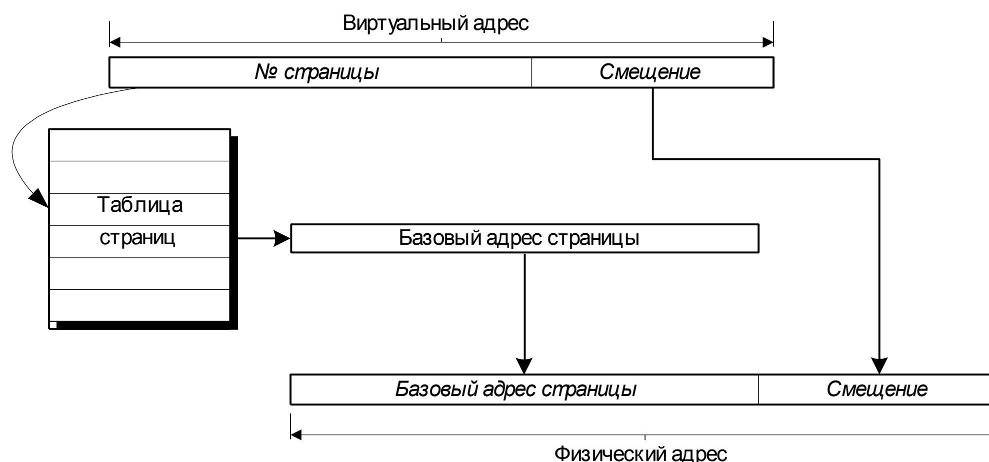

Рисунок 11 - Преобразование страничного адреса

В отличие от случая сегментной организации, вместо сложения базового адреса со смещением в данном случае можно просто собрать вместе номер физической страницы и смещение.

При переключении текущего процесса система просто изменяет адрес используемой таблицы страниц, тем самым полностью изменяя отображение виртуальных адресов на физические.

Страничная организация памяти не может привести к фрагментации, поскольку все страницы одинаковы по размеру, а потому каждая высвобожденная физическая страница может быть затем использована для любой понадобившейся виртуальной страницы.

Размер пространства виртуальных адресов каждого процесса может быть огромным, ибо он определяется только разрядностью адреса. Для 32-разрядных процессоров этот размер равен 2^{32} = 4 Гб. В настоящее время трудно представить программу, которой может всерьез понадобиться столько памяти, да и компьютер с таким объемом памяти – вещь не рядовая. На самом деле, программа обычно использует лишь небольшую часть своего адресного пространства, не более нескольких десятков или, в крайнем случае, сотен

мегабайт. Только эти используемые страницы и должны быть отображены на физическую память. Тем не менее, суммарный объем страниц, используемых всеми процессами в системе, обычно превосходит объем имеющейся физической памяти, поэтому использование страничного файла становится неизбежным.

Управление замещением страниц в физической памяти в современных PC строится по принципу *загрузки по требованию* (demand paging). Это означает следующее. Когда программа только лишь планирует использование определенной области виртуальной памяти (например, для хранения массива переменных, описанного в программе), соответствующие виртуальные страницы помечаются в таблице страниц как существующие, но находящиеся в данный момент на диске. Выделение страниц физической памяти не выполняется до тех пор, пока программа не обратится к одной из ячеек виртуальной страницы. При этом происходит аппаратное прерывание по отсутствию страницы в памяти. Это прерывание обрабатывает часть ОС, которая называется менеджером памяти. Менеджер должен выполнить следующие действия:

- найти свободную физическую страницу;
- если свободной страницы нет (а ее чаще всего нет), то по определенному алгоритму выбрать занятую страницу, которая будет вытеснена на диск;
- если выбранная страница «грязная», т.е. ее содержимое изменялось после того, как она последний раз была прочитана с диска, то «очистить» страницу, т.е. записать ее в соответствующий блок страничного файла;
- на освободившуюся физическую страницу прочитать блок страничного файла, закрепленный за запрошенной виртуальной страницей;
- откорректировать таблицу страниц, пометив вытесненную страницу как отсутствующую в физической памяти, а прочитанную – как присутствующую и при этом «чистую»;
- повторить обращение к запрошенному виртуальному адресу, теперь уже присутствующему в физической памяти.

Последующие обращения к виртуальным адресам той же страницы будут успешно выполняться, пока страница не будет, в свою очередь, вытеснена на диск.

Недостатком страничной организации является то, что при большом объеме виртуального адресного пространства сама таблица страниц должна быть очень большой. При размере страницы 4 Кб и адресном пространстве 4 Гб таблица должна содержать миллион записей!

СПИСОК ЛИТЕРТУРЫ

1. Илюшечкин В.М., Операционные системы. Учебное пособие. Изд.: «Бином», 2009 г. – 111с.
2. Дроздов С.Н.. Операционные системы: Конспект лекций. Таганрог: Изд-во ТРТУ, 2003. 136 с.
3. Кауфман В.Ш. Основы работы с Linux. Учебный курс. Изд.: ДМК-Пресс, 2110 г.
4. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. СПб.: Питер, 2001. - 544 с.
5. Операционные системы, среды и оболочки: учебное пособие / Т.Л. Партыка, И.И. Попов. – 2-е изд., испр. И доп. М.: ФОРУМ, 2009. – 528 с.