1 ВВЕДЕНИЕ

ДЕ1: ИСТОРИЯ РАЗВИТИЯ И ОБЩЕЕ ПРЕДСТАВЛЕНИЕ ОБ ОПЕРАЦИОННЫХ СИСТЕМАХ

ОБЩЕЕ ПРЕДСТАВЛЕНИЕ ОБ ОПЕРАЦИОННЫХ СИСТЕМАХ

ОС – это комплекс программ, которые обеспечивают эффективное выполнение программ, взаимодействие программ с внешними устройствами, и взаимодействие вычислительной системы с пользователем.

Операционная система в наибольшей степени определяет облик всей вычислительной системы в целом. Несмотря на это, пользователи, активно использующие вычислительную технику, зачастую испытывают затруднения при попытке дать определение операционной системе. Частично это связано с тем, что ОС выполняет две по существу мало связанные функции: обеспечение пользователю-программисту удобств посредством предоставления для него расширенной машины и повышение эффективности использования компьютера путем рационального управления его ресурсами.

На текущий момент используются только мультипрограммные (многозадачные) системы, которые могут теоретически приводить к 100% использования процессора.

Рассмотрим кратко концепцию обработки данных программным обеспечением (рис. 1.1).

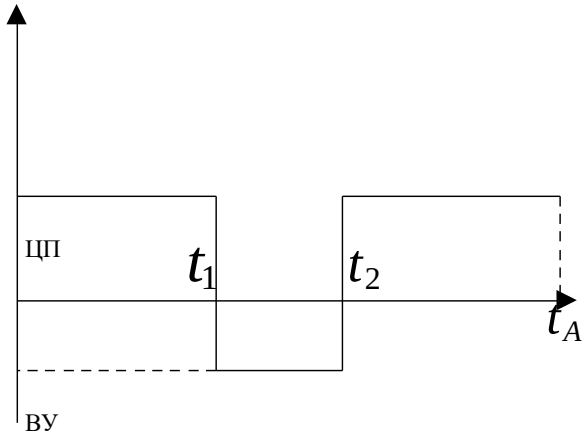


Рисунок 1.1 – Концепция обработки данных в ОС

В момент *t1* программе необходимо обратиться к внешнему устройству и в течение интервала время [*t1*, *t2*] процессор использоваться не будет, а с момента *t2* продолжается выполнение программы. Понятно, что в этом случае КПД

использования процессора далеко от 100%. Как можно повысить вычислительную нагрузку на процессор?

Одним из решений является идея мультипрограммирования (многозадачности). Идея мультипрограммирования состоит в том, что в ОЗУ помещаются несколько программ, и в тот промежуток времени, когда одна программа взаимодействует в внешним устройством другая использует ЦП (рис. 1.2).

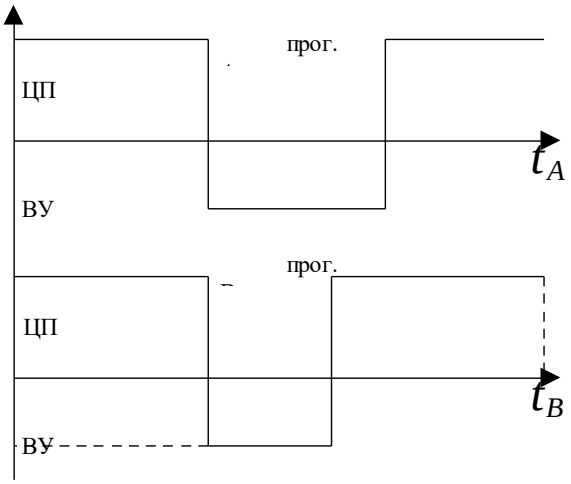


Рисунок 1.2 – Обработка данных на ЦП при многозадачности

В однозадачном режиме t = ta+tb.

В многозадачном t < ta+tb,

где t – время расчёта всего пакета.

Мультипрограммный режим является наиболее производительным, но не характерен для современных ОС. Наибольшее применение имеют ОС, которые работают по другому принципу – принцип (разделения) квантования времени.

Программы загруженные в ОЗУ начинают различается для ОС, т.к. задаются приоритеты. Каждой программе выделяется некоторый квант времени, по истечении которого задача прерывается, и управление передаётся другой задаче. Приоритеты задаёт администратор.

ОС как расширенная машина

Использование большинства компьютеров на уровне машинного языка затруднительно, особенно это касается ввода-вывода. Например, для организации чтения блока данных с гибкого диска программист может использовать 16 различных команд, каждая из которых требует 13 параметров, таких как номер блока на диске, номер сектора на дорожке и т. п. Когда выполнение операции с диском завершается, контроллер возвращает 23 значения, отражающих наличие и типы ошибок, которые, очевидно, надо анализировать. Даже если не входить в курс реальных проблем программирования ввода-вывода, ясно, что среди

программистов нашлось бы не много желающих непосредственно заниматься программированием этих операций. При работе с диском программисту-пользователю достаточно представлять его в виде некоторого набора файлов, каждый из которых имеет имя. Работа с файлом заключается в его открытии, выполнении чтения или записи, а затем в закрытии файла. Вопросы подобные таким, как следует ли при записи использовать усовершенствованную частотную модуляцию или в каком состоянии сейчас находится двигатель механизма перемещения считывающих головок, не должны волновать пользователя. Программа, которая скрывает от программиста все реалии аппаратуры и предоставляет возможность простого, удобного просмотра указанных файлов, чтения или записи - это, конечно, операционная система. Точно так же, как ОС ограждает программистов от аппаратуры дискового накопителя и предоставляет ему простой файловый интерфейс, операционная система берет на себя все малоприятные дела, связанные с обработкой прерываний, управлением таймерами и оперативной памятью, а также другие низкоуровневые проблемы. В каждом случае та абстрактная, воображаемая машина, с которой, благодаря операционной системе, теперь может иметь дело пользователь, гораздо проще и удобнее в обращении, чем реальная аппаратура, лежащая в основе этой абстрактной машины.

С этой точки зрения функцией ОС является предоставление пользователю некоторой расширенной или виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальную машину.

ОС как система управления ресурсами

Идея о том, что ОС прежде всего система, обеспечивающая удобный интерфейс пользователям, соответствует рассмотрению сверху вниз. Другой взгляд, снизу вверх, дает представление об ОС как о некотором механизме, управляющем всеми частями сложной системы. Современные вычислительные системы состоят из процессоров, памяти, таймеров, дисков, накопителей на магнитных лентах, сетевых коммуникационной аппаратуры, принтеров и других устройств. В соответствии со вторым подходом функцией ОС является распределение процессоров, памяти, устройств и данных между процессами, конкурирующими за эти ресурсы. ОС должна управлять всеми ресурсами вычислительной машины таким образом, чтобы обеспечить максимальную эффективность ее функционирования. Критерием эффективности может быть, например, пропускная способность или реактивность системы. Управление ресурсами включает решение двух общих, не зависящих от типа ресурса задач:

планирование ресурса - то есть определение, кому, когда, а для делимых ресурсов и в каком количестве, необходимо выделить данный ресурс;

отслеживание состояния ресурса - то есть поддержание оперативной информации о том, занят или не занят ресурс, а для делимых ресурсов, – какое количество ресурса уже распределено, а какое свободно.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, что, в конечном счете, и определяет их облик в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Так, например, алгоритм управления процессором в значительной степени определяет, является ли ОС системой разделения времени, системой пакетной обработки или системой реального времени.

История развития операционных систем

Рассмотрим кратко основные периоды в развитии ОС. И хотя история развития ОС по объективным причинам сильно переплетена с историей развития вычислительной техники (аппаратной части), в данном случае мы будет обращать внимание только на сами ОС.

Первый период (1945 -1955)

ОС как таковых нет вообще!

Известно, что компьютер был изобретен английским математиком Чарльзом Бэбиджем в конце восемнадцатого века. Его "аналитическая машина" так и не смогла но-настоящему заработать, потому что технологии того времени не удовлетворяли требованиям по изготовлению деталей точной механики, которые были необходимы для вычислительной техники. Известно также, что этот компьютер не имел операционной системы.

Некоторый прогресс в создании цифровых вычислительных машин произошел после второй мировой войны. В середине 40-х были созданы первые ламповые вычислительные устройства. В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. Не было никакого другого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм.

Второй период (1955 - 1965)

Появился первый транзистор, машины стали более компактными, значительно менее энергоёмкие. И достаточно надёжные, так что можно было решать задачи. Появилась необходимость загружать задачи в машины. Появились: разделение на программистов и пользователей, первые признаки систем пакетной обработки, первые алгоритмические языки, такие как Ada, Kobol, Algol

С середины 50-х годов начался новый период в развитии вычислительной техники, связанный с появлением новой технической базы - полупроводниковых элементов. Компьютеры второго поколения стали более надежными, теперь они смогли непрерывно работать настолько долго, чтобы на них можно было возложить выполнение действительно практически важных задач. Именно в этот период произошло разделение персонала на программистов и операторов, эксплуатационников и разработчиков вычислительных машин.

В эти годы появились первые алгоритмические языки, а следовательно и первые системные программы - компиляторы. Стоимость процессорного времени возросла, что потребовало уменьшения непроизводительных затрат времени между запусками программ. Появились первые системы пакетной обработки, которые просто автоматизировали запуск одной программ за другой и тем самым увеличивали коэффициент загрузки процессора. Системы пакетной обработки явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными для управления вычислительным процессом. В ходе реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Совокупность нескольких заданий получила название пакета заданий.

Третий период (1965 - 1980)

Следующий важный период развития вычислительных машин относится к 1965-1980 годам. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам, что дало гораздо большие возможности новому, третьему поколению компьютеров.

Для этого периода характерно также создание семейств программно-совместимых машин. Первым семейством программно-совместимых машин, построенных на интегральных микросхемах, явилась серия машин IBM/360. Построенное в начале 60-х годов это семейство значительно превосходило машины второго поколения по критерию цена/произ-водительность. Вскоре идея программно-совместимых машин стала общепризнанной.

Программная совместимость требовала и совместимости операционных систем. Такие операционные системы должны были бы работать и на больших, и на малых вычислительных системах, с большим и с малым количеством разнообразной периферии, в коммерческой области и в области научных исследований. Операционные системы, построенные с намерением удовлетворить всем этим противоречивым требованиям, оказались чрезвычайно сложными "монстрами". Они состояли из многих миллионов ассемблерных строк, написанных тысячами программистов, и содержали тысячи ошибок, вызывающих нескончаемый поток

исправлений. В каждой новой версии операционной системы исправлялись одни ошибки и вносились другие.

Однако, несмотря на необозримые размеры и множество проблем, OS/360 и другие ей подобные операционные системы машин третьего поколения действительно удовлетворяли большинству требований потребителей. Важнейшим достижением ОС данного поколения явилась реализация мультипрограммирования. Мультипрограммирование - это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ. Пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при последовательном выполнении программ (однопрограммный режим), а выполняет другую программу (многопрограммный режим). При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом.

Другое нововведение - спулинг (spooling). Спулинг в то время определялся как способ организации вычислительного процесса, в соответствии с которым задания считывались с перфокарт на диск в том темпе, в котором они появлялись в помещении вычислительного центра, а затем, когда очередное задание завершалось, новое задание с диска загружалось в освободившийся раздел.

Наряду с мультипрограммной реализацией систем пакетной обработки появился новый тип ОС - системы разделения времени. Вариант мультипрограммирования, применяемый в системах разделения времени, нацелен на создание для каждого отдельного пользователя иллюзии единоличного использования вычислительной машины.

Четвертый период (1980 - настоящее время)

Следующий период в эволюции операционных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и удешевление микросхем. Компьютер стал доступен отдельному человеку, и наступила эра персональных компьютеров. С точки зрения архитектуры персональные компьютеры ничем не отличались от класса миникомпьютеров типа PDP-11, но вот цена у них существенно отличалась. Если миникомпьютер дал возможность иметь собственную вычислительную машину отделу предприятия или университету, то персональный компьютер сделал это возможным для отдельного человека.

Компьютеры стали широко использоваться неспециалистами, что потребовало разработки "дружественного" программного обеспечения, это положило конец кастовости программистов.

На рынке операционных систем доминировали две системы: MS-DOS и UNIX. Однопрограммная однопользовательская ОС MS-DOS широко использовалась для компьютеров, построенных на базе микропроцессоров Intel 8088, а затем 80286, 80386 и 80486. Мультипрограммная многопользовательская ОС UNIX

доминировала в среде "не-интеловских" компьютеров, особенно построенных на базе высокопроизводительных RISC-процессоров.

В середине 80-х стали бурно развиваться сети персональных компьютеров, работающие под управлением сетевых или распределенных ОС.

В сетевых ОС пользователи должны быть осведомлены о наличии других компьютеров и должны делать логический вход в другой компьютер, чтобы воспользоваться его ресурсами, преимущественно файлами. Каждая машина в сети выполняет свою собственную локальную операционную систему, отличающуюся от ОС автономного компьютера наличием дополнительных средств, позволяющих компьютеру работать в сети. Сетевая ОС не имеет фундаментальных отличий от ОС однопроцессорного компьютера. Она обязательно содержит программную поддержку для сетевых интерфейсных устройств (драйвер сетевого адаптера), а также средства для удаленного входа в другие компьютеры сети и средства доступа к удаленным файлам, однако эти дополнения существенно не меняют структуру самой операционной системы.

КЛАССИФИКАЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

Операционные системы могут различаться особенностями реализации внутренних алгоритмов управления основными ресурсами компьютера (процессорами, памятью, устройствами), особенностями использованных методов проектирования, типами аппаратных платформ, областями использования и многими другими свойствами.

Ниже приведена классификация ОС по нескольким наиболее основным признакам.

Особенности алгоритмов управления ресурсами

От эффективности алгоритмов управления локальными ресурсами компьютера во многом зависит эффективность всей сетевой ОС в целом. Поэтому, характеризуя сетевую ОС, часто приводят важнейшие особенности реализации функций ОС по управлению процессорами, памятью, внешними устройствами автономного компьютера. В зависимости от особенностей использованного алгоритма управления процессором, операционные системы делят на многозадачные и однозадачные, многопользовательские и однопользовательские, на системы, поддерживающие многопоточную обработку и не поддерживающие ее, на многопроцессорные и однопроцессорные системы.

Различают многозадачную и мультипрограммную обработку информации. Если несколько программ загруженных в ОЗУ обмениваются информацией – осуществляют взаимодействие, то этот режим называется многозадачным. В мультипрограммном режиме программы не взаимодействуют.

Поддержка многозадачности. По числу одновременно выполняемых задач операционные системы могут быть разделены на два класса:

- однозадачные (например, MS-DOS, MSX) и
- многозадачные (ОС ЕС, OS/2, UNIX, Windows 2000+).

Однозадачные ОС в основном выполняют функцию предоставления пользователю виртуальной машины, делая более простым и удобным процесс взаимодействия пользователя с компьютером. Однозадачные ОС включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем.

Многозадачные ОС, кроме вышеперечисленных функций, управляют разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства.

Поддержка многопользовательского режима

По числу одновременно работающих пользователей ОС делятся на:

- однопользовательские (MS-DOS, Windows 3.x, ранние версии OS/2);
- многопользовательские (UNIX, Windows 2000+).

Главным отличием многопользовательских систем от однопользовательских является наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей. Следует заметить, что не всякая многозадачная система является многопользовательской, и не всякая однопользовательская ОС является однозадачной.

Вытесняющая и невытесняющая многозадачность

Важнейшим разделяемым ресурсом является процессорное время. Способ распределения процессорного времени между несколькими одновременно существующими в системе процессами (или нитями) во многом определяет специфику ОС. Среди множества существующих вариантов реализации многозадачности можно выделить две группы алгоритмов:

- невытесняющая многозадачность (NetWare, Windows 3.x, MS SQL Server);
- вытесняющая многозадачность (Windows NT, OS/2, UNIX).

Основным различием между вытесняющим и невытесняющим вариантами многозадачности является степень централизации механизма планирования процессов. В первом случае механизм планирования процессов целиком сосредоточен в операционной системе, а во втором - распределен между системой и прикладными программами. При невытесняющей многозадачности активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди

другой готовый к выполнению процесс. При вытесняющей многозадачности решение о переключении процессора с одного процесса на другой принимается операционной системой, а не самим активным процессом.

Поддержка многопоточности

Важным свойством операционных систем является возможность распараллеливания вычислений в рамках одной задачи. Многопоточная ОС разделяет процессорное время не между задачами, а между их отдельными ветвями (потоками, нитями).

Многопроцессорная обработка

Другим важным свойством ОС является отсутствие или наличие в ней средств поддержки многопроцессорной обработки - мультипроцессирование. Мультипроцессирование приводит к усложнению всех алгоритмов управления ресурсами.

В наши дни становится общепринятым введение в ОС функций поддержки многопроцессорной обработки данных. Такие функции имеются в операционных системах Solaris 2.x фирмы Sun, Open Server 3.x компании Santa Crus Operations, OS/2 фирмы IBM, Windows NT фирмы Microsoft и NetWare 4.1 фирмы Novell.

Многопроцессорные ОС могут классифицироваться по способу организации вычислительного процесса в системе с многопроцессорной архитектурой: асимметричные ОС и симметричные ОС. Асимметричная ОС целиком выполняется только на одном из процессоров системы, распределяя прикладные задачи по остальным процессорам. Симметричная ОС полностью децентрализована и использует весь пул процессоров, разделяя их между системными и прикладными задачами.

2 УПРАВЛЕНИЕ ПАМЯТЬЮ

ДЕ2: ТЕОРЕТИЧЕСКИЕ ОСНОВЫ УПРАВЛЕНИЯ ПАМЯТЬЮ

БАЗОВОЕ УПРАВЛЕНИЕ ПАМЯТЬЮ

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Распределению подлежит вся оперативная память, не занятая операционной системой. Обычно ОС располагается в самых младших адресах, однако может занимать и самые старшие адреса. Функциями ОС по управлению памятью являются: отслеживание свободной и занятой памяти, выделение памяти процессам и освобождение памяти при завершении процессов, вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место, а также настройка адресов программы на конкретную область физической памяти.

Типы адресов

Для идентификации переменных и команд используются символьные имена (метки), виртуальные адреса и физические адреса (рис. 2.1). Символьные имена присваивает пользователь при написании программы на алгоритмическом языке или ассемблере.

Виртуальные адреса вырабатывает транслятор, переводящий программу на машинный язык. Так как во время трансляции в общем случае не известно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что программа будет размещена, начиная с нулевого адреса. Совокупность виртуальных адресов процесса называется виртуальным адресным пространством. Каждый процесс имеет собственное виртуальное адресное пространство. Максимальный размер виртуального адресного пространства ограничивается разрядностью адреса, присущей данной архитектуре компьютера, и, как правило, не совпадает с объемом физической памяти, имеющимся в компьютере.

Рисунок 2.1 – Типы адресов оперативной памяти

Физические адреса соответствуют номерам ячеек оперативной памяти, где в действительности расположены или будут расположены переменные и команды. Переход от виртуальных адресов к физическим может осуществляться двумя способами. В первом случае замену виртуальных адресов на физические делает специальная системная программа - перемещающий загрузчик. Перемещающий загрузчик на основании имеющихся у него исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, и информации, предоставленной транслятором об адресно-зависимых константах программы, выполняет загрузку программы, совмещая ее с заменой виртуальных адресов физическими.

Второй способ заключается в том, что программа загружается в память в неизмененном виде в виртуальных адресах, при этом операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Второй способ является более гибким, он допускает перемещение программы во время ее выполнения, в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти. Вместе с тем использование перемещающего загрузчика уменьшает накладные расходы, так как преобразование каждого виртуального адреса происходит только один раз во время загрузки, а во втором случае - каждый раз при обращении по данному адресу.

В некоторых случаях (обычно в специализированных системах), когда заранее точно известно, в какой области оперативной памяти будет выполняться программа, транслятор выдает исполняемый код сразу в физических адресах.

Методы распределения памяти без использования дискового пространства

Все методы управления памятью могут быть разделены на два класса: методы, которые используют перемещение процессов между оперативной памятью и диском, и методы, которые не делают этого (рис. 2.2).

Рисунок 2.2 – Классификация методов распределения памяти

Начнем с последнего, более простого класса методов.

Распределение памяти фиксированными разделами

Самым простым способом управления оперативной памятью является разделение ее на несколько разделов фиксированной величины. Это может быть выполнено вручную оператором во время старта системы или во время ее генерации. Очередная задача, поступившая на выполнение, помещается либо в общую очередь (рис. 2.3,а), либо в очередь к некоторому разделу (рис. 2.3,б).

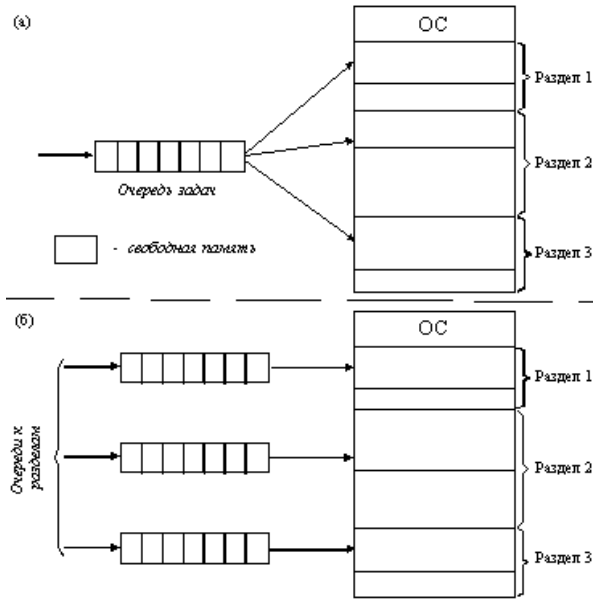


Рисунок 2.3 – Распределение памяти фиксированными разделами:
а - с общей очередью; б - с отдельными очередями

Подсистема управления памятью в этом случае выполняет следующие задачи:

- сравнивая размер программы, поступившей на выполнение, и свободных разделов, выбирает подходящий раздел,
- осуществляет загрузку программы и настройку адресов.

При очевидной простоте реализации данный метод имеет существенный недостаток – жесткость. Так как в каждом разделе может выполняться только одна программа, то уровень многозадачности заранее ограничен числом разделов, независимо от того, какой размер имеют программы. Даже если программа имеет небольшой объем, она будет занимать весь раздел, что приводит к неэффективному использованию памяти. С другой стороны, даже если объем оперативной памяти машины позволяет выполнить некоторую программу, разбиение памяти на разделы не позволяет сделать этого.

Распределение памяти разделами переменной величины

В этом случае память машины не делится заранее на разделы. Сначала вся память свободна. Каждой вновь поступающей задаче выделяется необходимая ей память. Если достаточный объем памяти отсутствует, то задача не принимается на выполнение и стоит в очереди. После завершения задачи память освобождается, и на это место может быть загружена другая задача. Таким образом, в произвольный момент времени оперативная память представляет собой случайную последовательность занятых и свободных участков (разделов) произвольного размера. На рисунке 2.4 показано состояние памяти в различные моменты времени при использовании динамического распределения. Так в момент t0 в памяти находится только ОС, а к моменту t1 память разделена между 5 задачами, причем задача П4, завершаясь, покидает память. На освободившееся после задачи П4 место загружается задача П6, поступившая в момент t3.

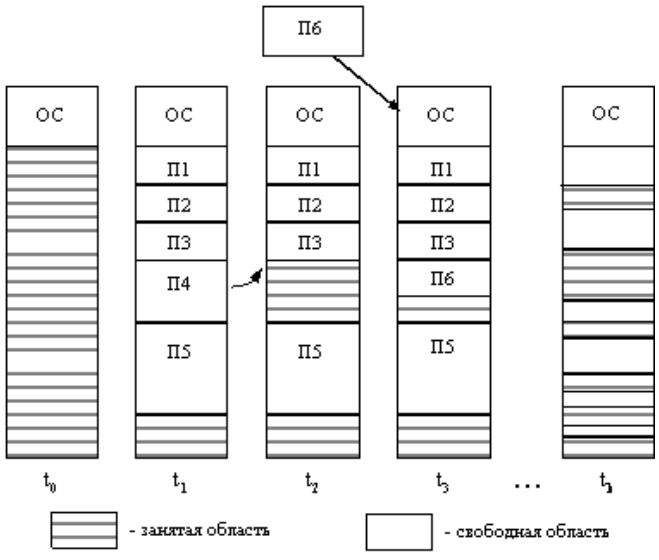


Рисунок 2.4 – Распределение памяти динамическими разделами

Задачами операционной системы при реализации данного метода управления памятью является:

- ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти,
- при поступлении новой задачи - анализ запроса, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения поступившей задачи,
- загрузка задачи в выделенный ей раздел и корректировка таблиц свободных и занятых областей,
- после завершения задачи корректировка таблиц свободных и занятых областей.

Программный код не перемещается во время выполнения, то есть может быть проведена единовременная настройка адресов посредством использования перемещающего загрузчика.

Выбор раздела для вновь поступившей задачи может осуществляться по разным правилам, таким, например, как "первый попавшийся раздел достаточного размера", или "раздел, имеющий наименьший достаточный размер", или "раздел, имеющий наибольший достаточный размер". Все эти правила имеют свои преимущества и недостатки.

По сравнению с методом распределения памяти фиксированными разделами данный метод обладает гораздо большей гибкостью, но ему присущ очень серьезный недостаток - фрагментация памяти. Фрагментация - это наличие большого числа несмежных участков свободной памяти очень маленького размера (фрагментов). Настолько маленького, что ни одна из вновь поступающих программ не может поместиться ни в одном из участков, хотя суммарный объем фрагментов может составить значительную величину, намного превышающую требуемый объем памяти.

Перемещаемые разделы

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших либо в сторону младших адресов, так, чтобы вся свободная память образовывала единую свободную область (рисунок 2.5). В дополнение к функциям, которые выполняет ОС при распределении памяти переменными разделами, в данном случае она должна еще время от времени копировать содержимое разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется "сжатием". Сжатие может выполняться либо при каждом завершении задачи, либо только тогда, когда для вновь поступившей задачи нет свободного раздела достаточного размера. В первом случае требуется меньше вычислительной работы при корректировке таблиц, а во втором - реже выполняется процедура сжатия. Так как программы перемещаются по оперативной памяти в ходе своего выполнения, то

преобразование адресов из виртуальной формы в физическую должно выполняться динамическим способом.

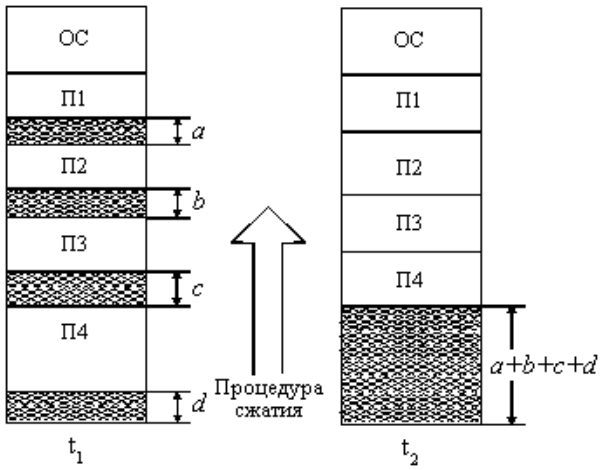


Рисунок 2.5 – Распределение памяти перемещаемыми разделами

Хотя процедура сжатия и приводит к более эффективному использованию памяти, она может потребовать значительного времени, что часто перевешивает преимущества данного метода.

УПРАВЛЕНИЕ ВИРТУАЛЬНОЙ ПАМЯТЬЮ

Уже достаточно давно пользователи столкнулись с проблемой размещения в памяти программ, размер которых превышал имеющуюся в наличии свободную память. Решением было разбиение программы на части, называемые оверлеями. 0-ой оверлей начинал выполняться первым. Когда он заканчивал свое выполнение, он вызывал другой оверлей. Все оверлеи хранились на диске и перемещались между памятью и диском средствами операционной системы. Однако разбиение программы на части и планирование их загрузки в оперативную память должен был осуществлять программист.

Развитие методов организации вычислительного процесса в этом направлении привело к появлению метода, известного под названием виртуальная память. Виртуальным называется ресурс, который пользователю или пользовательской программе представляется обладающим свойствами, которыми он в действительности не обладает. Так, например, пользователю может быть предоставлена виртуальная оперативная память, размер которой превосходит всю имеющуюся в системе реальную оперативную память. Пользователь пишет программы так, как будто в его распоряжении имеется однородная оперативная память большого объема, но в действительности все данные, используемые программой, хранятся на одном или нескольких разнородных запоминающих устройствах, обычно на дисках, и при необходимости частями отображаются в реальную память.

Таким образом, виртуальная память - это совокупность программно-аппаратных средств, позволяющих пользователям писать программы, размер которых превосходит имеющуюся оперативную память; для этого виртуальная память решает следующие задачи:

- размещает данные в запоминающих устройствах разного типа, например, часть программы в оперативной памяти, а часть на диске;
- перемещает по мере необходимости данные между запоминающими устройствами разного типа, например, подгружает нужную часть программы с диска в оперативную память;
- преобразует виртуальные адреса в физические.

Все эти действия выполняются автоматически, без участия программиста, то есть механизм виртуальной памяти является прозрачным по отношению к пользователю.

Наиболее распространенными реализациями виртуальной памяти является страничное, сегментное и странично-сегментное распределение памяти, а также свопинг.

Сегментная организация виртуальной памяти

При страничной организации виртуальное адресное пространство процесса делится механически на равные части. Это не позволяет дифференцировать способы доступа к разным частям программы (сегментам), а это свойство часто бывает очень полезным. Например, можно запретить обращаться с операциями записи и чтения в кодовый сегмент программы, а для сегмента данных разрешить только чтение. Кроме того, разбиение программы на "осмысленные" части делает принципиально возможным разделение одного сегмента несколькими процессами. Например, если два процесса используют одну и ту же математическую подпрограмму, то в оперативную память может быть загружена только одна копия этой подпрограммы.

Рассмотрим, каким образом сегментное распределение памяти реализует эти возможности (рисунок 2.6). Виртуальное адресное пространство процесса делится на сегменты, размер которых определяется программистом с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т.п. Иногда сегментация программы выполняется по умолчанию компилятором.

При загрузке процесса часть сегментов помещается в оперативную память (при этом для каждого из этих сегментов операционная система подыскивает подходящий участок свободной памяти), а часть сегментов размещается в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки. Во время загрузки система создает таблицу сегментов процесса (аналогичную таблице страниц), в которой для каждого сегмента

указывается начальный физический адрес сегмента в оперативной памяти, размер сегмента, правила доступа, признак модификации, признак обращения к данному сегменту за последний интервал времени и некоторая другая информация. Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.

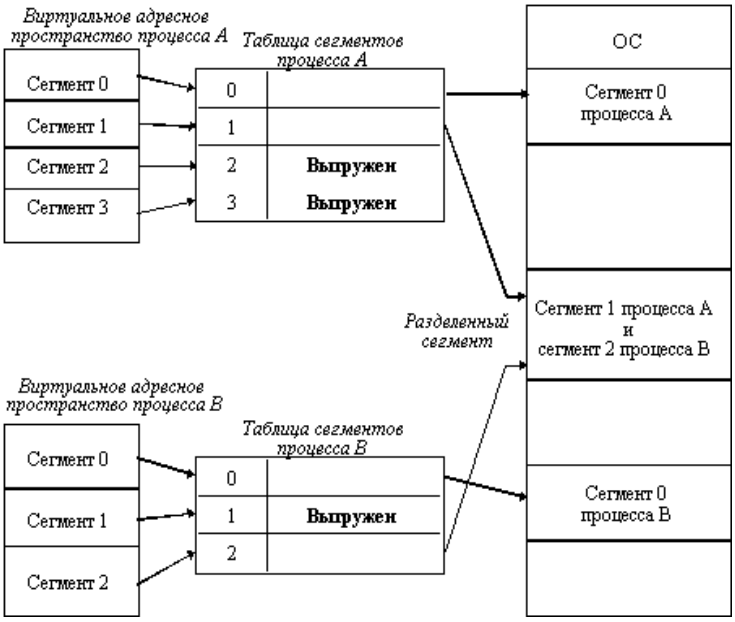


Рисунок 2.6 – Распределение памяти сегментами

Система с сегментной организацией функционирует аналогично системе со страничной организацией: время от времени происходят прерывания, связанные с отсутствием нужных сегментов в памяти, при необходимости освобождения памяти некоторые сегменты выгружаются, при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Кроме того, при обращении к памяти проверяется, разрешен ли доступ требуемого типа к данному сегменту.

Виртуальный адрес при сегментной организации памяти может быть представлен парой (g, s), где g - номер сегмента, а s - смещение в сегменте. Физический адрес получается путем сложения начального физического адреса сегмента, найденного в таблице сегментов по номеру g, и смещения s.

Недостатком данного метода распределения памяти является фрагментация на уровне сегментов и более медленное по сравнению со страничной организацией преобразование адреса.

Страничная организация виртуальной памяти

На рисунке 2.7 показана схема страничного распределения памяти. Виртуальное адресное пространство каждого процесса делится на части одинакового,

фиксированного для данной системы размера, называемые виртуальными страницами. В общем случае размер виртуального адресного пространства не является кратным размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

Вся оперативная память машины также делится на части такого же размера, называемые физическими страницами (или блоками).

Размер страницы обычно выбирается равным степени двойки: 512, 1024 и т.д., это позволяет упростить механизм преобразования адресов.

При загрузке процесса часть его виртуальных страниц помещается в оперативную память, а остальные - на диск. Смежные виртуальные страницы не обязательно располагаются в смежных физических страницах. При загрузке операционная система создает для каждого процесса информационную структуру - таблицу страниц, в которой устанавливается соответствие между номерами виртуальных и физических страниц для страниц, загруженных в оперативную память, или делается отметка о том, что виртуальная страница выгружена на диск. Кроме того, в таблице страниц содержится управляющая информация, такая как признак модификации страницы, признак невыгружаемости (выгрузка некоторых страниц может быть запрещена), признак обращения к странице (используется для подсчета числа обращений за определенный период времени) и другие данные, формируемые и используемые механизмом виртуальной памяти.

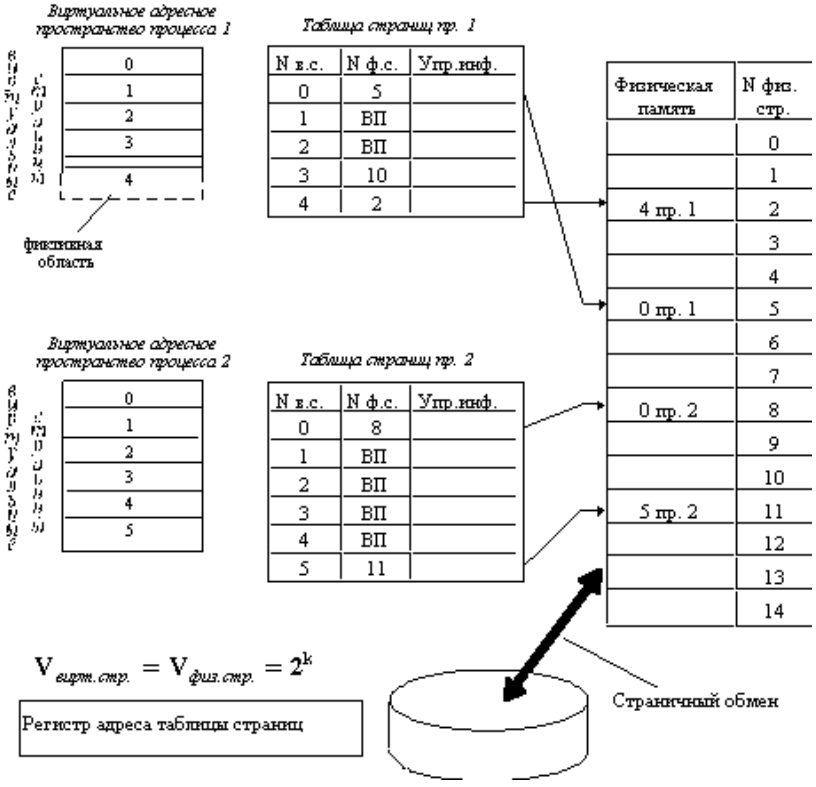


Рисунок 2.7 – Страничное распределение памяти

При активизации очередного процесса в специальный регистр процессора загружается адрес таблицы страниц данного процесса.

При каждом обращении к памяти происходит чтение из таблицы страниц информации о виртуальной странице, к которой произошло обращение. Если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое страничное прерывание. Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди готовых. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то решается вопрос, какую страницу следует выгрузить из оперативной памяти.

В данной ситуации может быть использовано много разных критериев выбора, наиболее популярные из них следующие:

- дольше всего не использовавшаяся страница,
- первая попавшаяся страница,
- страница, к которой в последнее время было меньше всего обращений.

В некоторых системах используется понятие рабочего множества страниц. Рабочее множество определяется для каждого процесса и представляет собой перечень наиболее часто используемых страниц, которые должны постоянно находиться в оперативной памяти и поэтому не подлежат выгрузке.

После того, как выбрана страница, которая должна покинуть оперативную память, анализируется ее признак модификации (из таблицы страниц). Если выталкиваемая страница с момента загрузки была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то она может быть просто уничтожена, то есть соответствующая физическая страница объявляется свободной.

Рассмотрим механизм преобразования виртуального адреса в физический при страничной организации памяти (рисунок 2.8). Виртуальный адрес при страничном распределении может быть представлен в виде пары (p, s), где p - номер виртуальной страницы процесса (нумерация страниц начинается с 0), а s - смещение в пределах виртуальной страницы. Учитывая, что размер страницы равен 2 в степени к, смещение s может быть получено простым отделением k младших разрядов в двоичной записи виртуального адреса. Оставшиеся старшие разряды представляют собой двоичную запись номера страницы p.

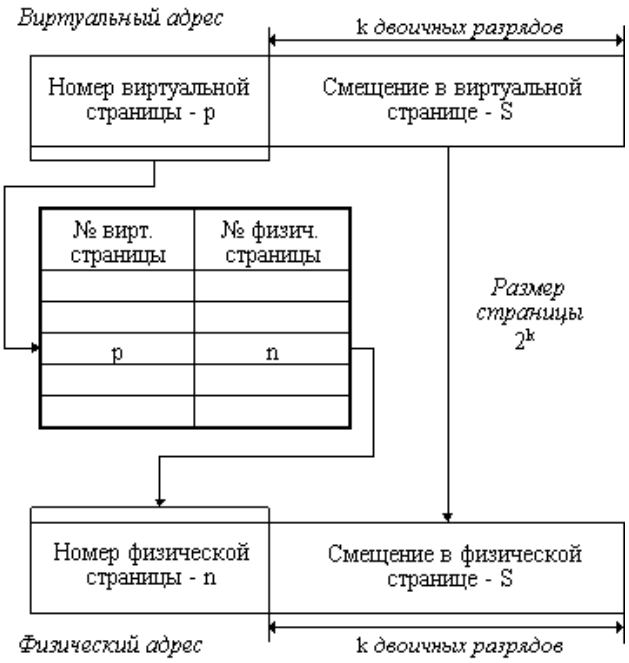


Рисунок 2.8 – Механизм преобразования виртуального адреса в физический при страничной организации памяти

При каждом обращении к оперативной памяти аппаратными средствами выполняются следующие действия:

- на основании начального адреса таблицы страниц (содержимое регистра адреса таблицы страниц), номера виртуальной страницы (старшие разряды виртуального адреса) и длины записи в таблице страниц (системная константа) определяется адрес нужной записи в таблице,
- из этой записи извлекается номер физической страницы,
- к номеру физической страницы присоединяется смещение (младшие разряды виртуального адреса).

Использование того факта, что размер страницы равен степени 2, позволяет применить операцию конкатенации (присоединения) вместо более длительной операции сложения, что уменьшает время получения физического адреса, а значит повышает производительность компьютера.

На производительность системы со страничной организацией памяти влияют временные затраты, связанные с обработкой страничных прерываний и преобразованием виртуального адреса в физический. При часто возникающих страничных прерываниях система может тратить большую часть времени впустую, на свопинг страниц. Чтобы уменьшить частоту страничных прерываний, следовало бы увеличивать размер страницы. Кроме того, увеличение размера страницы уменьшает размер таблицы страниц, а значит уменьшает затраты памяти. С другой стороны, если страница велика, значит велика и фиктивная область в последней виртуальной странице каждой программы. В среднем на каждой программе теряется половина объема страницы, что в сумме при большой странице может составить существенную величину. Время преобразования виртуального адреса в

физический в значительной степени определяется временем доступа к таблице страниц. В связи с этим таблицу страниц стремятся размещать в "быстрых" запоминающих устройствах. Это может быть, например, набор специальных регистров или память, использующая для уменьшения времени доступа ассоциативный поиск и кэширование данных.

Страничное распределение памяти может быть реализовано в упрощенном варианте, без выгрузки страниц на диск. В этом случае все виртуальные страницы всех процессов постоянно находятся в оперативной памяти. Такой вариант страничной организации хотя и не предоставляет пользователю виртуальной памяти, но почти исключает фрагментацию за счет того, что программа может загружаться в несмежные области, а также того, что при загрузке виртуальных страниц никогда не образуется остатков.

Странично-сегментная организация виртуальной памяти

Как видно из названия, данный метод представляет собой комбинацию страничного и сегментного распределения памяти и, вследствие этого, сочетает в себе достоинства обоих подходов. Виртуальное пространство процесса делится на сегменты, а каждый сегмент в свою очередь делится на виртуальные страницы, которые нумеруются в пределах сегмента. Оперативная память делится на физические страницы. Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть на диске. Для каждого сегмента создается своя таблица страниц, структура которой полностью совпадает со структурой таблицы страниц, используемой при страничном распределении. Для каждого процесса создается таблица сегментов, в которой указываются адреса таблиц страниц для всех сегментов данного процесса. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс. На рисунке 2.9 показана схема преобразования виртуального адреса в физический для данного метода.

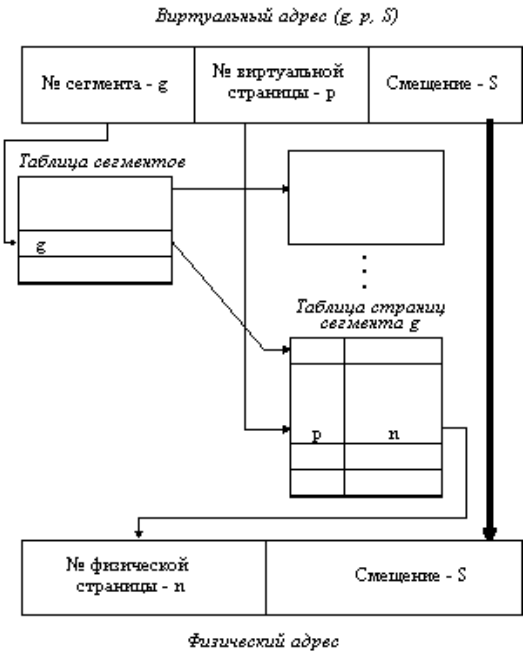


Рисунок 2.9 – Схема преобразования виртуального адреса в физический для сегментно-страничной организации памяти

ДЕЗ: УПРАВЛЕНИЕ ПАМЯТЬЮ В СИСТЕМАХ UNIX

РЕАЛИЗАЦИЯ УПРАВЛЕНИЯ ВИРТУАЛЬНОЙ ПАМЯТЬЮ В СИСТЕМАХ UNIX

Модель памяти, используемая в системе UNIX, довольно проста, что должно обеспечить переносимость программ, а также реализацию операционной системы UNIX на машинах с сильно отличающимися модулями памяти, варьирующимися от элементарных (например, оригинальная IBM PC) до сложного оборудования со страничной организацией. Эта область практически не изменилась за последние несколько десятков лет. Разработанные уже давно архитектурные решения хорошо себя зарекомендовали и не требуют серьезной переработки. Мы рассмотрим модель управления памятью в системе UNIX и методы ее реализации.

Основные понятия

У каждого процесса в системе UNIX есть адресное пространство, состоящее из трех сегментов: текста (программы), данных и стека. Пример адресного пространства процесса изображен на рисунке 2.10, а. Текстовый (программный) сегмент содержит машинные команды, образующие исполняемый код программы. Он создается компилятором и ассемблером при трансляции программы, написанной на языке высокого уровня (например, C или C++) в машинный код. Как правило, текстовый сегмент разрешен только для чтения. Самомодифицирующиеся программы вышли из моды примерно в 1950 году, так как их было слишком сложно

понимать и отлаживать. Таким образом, текстовый сегмент не изменяется ни в размерах, ни по своему содержанию.

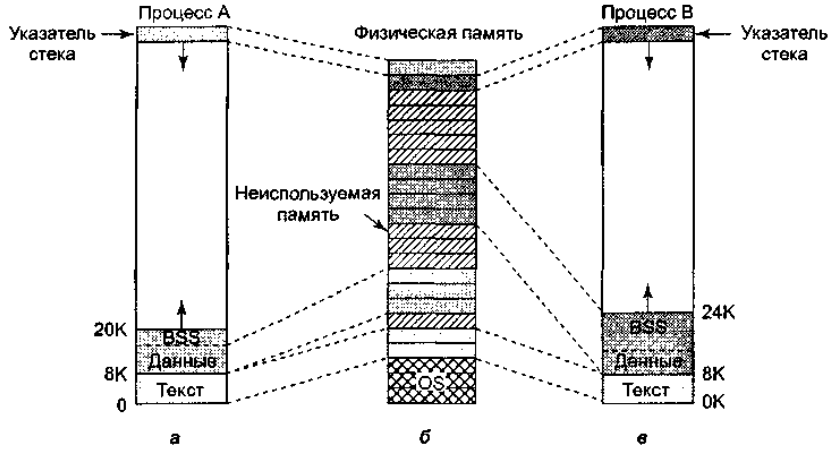


Рисунок 2.10 – Виртуальное адресное пространство процесса A (а); физическая память (б); виртуальное адресное пространство процесса B (в)

Сегмент данных содержит переменные, строки, массивы и другие данные программы. Он состоит из двух частей: инициализированных данных и неинициализированных данных. По историческим причинам вторая часть называется BSS (Bulk Storage System — запоминающее устройство большой емкости, массовое ЗУ). Инициализированная часть сегмента данных содержит переменные и константы компилятора, значения которых должны быть заданы при запуске программы.

Например, на языке C можно объявить символьную строку и в то же время задать ее значение, то есть проинициализировать ее. Когда программа запускается, она предполагает, что в этой строке уже содержится некий осмысленный текст. Чтобы реализовать это, компилятор назначает строке определенное место в адресном пространстве и гарантирует, что в момент запуска программы по этому адресу будет располагаться соответствующая строка. С точки зрения операционной системы, инициализированные данные не отличаются от текста программы — и тот и другой сегменты содержат сформированные компилятором последовательности битов, загружаемые в память при запуске программы.

Неинициализированные данные необходимы лишь с точки зрения оптимизации. Когда начальное значение глобальной переменной явно не указано, то, согласно семантике языка C, ее значение устанавливается равным 0. На практике большинство глобальных переменных не инициализируются, и, таким образом, их начальное значение равно 0. Это можно реализовать следующим образом: создать целый сегмент исполняемого двоичного файла, точно равного по размеру числу байтов данных, и проинициализировать весь этот сегмент нулями. Однако из экономии места на диске этого не делается. Файл содержит только те переменные, начальные значения которых явно заданы. Вместо неинициализированных переменных компилятор помещает в исполняемый файл просто одно слово,

содержащее размер области неинициализированных данных в байтах. При запуске программы операционная система считывает это слово, выделяет нужное число байтов и обнуляет их.

Рассмотрим это еще раз на нашем примере (см. рис. 2.10, а). Здесь текст программы занимает 8 Кбайт, и инициализированные данные также занимают 8 Кбайт. Размер сегмента неинициализированных данных (BSS) равен 4 Кбайт. Исполняемый файл содержит только 16 Кбайт (текст + инициализированные данные), плюс короткий заголовок, в котором операционной системе указывается выделить программе дополнительно 4 Кбайт после неинициализированных данных и обнулить их перед выполнением программы. Этот трюк позволяет сэкономить 4 Кбайт нулей на диске в исполняемом файле.

В отличие от текстового сегмента, который не может изменяться, сегмент данных может модифицироваться. Программы изменяют свои переменные постоянно. Более того, многим программам требуется выделение дополнительной памяти динамически, во время выполнения. Чтобы реализовать это, операционная система UNIX разрешает сегменту данных расти при динамическом выделении памяти программам и уменьшаться при освобождении памяти программами. Программа может установить размер своего сегмента данных с помощью системного вызова brk. Таким образом, чтобы получить больше памяти, программа может увеличить размер своего сегмента данных. Этим системным вызовом пользуется библиотечная процедура malloc, используемая для выделения памяти.

Третий сегмент — это сегмент стека. На большинстве компьютеров он начинается около старших адресов виртуального адресного пространства и растет вниз к 0. Если указатель стека оказывается ниже нижней границы сегмента стека, как правило, происходит аппаратное прерывание, при котором операционная система понижает границу сегмента стека на одну страницу памяти. Программы не управляют явно размером сегмента стека.

Когда программа запускается, ее стек не пуст. Напротив, он содержит все переменные окружения (оболочки), а также командную строку, введенную в оболочке при вызове этой программы. Таким образом, программа может узнать параметры, с которыми она была запущена. Например, когда вводится команда

cp src dest

запускается программа cp со строкой «cp src dest» в стеке, что позволяет ей определить имена файлов, с которыми ей предстоит работать. Строка представляется в виде массива указателей на отдельные аргументы командной строки, что облегчает ее обработку.

Когда два пользователя запускают одну и ту же программу, например текстовый редактор, в памяти можно хранить две копии программы редактора. Однако такой подход является неэффективным. Вместо этого большинством систем UNIX поддерживаются текстовые сегменты совместного использования. На рис. 2.10, б и

в мы видим два процесса, A и B, совместно использующие общин текстовый сегмент. Отображение выполняется аппаратным обеспечением виртуальной памяти.

Сегменты данных и стека никогда не бывают общими, кроме как после выполнения системного вызова fork, и то только те страницы, которые не модифицируются любым из процессов. Если размер любого из сегментов должен быть увеличен, то отсутствие свободного места в соседних страницах памяти не является проблемой, так как соседние виртуальные страницы памяти не обязаны отображаться на соседние физические страницы.

На некоторых компьютерах аппаратное обеспечение поддерживает раздельные адресные пространства для команд и для данных. Если такая возможность есть, система UNIX может ею воспользоваться. Например, на компьютере с 32-разрядными адресами при возможности использования раздельных адресных пространств можно получить 232 бит1 адресного пространства для команд и еще 232 бит адресного пространства для данных. Передача управления по адресу 0 будет восприниматься как передача управления по адресу 0 в текстовом пространстве, тогда как при обращении к данным по адресу 0 будет использоваться адрес 0 в пространстве данных. Таким образом, это свойство удваивает доступное адресное пространство.

Многими версиями UNIX поддерживается отображение файлов на адресное пространство памяти. Это свойство позволяет отображать файл на часть адресного пространства процесса, так чтобы можно было читать из файла и писать в файл, как если бы это был массив, хранящийся в памяти. Отображение файла на адресное пространство памяти делает произвольный доступ к нему существенно более легким, нежели при использовании системных вызовов, таких как read и write. Совместный доступ к библиотекам предоставляется именно при помощи этого механизма. На рис. 2.11 показан файл, одновременно отображенный на адресные пространства двух процессов по различным виртуальным адресам.

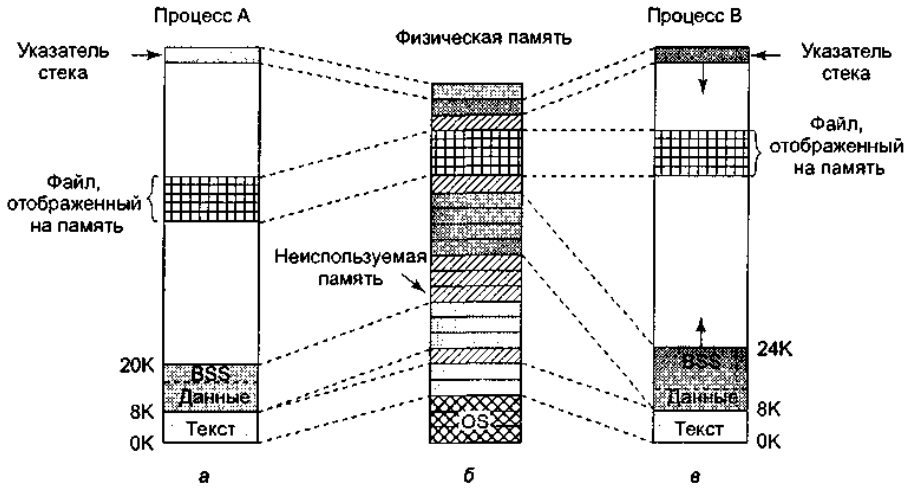


Рисунок 2.11 – Два процесса совместно используют один отображенный на память файл

Дополнительное преимущество отображения файла на память заключается в том, что два или более процессов могут одновременно отобразить на свое адресное пространство один и тот же файл. Запись в этот файл одним из процессов мгновенно становится видимой всем остальным. Таким образом, отображение на адресное пространство памяти временного файла (который будет удален после завершения работы процессов) представляет собой механизм реализации общей памяти для нескольких процессов, причем у такого механизма будет высокая пропускная способность. В предельном случае два или более процессов могут отобразить на память файл, покрывающий все адресное пространство, получая, таким образом, форму совместного использования памяти, что-то среднее между процессами и потоками. В этом случае, как и у потоков, все адресное пространство используется совместно, но каждый процесс может управлять собственными файлами и сигналами, что отличает этот вариант от потоков. Однако на практике такое никогда не применяется.

Реализация управления памятью в UNIX

До версии 3BSD большинство систем UNIX основывались на свопинге (подкачке), работавшем следующим образом. Когда загружалось больше процессов, чем могло поместиться в памяти, некоторые из них выгружались на диск. Выгружаемый процесс всегда выгружался на диск целиком (исключение представляли только совместно используемые текстовые сегменты). Таким образом, процесс мог быть либо в памяти, либо на диске.

Свопинг

Перемещением данных между памятью и диском управлял верхний уровень двухуровневого планировщика, называвшийся свопером (swapper). Выгрузка данных из памяти на диск инициировалась, когда у ядра кончалась свободная память из-за одного из следующих событий:

1. Системному вызову fork требовалась память для дочернего процесса.

2. Системный вызов brk собирался расширить сегмент данных.

3. Разросшемуся стеку требовалась дополнительная память.

Кроме того, когда наступало время запустить процесс, уже достаточно долго находящийся на диске, часто бывало необходимо удалить из памяти другой процесс, чтобы освободить место для запускаемого процесса.

Выбирая жертву, свопер сначала рассматривал блокированные (например, ожиданием ввода с терминала) процессы. Лучше удалить из памяти процесс, который не может работать, чем работоспособный процесс. Если такие процессы находились, из них выбирался процесс с наивысшим значением суммы приоритета и времени пребывания в памяти. Таким образом, хорошими кандидатами на выгрузку были процессы, потребившие большое количество процессорного

времени, либо находящиеся в памяти уже достаточно долгое время, даже если большую его часть они занимались вводом-выводом. Если блокированных процессов не было, тогда на основе тех же критериев выбирался готовый процесс.

Каждые несколько секунд свопер исследовал список выгруженных процессов, проверяя, не готов ли какой-либо из этих процессов к работе. Если процессы в состоянии готовности обнаруживались, из них выбирался процесс, дольше всех находящийся на диске. Затем свопер проверял, будет ли это легкий свопинг или тяжелый. Легким свопингом считался тот, для которого не требовалось дополнительное высвобождение памяти. При этом нужно было всего лишь загрузить выгруженный на диск процесс. Тяжелым свопингом назывался свопинг, при котором для загрузки в память выгруженного на диск процесса из нее требовалось удалить один или несколько других процессов.

Затем весь этот алгоритм повторялся до тех пор, пока не выполнялось одно из следующих двух условий: (1) на диске не оставалось процессов, готовых к работе, или (2) в памяти не оставалось места для новых процессов. Чтобы не терять большую часть производительности системы на свопинг, ни один процесс не выгружался на диск, если он пробыл в памяти менее 2 с.

Свободное место в памяти и на устройстве перекачки учитывалось при помощи связного списка свободных пространств. Когда требовалось свободное пространство в памяти или на диске, из списка выбиралось первое подходящее свободное пространство. После этого в список возвращался остаток от свободного пространства.

Постраничная подкачка в системе UNIX

Все версии операционной системы UNIX для компьютеров PDP-11 и Interdata, а также начальная версия для машины VAX были основаны на свопинге, о котором только что рассказывалось. Однако, начиная с версии 3BSD, университет в Беркли добавил к системе страничную подкачку, чтобы предоставить возможность работать с программами самых больших размеров. Практически во всех версиях системы UNIX теперь есть страничная подкачка по требованию, появившаяся впервые в версии 3BSD. Ниже мы опишем строение версии 4BSD, но System V основана на 4BSD и во многом схожа с нею.

Идея, лежащая в основе страничной подкачки в системе 4BSD, проста: чтобы работать, процессу не нужно целиком находиться в памяти. Все, что в действительности требуется, — это структура пользователя и таблицы страниц. Если они загружены, то процесс считается находящимся в памяти и может быть запущен планировщиком. Страницы с сегментами текста, данных и стека загружаются в память динамически, по мере обращения к ним. Если пользовательской структуры и таблицы страниц нет в памяти, то процесс не может быть запущен, пока свопер не загрузит их.

В системе Berkeley UNIX не используется модель рабочего набора или любая другая форма опережающей подкачки страниц, так как для этого требуется знать, какие страницы используются в данный момент, а какие нет. Поскольку в машине VAX не было битов обращений к памяти, эту информацию было получить непросто (хотя это можно было поддержать программно за счет значительных дополнительных накладных расходов).

Страничная подкачка реализуется частично ядром и частично новым процессом, называемым страничным демоном. Страничный демон — это процесс 2. Как и все демоны, страничный демон периодически запускается и смотрит, есть ли для него работа. Если он обнаруживает, что количество страниц в списке свободных страниц слишком мало, страничный демон инициирует действия по освобождению дополнительных страниц.

Организация памяти в 4BSD показана на рис. 2.12. Память делится на три части. Первые две части, ядро операционной системы и карта памяти, фиксированы в физической памяти (то есть никогда не выгружаются). Остальная память компьютера делится на страничные блоки, каждый из которых может содержать страницу текста, данных или стека или находиться в списке свободных страниц.

Карта памяти содержит информацию о содержимом страничных блоков. Для каждого страничного блока в карте памяти есть запись фиксированной длины. При килобайтных страничных блоках и 16-байтовых записях карты памяти на карту памяти расходуется менее 2 % от общего объема памяти. Первые два поля записи карты памяти используются только тогда, когда соответствующий страничный блок находится в списке свободных страниц. В этом случае они сшивают свободные страницы в двусвязный список. Следующие три записи используются, когда страничный блок содержит информацию. У каждой страницы в памяти есть фиксированное место хранения на диске, в которое она помещается, когда выгружается из памяти. Еще три поля содержат ссылку на запись в таблице процессов, тип хранящегося в странице сегмента и смещение в сегменте процесса. Последнее поле содержит некоторые флаги, нужные для алгоритма страничной подкачки.

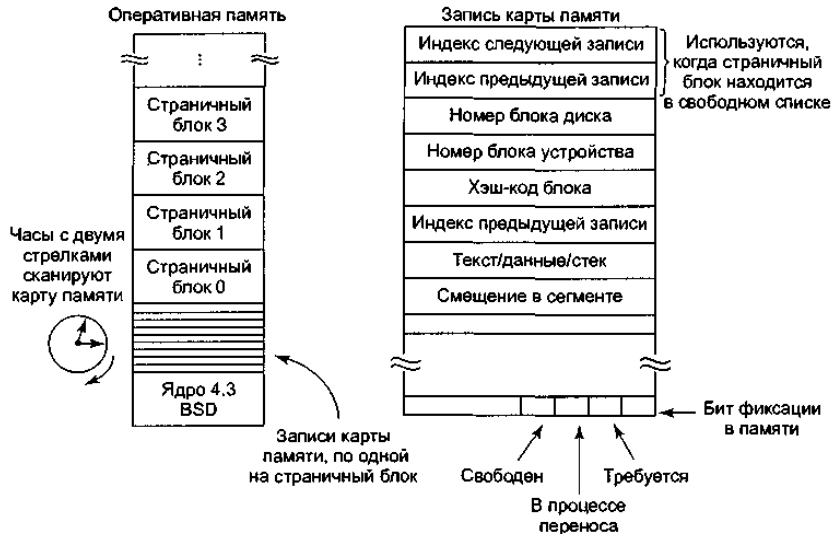


Рисунок 2.12 – Карта памяти в 4BSD

При запуске процесс может вызвать страничное прерывание, если одной или нескольких его страниц не окажется в памяти. При страничном прерывании операционная система берет первый страничный блок из списка свободных страниц, удаляет его из списка и считывает в него требуемую страницу. Если список свободных страниц пуст, выполнение процесса приостанавливается до тех пор, пока страничный демон не освободит страничный блок.

СИСТЕМНЫЕ ФУНКЦИИ УПРАВЛЕНИЯ ПАМЯТЬЮ В СИСТЕМАХ UNIX

Функция «malloc»

```
char *malloc (unsigned size);
int mallopt (int cmd, int value);
```

Функции malloc и free предоставляют простой универсальный механизм выделения и освобождения памяти. Функция malloc возвращает указатель на блок размером не менее size байт, который можно использовать в произвольных целях.

С помощью функции mallopt можно управлять алгоритмами выделения и освобождения памяти. Допускаются следующие значения аргумента cmd:

- M_MXFAST – Установить величину maxfast равной аргументу value. Алгоритм отводит место сразу для большой группы блоков, размер которых не превосходит maxfast, а затем по запросам очень быстро выдает их. Подразумеваемое значение maxfast равно 24.
- M_NLBLKS – Установить величину numblks равной аргументу value. Каждая из вышеупомянутых "больших групп" содержит numblks блоков. Число блоков должно быть больше 0; подразумеваемое значение numblks равно 100.
- M_GRAIN – Установить величину grain равной аргументу value. Размеры всех блоков, не превосходящие maxfast, округляются вверх до

ближайшего кратного grain. Значение grain должно быть больше 0; подразумеваемое значение таково, чтобы обеспечить правильное выравнивание по границе данных любого типа. При изменении grain аргумент value округляется вверх до ближайшего кратного значения по умолчанию.

– M_KEEP – Сохранить данные в освобождаемом блоке до следующих вызовов функций malloc, realloc или calloc. Эта опция предоставлена только для совместимости со старой версией malloc; пользоваться ей не рекомендуется.

Функция «free»

```
void free(char *ptr);
```

Аргументом функции free является указатель на блок, предварительно выделенный с помощью malloc; после выполнения free блок может быть выделен вновь, а хранящаяся в нем информация теряется (см. однако описание функции mallopt, команда M_KEEP).

Функция «brk»

```
int brk(char *addr);
char *sbrk(int incr);
```

Sbrk и brk используются для динамического изменения размера сегмента данных текущего процесса. Изменение размера производится путем установки новой границы для сегмента данных, т.е. адреса первой ячейки памяти, находящейся за концом сегмента данных процесса.

Sbrk добавляет incr байт к значению границы сегмента и соответствующим образом изменяет выделенное сегменту место. Значение параметра incr может быть отрицательным, что уменьшает размер сегмента данных.

В случае успешного завершения, функция sbrk возвращает указатель на начало выделенного места, а brk возвращает ноль.

ДЕ4: УПРАВЛЕНИЕ ПАМЯТЬЮ В СИСТЕМАХ MICROSOFT WINDOWS

2.6 РЕАЛИЗАЦИИ УПРАВЛЕНИЯ ВИРТУАЛЬНОЙ ПАМЯТЬЮ В MICROSOFT WINDOWS

В современных операционных системах Windows крайне сложная система виртуальной памяти. В Windows существует множество функций Win32 для использования виртуальной памяти и часть исполняющей системы плюс шесть выделенных потоков ядра для управления ею. В следующих разделах мы

рассмотрим основные понятия, вызовы Win32 и, наконец, реализацию управления памятью.

Основные понятия

В операционной системе Windows у каждого пользовательского процесса есть собственное виртуальное адресное пространство. Виртуальные адреса 32-разрядные, поэтому у каждого процесса 4 Гбайт виртуального адресного пространства. Нижние 2 Гбайт за вычетом около 256 Мбайт доступны для программы и данных процесса; верхние 2 Гбайт защищенным образом отображаются на память ядра. Страницы виртуального адресного пространства имеют фиксированный размер (4 Кбайт на компьютере с процессором Pentium) и подгружаются по требованию. Конфигурация виртуального адресного пространства для трех пользовательских процессов в слегка упрощенном виде показана на рис. 2.13. Белым цветом на рисунке изображена область приватных данных процесса. Затененные области представляют собой память, совместно используемую всеми процессами. Нижние и верхние 64 Кбайт каждого виртуального адресного пространства в обычном состоянии не отображаются на физическую память. Это делается преднамеренно, чтобы облегчить перехват программных ошибок. Недействительные указатели часто имеют значение 0 или -1, и попытки их использования в системе Windows 2000 вызовут немедленное прерывание вместо чтения или, что еще хуже, записи слова по неверному адресу. Однако когда запускаются старые программы MS-DOS в режиме эмуляции, нижние 64 Кбайт могут отображаться на физическую память.

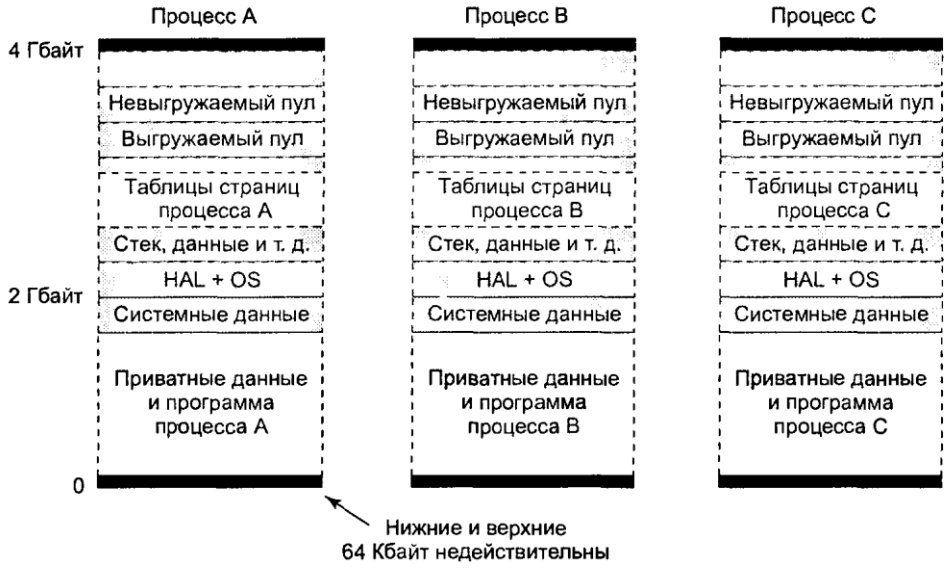


Рисунок 2.13 – Конфигурация виртуального адресного пространства для трех пользовательских процессов

Начиная с адреса 64 К, могут располагаться приватные данные и программа пользователя. Они могут занимать почти 2 Гбайт. Последний фрагмент этих 2 Гбайт памяти содержит некоторые системные указатели и таймеры, используемые совместно всеми пользователями в режиме доступа «только чтение». Отображение

этих данных в эту область памяти позволяет всем процессам получать к ним доступ без лишних системных вызовов.

Верхние 2 Гбайт виртуального адресного пространства содержат операционную систему, включая код, данные и выгружаемый и невыгружаемый пулы (используемые для объектов и т. д.). Верхние 2 Гбайт используются совместно всеми процессами, кроме таблиц страниц, которые являются индивидуальными для каждого процесса. Верхние 2 Гбайт процессам в режиме пользователя запрещены для записи, а по большей части также запрещены и для чтения. Причина, по которой они размещаются здесь, заключается в том, что когда поток обращается к системному вызову, он переключается в режим ядра, но остается все тем же потоком. Если сделать всю операционную систему и все ее структуры данных (как и весь пользовательский процесс) видимыми в адресном пространстве потока, когда он переключается в режим ядра, то отпадает необходимость в изменении карты памяти или выгрузке кэша при входе в ядро. Все, что нужно сделать, — это переключиться на стек режима ядра. Платой за более быстрые системные вызовы при данном подходе является уменьшение приватного адресного пространства для каждого процесса. Большим базам данных уже сейчас становится тесно в таких рамках, вот почему в версиях Windows 2000 Advanced server и Datacenter Server есть возможность использования 3 Гбайт для адресного пространства пользовательских процессов.

Каждая виртуальная страница может находиться в одном из трех состояний: свободном, зарезервированном и фиксированном. Свободная страница не используется в настоящий момент, и ссылка на нее вызывает страничное прерывание. Когда процесс запускается, все его страницы находятся в свободном состоянии, пока программа и исходные данные не будут отображены на их адресное пространство. Как только данные или программа отображаются на страницу, страница называется фиксированной. Обращение к фиксированной странице преобразуется при помощи аппаратного обеспечения виртуальной памяти и завершается успехом, если эта страница находится в оперативной памяти. В противном случае происходит страничное прерывание, операционная система находит требуемую страницу на диске и считывает ее в оперативную память.

Виртуальная страница может также находиться в зарезервированном состоянии, в таком случае эта страница не может отображаться, пока резервирование не будет явно удалено. Например, когда создается новый поток, в виртуальном адресном пространстве резервируется 1 Мбайт пространства для стека, но фиксируется только одна страница. Такая техника означает, что стек может вырасти до 1 Мбайт без опасения, что какой-либо другой поток захватит часть необходимого непрерывного виртуального адресного пространства. Помимо состояния (свободная, зарезервированная или фиксированная), у страниц есть также и другие атрибуты, например страница может быть доступной для чтения, записи или исполнения.

При выделении фиксированным страницам места резервного хранения используется интересный компромисс. Простая стратегия в данном случае состояла бы в отведении для каждой фиксированной страницы одной страницы в файле подкачки во время фиксации страницы. Это означало бы, что всегда есть место, куда записать каждую фиксированную страницу, если потребуется удалить ее из памяти. Недостаток такой стратегии заключается в том, что при этом может потребоваться файл подкачки размером со всю виртуальную память всех процессов. На большой системе, которой редко требуется выгрузка виртуальной памяти на диск, такой подход приведет к излишнему расходованию дискового пространства.

Чтобы не тратить пространство на диске понапрасну, в Windows 2000 фиксированным страницам, у которых нет естественного места хранения на диске (например, страницам стека), не выделяются страницы на диске до тех пор, пока не настанет необходимость их выгрузки на диск. Такая схема усложняет систему, так как во время обработки страничного прерывания может понадобиться обращение к файлам, в которых хранится информация о соответствии страниц, а чтение этих файлов может вызвать дополнительные страничные прерывания. С другой стороны, для страниц, которые никогда не выгружаются, пространства на диске не требуется.

Подобный выбор (усложнение системы или увеличение производительности и дополнительные функции), как правило, разрешается в пользу последнего, так как достоинства лучшей производительности и большего числа функций очевидны, тогда как недостатки усложнения системы (сложность поддержки и увеличение частоты сбоев) бывает сложно учесть. У свободных и зарезервированных страниц никогда не бывает теневых страниц на диске и обращение к ним всегда приводит к страничным прерываниям.

Теневые страницы на диске организованы в один или несколько файлов подкачки. Может быть организовано до 16 файлов подкачки, для повышения производительности операций ввода-вывода они могут быть распределены по отдельным дискам, которых также может быть до 16. У каждого файла есть начальный размер и максимальный размер, до которого он может вырасти при необходимости. Эти файлы могут сразу быть созданы максимального размера во время установки системы, чтобы уменьшить вероятность их сильной фрагментации, но с помощью панели управления позднее можно создать новые файлы. Операционная система следит за тем, какие виртуальные страницы на какую часть файла подкачки отображаются. Страницы, содержащие исполняемый текст программ, не дублируются в файлах подкачки. В файлах подкачки хранятся только изменяемые страницы.

В Windows 2000, как и во многих версиях UNIX, файлы могут отображаться напрямую на области виртуального адресного пространства (то есть занимать множество соседних страниц). После того как файл отображен на адресное

пространство, он может читаться и писаться при помощи обычных команд обращения к памяти. Отображаемые на память файлы реализуются тем же способом, что и фиксированные страницы, но теневые страницы хранятся не в файле подкачки, а в файле пользователя. Поэтому при отображении файла на память версия файла, находящаяся в памяти, может отличаться от дисковой версии (вследствие записи в виртуальное адресное пространство). Однако когда отображение файла прекращается или файл принудительно выгружается на диск, дисковая версия снова приводится в соответствие с последними изменениями файла в памяти.

В Windows 2000 два и более процессов могут одновременно отображать на свои виртуальные адресные пространства, возможно, в различные адреса, одну и ту же часть одного и того же файла, как показано на рис. 2.14 (Файл lib.dll на рисунке отображен одновременно на два адресных пространства.) Читая и записывая слова памяти, процессы могут общаться друг с другом и передавать друг другу информацию с очень большой скоростью, так как копирование при этом не требуется. У различных процессов могут быть различные права доступа. Поскольку все процессы, использующие отображаемый на память файл, совместно используют одни и те же страницы, изменения, произведенные одним процессом, немедленно становятся видимыми для всех остальных процессов, даже если файл на диске еще не был обновлен. Также предпринимаются меры, благодаря которым процесс, открывающий файл для нормального чтения, видит текущие страницы в ОЗУ, а не устаревшие страницы с диска.

Следует отметить, что при совместном использовании двумя программами одного файла DLL может возникнуть проблема, если одна из программ изменит статические данные файла. Если не предпринять специальных действий, то другой процесс увидит измененные данные, что, скорее всего, не соответствует намерениям этого процесса. Эта проблема решается таким способом: все отображаемые страницы помечаются как доступные только для чтения, хотя в то же время некоторые из них тайно помечаются как в действительности доступные и для записи. Когда к такой странице происходит обращение операции записи, создается приватная копия этой страницы и отображается на память. Теперь в эту страницу можно писать, не опасаясь задеть других пользователей или оригинальную копию на диске. Такая техника называется копированием при записи.

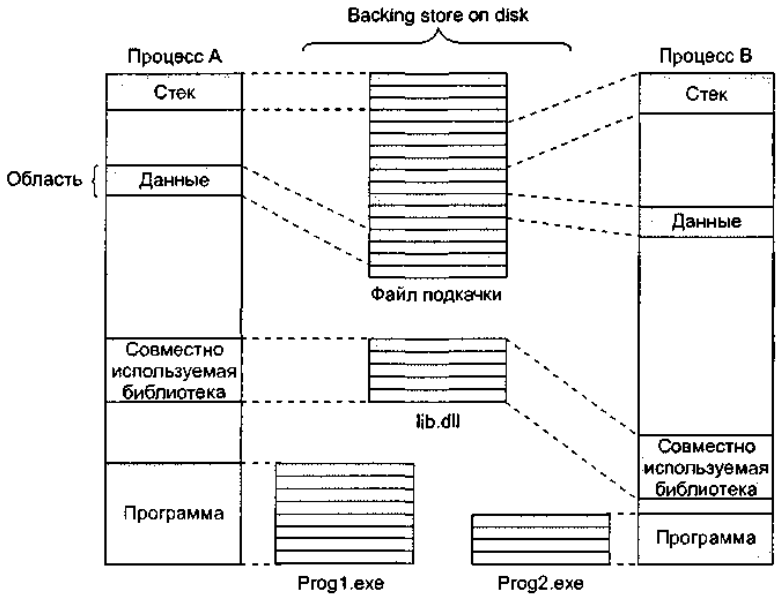


Рисунок 2.14 – Отображаемые на память области с их теневыми страницами на диске

Следует отметить, что и в случае, если текст программы отображается на два адресных пространства по различным адресам, также возникают проблемы с адресацией. Что будет, если первой командой будет JMP 300? Если процесс 1 отобразит эту программу на адрес 65 536, программа легко может быть настроена на новый адрес заменой этой команды командой JMP 65836. Но что будет, если второй процесс отобразит эту программу с адреса 131 072? Вместо передачи управления по адресу 131 372 команда JMP 65836 передаст его по адресу 65 836, после чего вся программа будет работать неверно. Решение заключается в использовании в совместно используемой программе только относительных смещений вместо абсолютных виртуальных адресов. К счастью, у большинства компьютеров есть команды, использующие относительную адресацию, а также команды, использующие абсолютную адресацию. Компиляторы могут использовать относительную адресацию, но они должны знать заранее, какой тип адресации использовать. Как правило, относительная адресация не используется постоянно, так как при этом снижается эффективность программы. Тип используемой адресации обычно задается ключом компиляции. Техника создания участка программы, который может работать независимо от адреса без настройки, называется PIC (Position Independent Code — позиционно-независимый код).

Реализация управления памятью

В операционной системе Windows 2000 поддерживается подгружаемое по требованию одинарное линейное 4-гигабайтное адресное пространство для каждого процесса. Сегментация в любой форме не поддерживается. Теоретически размер страниц может быть любой степенью двух, вплоть до 64 Кбайт. На компьютерах с процессором Pentium страницы имеют фиксированный размер в 4

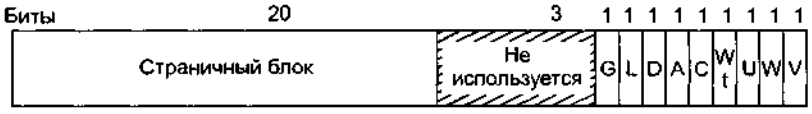
Кбайт. На компьютерах с процессором Itanium они могут быть 8 или 16 Кбайт. Кроме того, сама операционная система может использовать страницы по 4 Мбайт, чтобы снизить размеры таблицы страниц.

В отличие от планировщика, выбирающего отдельные потоки для запуска и не заботящегося о процессах, менеджер памяти занимается исключительно процессами и не беспокоится о потоках. В конце концов, именно процессы, а не потоки владеют адресным пространством, которым занимается менеджер памяти. При выделении области виртуального адресного пространства, в котором хранится информация о диапазоне отображаемых адресов, файле резервного хранения и смещении в файле для отображаемой части файла, а также режим доступа. Когда происходит обращение к первой странице, создается каталог таблиц страниц, а указатель на нее помещается в описатель виртуальной памяти. Адресное пространство полностью описывается списком своих описателей виртуальной памяти. Такая схема позволяет поддерживать несплошные адресные пространства, так как неиспользуемые области между отображаемыми областями не потребляют ресурсов.

Обработка страничных прерываний

В операционной системе Windows 2000 опережающая подкачка страниц не используется ни в каком виде. Когда запускается процесс, в памяти не находится ни одной страницы процесса. При каждом страничном прерывании происходит передача управления ядру (которое понимается так, как изображено на рис. 2.14). Ядро формирует машинно-независимый описатель, в который помещается информация о том, что случилось, и передает его части исполняющей системы, выполняющей функции менеджера памяти. Менеджер памяти проверяет полученный описатель на корректность. Если страница, вызвавшая прерывание, попадает в фиксированную или зарезервированную область, он ищет адрес в списке описателей виртуальной памяти, находит (или создает) таблицу страниц и ищет в ней соответствующий элемент.

Элементы таблицы страниц различаются в разных архитектурах. Для компьютеров с процессором Pentium элемент таблицы для отображаемой страницы показан на рис. 2.15. У неотображаемых страниц также есть записи в таблице, но их формат несколько отличается. Например, если неотображаемая страница должна быть обнулена перед употреблением, этот факт отражается в таблице.



G: Глобальная страница для всех процессов
L: Большая (4 Мбайт) страница
D: Страница «грязная»
A: К странице были обращения
C: Кэширование разрешено/запрещено
Wt: Сквозная запись (без кэширования)
U: К странице возможен доступ в режиме пользователя
W: Запись в страницу разрешена
V: Действительная запись таблицы

Рисунок 2.15 – Запись таблицы для отображаемой страницы на компьютере с процессором Pentium

Для алгоритма подкачки наиболее важными битами записи таблицы страниц являются биты A и D. Они устанавливаются аппаратно и позволяют отслеживать наличие обращений к странице и записи в нее с момента последнего сброса этих битов. Страничные прерывания подразделяются на пять категорий:

- Страница, к которой было обращение, не является фиксированной.
- Произошло нарушение защиты.
- Запись в совместно используемую страницу.
- Стеку требуется дополнительная память.
- Страница, к которой было обращение, является фиксированной, но в настоящий момент она не загружена в память.

Первый и второй случаи представляют собой фатальные ошибки, которые не могут быть исправлены или проигнорированы. У третьего случая симптомы схожи со вторым (попытка записи в страницу, для которой разрешено только чтение), но лечение этого случая возможно. В этом случае страница копируется в новый физический страничный блок, после чего для копии разрешается чтение/запись. Таким образом, работает копирование при записи. (Если совместно используемая страница помечена как доступная для записи во всех процессах, использующих ее, страничного прерывания при записи в такую страницу не возникает и копии при записи не возникает также.) В четвертом случае требуется выделение нового страничного блока и его отображение. Однако правила безопасности требуют, чтобы эта страница содержала только нули, что не позволяет новому процессу узнать, чем занимался предыдущий владелец страницы. Таким образом, нужно найти страницу, содержащую одни нули или, если это невозможно, нужно выделить другой страничный блок и обнулить его на месте. Наконец, пятый случай представляет собой нормальное страничное прерывание. Менеджер памяти находит страницу на диске и считывает ее в память.

Фактический механизм получения и отображения страниц весьма стандартен, поэтому мы не станем обсуждать здесь этот вопрос. Следует только отметить, что операционная система Windows 2000 не читает отдельные страницы прямо с диска. Вместо этого считывается несколько последовательных страниц, как правило, от 1 до 8, чтобы минимизировать количество обращений к диску. Для страниц, содержащих код программы, используются серии из большего числа страниц, чем при считывании страниц данных.

Управление физической памятью

В операционной системе Windows 2000 свободные страницы учитываются в четырех списках. Теперь настала пора познакомиться с ними поближе. Каждая страница памяти находится в одном или нескольких рабочих наборах или в одном из этих четырех списков, показанных на рис. 2.16. В списке чистых (резервных) и

в списке грязных (модифицированных) страниц учитываются страницы, которые недавно были удалены из рабочих наборов, но все еще находятся в памяти и все еще ассоциированы с процессами, использовавшими их. Различие между ними заключается в том, что у чистых страниц есть копия на диске, тогда как у модифицированных страниц таких копий нет, и эти страницы еще предстоит сохранить. В список свободных страниц входят чистые страницы, уже не ассоциированные ни с какими процессами. В список обнуленных страниц входят страницы, не ассоциированные ни с какими процессами и заполненные нулями. Пятый список состоит из физически дефектных страниц памяти. Это гарантирует, что эти страницы ни для чего не используются.

Страницы перемещаются между рабочими наборами и различными списками менеджером рабочих наборов и другими потоками-демонами ядра. Рассмотрим эти переходы. Когда менеджер рабочих наборов удаляет страницу из рабочего набора, страница попадает на дно списка чистых страниц или списка модифицированных страниц в зависимости от своего состояния. Этот переход показан на рисунке как (1). В обоих списках хранятся действительные страницы, поэтому если происходит страничное прерывание и требуется одна из этих страниц, она удаляется из списка и возвращается в свой рабочий набор без операции дискового ввода-вывода (2). Когда процесс завершает свою работу, то все его страницы, которые не используются другими процессами, попадают в список свободных страниц (3). Эти страницы уже не ассоциированы с каким-либо процессом и не могут возвращаться в рабочие наборы по страничному прерыванию.

Другие переходы вызываются другими демонами. Раз в 4 сек. запускается поток свопера в поисках процесса, все потоки которого бездействовали в течение определенного интервала времени. Если ему удается найти такие процессы, он открепляет стеки этих процессов и перемещает страницы процессов в списки чистых и грязных страниц (1).

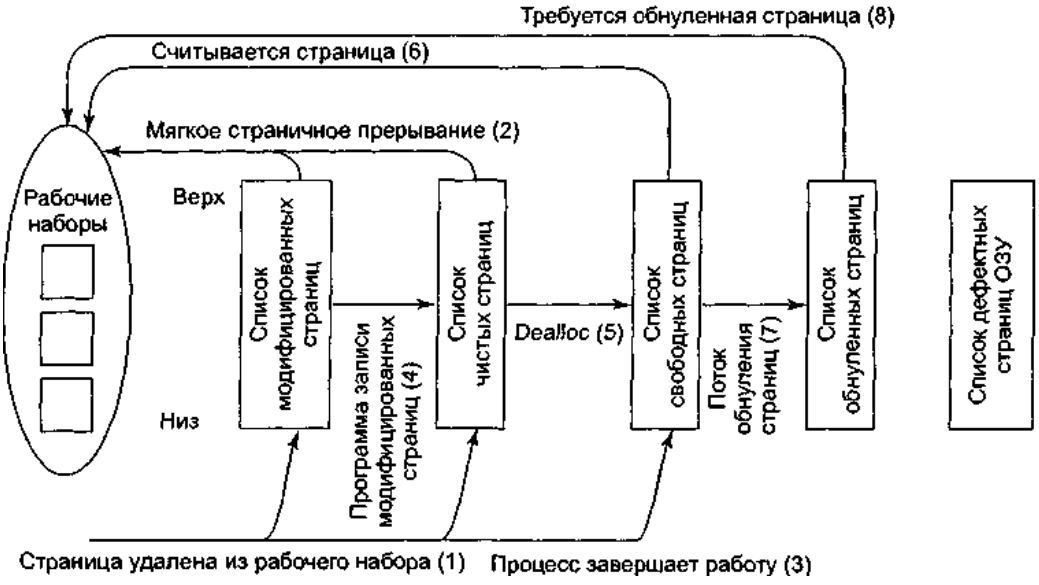


Рисунок 2.16 – Списки страниц и переходы между ними

Два других демона, демон записи отображенных страниц и демон записи модифицированных страниц, просыпаются время от времени, чтобы проверить, достаточно ли чистых страниц. Если количество чистых страниц ниже определенного уровня, они берут страницы из верхней части списка модифицированных страниц, записывают их на диск, а затем помещают их в список чистых страниц (4). Первый демон занимается записью в отображаемые файлы, а второй пишет страницы в файлы подкачки. В результате их деятельности грязные страницы становятся чистыми.

Причина наличия двух демонов, занимающихся очисткой страниц, заключается в том, что отображаемый на память файл может вырасти в результате записи в него. При этом росте потребуются новые свободные блоки диска. Отсутствие в памяти свободного места для записи в него страниц может привести к взаимоблокировке. Второй поток может вывести ситуацию из тупика, записывая страницы в файл подкачки, который никогда не увеличивается в размерах. Никто и не говорил, что операционная система Windows 2000 проста.

Обсудим другие переходы на рис. 2.16. Если процесс освобождает страницу, эта страница более не связана с процессом и может быть помещена в список свободных страниц (5), если только она не используется совместно другими процессами. Когда страничное прерывание требует страничный блок, чтобы поместить в него страницу, которая должна быть считана, этот блок по возможности берется из списка свободных страниц (6). Не имеет значения, что эта страница может все еще содержать конфиденциальную информацию, так как вся она будет тут же целиком перезаписана. При увеличении стека ситуация складывается иная. В этом случае требуется пустой страничный блок и правила безопасности требуют, чтобы страница содержала все нули. По этой причине другой демон ядра, поток обнуления страниц, работает с минимальным приоритетом, стирая содержимое страниц в списке свободных страниц и помещая их в список обнуленных страниц (7). Когда центральный процессор простаивает и в списке свободных страниц есть страницы, поток обнуления страниц может обнулять их, так как обнуленная страница более полезна, чем просто свободная страница.

Наличие всех этих списков приводит к необходимости принятия некоторых стратегических решений. Например, предположим, что страница должна быть считана с диска, а список свободных страниц пуст. Теперь система вынуждена выбирать между чистыми страницами из списка резервных страниц (которые могут потребоваться при очередном страничном прерывании) и пустыми страницами из списка обнуленных страниц (в результате работало обнулению страниц окажется выполненной впустую). Что лучше? Если центральный процессор ничем не занят, а обнуляющий страницы поток запускается часто, лучше взять обнуленную страницу, так как в них нет недостатка. Однако если центральный процессор всегда занят, а диск по большей части простаивает, лучше взять страницу из списка

резервных страниц, чтобы избежать излишних затрат процессорного времени на обнуление еще одной страницы, когда вырастет стек.

Еще одна загадка. Насколько агрессивно должны демоны перемещать страницы из списка грязных страниц в список чистых страниц? Лучше иметь большое количество чистых страниц, чем большое количество грязных страниц, так как чистую страницу можно использовать мгновенно. Однако агрессивная политика очистки страниц означает большее количество операций дискового ввода-вывода; кроме того, есть шанс, что только что очищенная страница снова будет затребована в рабочий набор страничным прерыванием и снова испачкана.

В операционной системе Windows 2000 конфликты подобного рода разрешаются при помощи сложных эвристических алгоритмов, угадывания, учета предыстории, правил большого пальца и настройки параметров, устанавливаемых системным администратором. Более того, эта программа настолько сложна, что разработчики не любят менять в ней что-либо из-за страха сломать в системе какой-нибудь фрагмент, структуру и назначение которого сегодня уже никто не понимает.

Для отслеживания всех страниц и всех списков операционная система Windows 2000 содержит базу данных страничных блоков, состоящую из записей по числу страниц ОЗУ (рис. 2.17). Эта таблица проиндексирована по номеру физического страничного блока. Записи таблицы имеют фиксированную длину, но для различных типов записей используются различные форматы (например, для действительных записей и для недействительных). Действительные записи содержат информацию о состоянии страницы, а также счетчик, хранящий число ссылок на эту страницу в таблицах страниц. Этот счетчик позволяет системе определить, когда страница уже более не используется. Если страница находится в рабочем наборе, то в записи также указывается номер рабочего набора. Кроме того, в записи содержится указатель на таблицу страниц, в которой есть указатель на эту страницу (если такая таблица страниц есть). Страницы, используемые совместно, учитываются особо. Также запись содержит ссылку на следующую страницу в списке (если такая есть) и различные другие поля и флаги, такие как «страница читается», «страница пишется» и т. д.

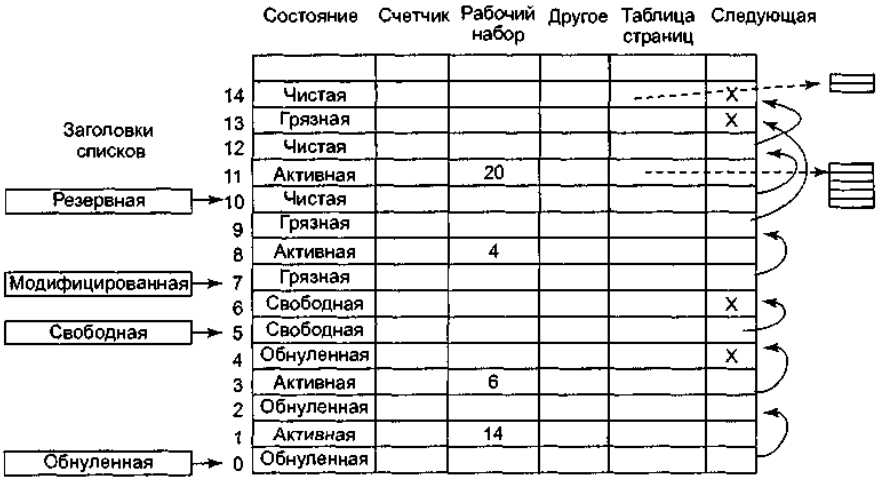


Рисунок 2.17 – Некоторые основные поля базы данных страничных блоков

Итак, управление памятью представляет собой очень сложную подсистему с большим количеством структур данных, алгоритмов и эвристических методов. Во многом она является саморегулируемой, но у нее есть также множество кнопок, на которые может нажимать системный администратор, чтобы влиять на производительность системы. Увидеть эти кнопки и связанные указатели можно при помощи различных программ, входящих в состав упоминавшихся ранее разнообразных наборов инструментальных средств. Вероятно, важнее всего здесь помнить, что управление памятью в реальных системах намного сложнее простого алгоритма подкачки, вроде алгоритма часов или алгоритма старения.

2.7 СИСТЕМНЫЕ ФУНКЦИИ УПРАВЛЕНИЯ ПАМЯТЬЮ В ОПЕРАЦИОННЫХ СИСТЕМАХ MICROSOFT WINDOWS

Интерфейс Win32 API содержит множество функций, позволяющих процессу явно управлять своей виртуальной памятью. Все они работают с областью, состоящей либо из одной страницы, или с двумя или более страницами, располагающимися последовательно в виртуальном адресном пространстве.

Наиболее важные из них описаны ниже в таблице 2.1.

Таблица 2.1 – Основные функции Win32 управления виртуальной памятью

Функция	Описание
VirtualAlloc	Зарезервировать или зафиксировать участок виртуальной памяти из нескольких страниц.
VirtualFree	Освободить участок или отменить его фиксацию.
VirtualProtect	Изменить режим доступа (чтение/запись/выполнение) к области.
VirtualQuery	Узнать состояние области.
VirtualLock	Сделать область резидентной в памяти (то есть запретить ее выгрузку).
VirtualUnlock	Разрешить выгрузку области.
CreateFileMapping	Создать объект отображаемого файла и (по желанию) присвоить ему имя.
MapViewOfRJe	Отобразить файл (часть файла) на адресное пространство.
UnmapViewOfFile	Удалить отображаемый файл из адресного пространства.
OpenFileMapping	Открыть созданный ранее объект отображаемого файла.

Первые четыре функции API используются для выделения и освобождения областей виртуального адресного пространства, изменения их режима защиты и получения информации о текущем режиме. Выделенные области всегда начинаются с 64-килобайтных границ, что сводит к минимуму проблемы переноса

системы на компьютеры будущего с большим размером страниц (до 64 Кбайт), чем у нынешних машин. Количество выделенного адресного пространства может быть меньше, чем 64 Кбайт, но оно должно состоять из целого числа страниц. Следующие две функции API дают процессу возможность зафиксировать страницы в памяти, запрещая их выгрузку, и отменить их фиксацию. Такая возможность может быть полезной, например, для программы реального времени. Операционной системой накладывается предел на количество фиксируемых страниц. На самом деле фиксированные страницы могут быть удалены из памяти, но только в том случае, когда весь процесс выгружается на диск. Когда процесс снова загружается в память, все его зафиксированные страницы также загружаются, прежде чем поток получает управление. Хотя они и не показаны в таблице 2.1, в операционной системе Windows 2000 также есть функции API, позволяющие процессу получать доступ к виртуальной памяти других процессов, которыми он может управлять (то есть тех, от которых у него есть дескриптор).

Последние четыре функции API, перечисленные в таблице, управляют отображением файлов на адресное пространство памяти. Чтобы отобразить файл на адресное пространство памяти, сначала следует создать объект отображения при помощи функции CreateFileMapping. Эта функция возвращает дескриптор объекта отображения, а также может ввести имя объекта в файловую систему, чтобы другие процессы могли пользоваться этим объектом. Следующие две функции включают и выключают отображение файла на адресное пространство памяти. Последняя функция в таблице может использоваться процессом для отображения на память файла, уже использующегося подобным образом другим процессом. Таким образом, несколько процессов могут совместно использовать области своих виртуальных адресных пространств. Эта техника позволяет им записывать данные в ограниченные области памяти других процессов.

3 УПРАВЛЕНИЕ ПРОЦЕССАМИ

ДЕ5: ТЕОРЕТИЧЕСКИЕ ОСНОВЫ УПРАВЛЕНИЯ ПРОЦЕССАМИ В МНОГОЗАДАЧНЫХ ОПЕРАЦИОННЫХ СИСТЕМАХ

3.1 МНОГОЗАДАЧНОСТЬ В ОПЕРАЦИОННЫХ СИСТЕМАХ

Процесс – это наименьшая часть ОС, реально претендующая на ресурсы вычислительной системы (это активная задача в ОС).

Понятие "процесс" появилось в 60-х гг. при разработке ОС Multics, потом несколько раз претерпевало изменения.

У понятия процесс существует много определений, и ни одно из них не полное. Под процессом понимают:

1) Процесс – некоторые данные

2) Процесс – некоторый алгоритм, который оперирует этими данными.

Для операционной системы процесс представляет собой единицу работы, заявку на потребление системных ресурсов.

Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

Состояние процессов

Процессы во время своей жизни могут существовать во множестве состояний, которые характеризуют те или иные действия, предпринимаемые ОС по отношению к данному процессу. Процесс является "пассивным" и практически не принимает участия в управлении самим собой.

В многозадачной (многопроцессной) системе процесс может находиться в одном из четырёх состояний:

- ВЫПОЛНЕНИЕ - активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;
- ОЖИДАНИЕ – это состояние, в котором процессы не могут продолжать своё выполнение, вследствие того, что необходим некоторый ресурс, занятый в данный момент другим процессом.
- ГОТОВНОСТЬ – в этом состоянии находятся процессы, готовые к немедленному выполнению, т.е. готовые в любой момент принять

управление над ЦП, как только обеспеченно ОС. Это состояние потенциальной активности, т.е. в любой момент процесс может быть переведён в активное состояние.

– ПАССИВНОЕ - процесс загружен в память, но ещё не инициализирован ОС для своего выполнения. Фактически в данном случае процесс уже контролирует один из ресурсов ОС – память, но не может предпринимать никаких активных действий.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе. Типичный граф состояний процесса показан на рисунке 3.1.

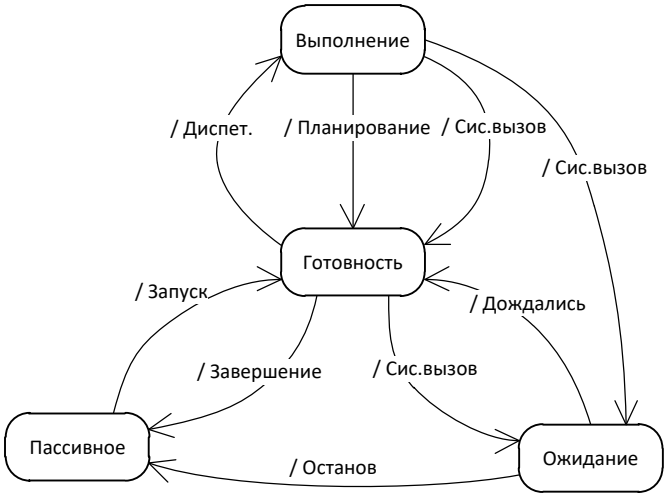


Рисунок 3.1 – Диаграмма состояний процессов

Эти четыре состояния являются базовыми и могут разбиваться на подсостояния, что с одной стороны, улучшает процесс планирования, а с другой стороны, увеличивает "накладные расходы" по планированию.

Кроме того процесс можно рассматривать как систему массового обслуживания, т.е. если мы знаем интенсивности переходов процесса из одного состояния в другое, то мы сможем определить вероятность его нахождения в одном из состояний.

Система называется зависимой по вводу-выводу, если большее число процессов из очереди на ожидания зависят от операций, связанных с внешними носителями. (большая часть процессов в системе наиболее вероятно находятся в состоянии ожидания)

ОС называется зависимой по вычислениям, если большинство процессов очереди ожидания зависят от других процессов, выполняющих некоторые вычисления.

Контекст и дескриптор процесса

Представителями процессов о ОС является блок состояния процесса – PCB (Process Control Block), который содержит все необходимые ОС сведения о процессе, такие как:

- имя процесса;
- уникальный идентификатор процесса PID(Process Identifier);
- текущее состояние процесса;
- указатели на все ресурсы, занятые процессом (открытые файлы, незавершённые операции ввода-вывода);
- состояние регистров и программного счётчика IP(Instruction Pointer).

Этот управляемый операционный блок в принципе является всем, что необходимо системе для управления этим процессом. Доступ к нему должен быть быстр, поэтому во многих аппаратных платформах реализованы специальные аппаратные регистры, в которых храниться PCB процессов, к которым происходит активное обращение.

Кроме PCB, в операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента и т.д. В некоторых ОС (например, в ОС UNIX) информацию такого рода, используемую ОС для планирования процессов, называют дескриптором процесса (handle).

Очереди процессов представляют собой дескрипторы отдельных процессов, объединенные в списки. Таким образом, каждый дескриптор, кроме всего прочего, содержит, по крайней мере, один указатель на другой дескриптор, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

Алгоритмы планирования процессов

Планирование процессов включает в себя решение следующих задач:

- определение момента времени для смены выполняемого процесса
- выбор процесса на выполнение из очереди готовых процессов;
- переключение контекстов "старого" и "нового" процессов.

Алгоритмы планирования делятся на два класса:

- алгоритмы, основанные на квантовании
- алгоритмы, основанные на приоритетах (системы с явным приоритетом).

Алгоритмы бесприоритетного планирования

В таких алгоритмах смена активного процесса происходит, если: процесс завершился и покинул систему, произошла ошибка, процесс перешел в состояние ОЖИДАНИЕ, исчерпан квант процессорного времени, отведенный данному процессу.

Процесс, который исчерпал свой квант, переводится в состояние ГОТОВНОСТЬ и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых процессов. Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется в системах разделения времени.

Обычно выделяют два бесприоритетных алгоритма:

- Алгоритм циклического планирования. Алгоритм заключается в том, что последовательно выбираются все процессы из очереди (подряд).
- Алгоритм случайного выбора. Процессы выбираются из очереди случайным образом. Для данного алгоритма требуется наличие качественного генератора случайных чисел с равномерным распределением.

Алгоритмы, основанные на приоритетах

Другая группа алгоритмов использует понятие "приоритет" процесса. Предполагается, что на выполнение выбирается процесс с наибольшим “приоритетом”. Вводится т.н. “весовая функция”, которая в общем случае для всех процессов в момент планирования. Результат вычисления этой функции есть приоритет.

Приоритет - это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности, процессорного времени: чем выше приоритет, тем выше привилегии. Приоритет может оставаться фиксированным на протяжении всей жизни процесса либо изменяться во времени в соответствии с некоторым законом. В последнем случае приоритеты называются динамическими.

Существует две разновидности приоритетных алгоритмов:

- алгоритмы, использующие относительные приоритеты.
- алгоритмы, использующие абсолютные приоритеты.

На выполнение выбирается процесс, имеющий наивысший приоритет. В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ОЖИДАНИЕ (или же произойдет ошибка, или процесс завершится). В системах с абсолютными

приоритетами выполнение активного процесса прерывается еще при одном условии: если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности.

В случае отсутствия в системе реальных процессов, как правило, управление передаётся специальному процессу, который называется Idle (бездельник). Этот процесс имеет наименьший приоритет в системе по определению и в случае добавления в систему любого процесса, он будет прерван.

Многие операционные системы построены на смешанных алгоритмах. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора процесса из очереди готовых определяется приоритетами процессов.

По сложности различают различные следующие алгоритмы:

а) Алгоритм простого приоритетного планирования

В общем случае предполагается, что на выполнение будет выбран процесс с максимальным приоритетом. При этом процессы с одинаковым приоритетом выбираются согласно одного из алгоритмов бесприоритетного планирования.

Реализуется в системах, ориентированных на интерфейс с пользователем. В этих системах процессы обычно ожидают некоторых событий, и в этом случае планирование осуществляется так: выбирается процесс с максимальным приоритетом и проверяется, есть ли события, которые этот процесс должен обработать, и если событие есть, то управление передаётся этому процессу, а если этого события нет ни для одного процесса с высшим приоритетом, ОС начинает просматривать процессы с более низким приоритетом.

В системах, ориентированных на вычисления планирование осуществляется следующим образом: на некотором промежутке времени, размер которого достаточно большой по сравнению с размером кванта, процессы с большим приоритетом, получают статистически больше квантов, чем процессы с меньшим приоритетом.

б) Алгоритм политического планирования

Приоритет задается в виде функции, определяющей приоритетность процесса: это функция, аргументом которой является разница между количеством обещанных данному процессу ресурсов и количеством реально полученных процессом ресурсов.

$Ф(n_{обещали} - n_{получил})$

Обещает ресурсы и процессорное время ОС, согласно приоритетам процесса. Т.о. у процесса всегда существует некоторый “политический вес”, который практически является приоритетом.

в) Алгоритм адаптивного планирования

Является одним из самых неудобных и трудоёмких для ОС.

Предполагает, как и предыдущих алгоритмах, наличие весовой функции, но в этом алгоритме ОС мало знать приоритет процесса. ОС превращается в некоторый надзирательный орган, собирающий всю информацию о процессе, т.е. весовая функция зависит от приоритета процесса, от того, какое количество ресурсов было выделено процессу и от динамики развития использования ресурсов за последний интервал времени. Т.е., после кванта времени выполнения процесса для него рассчитываются некоторые критические характеристики:

- какое количество памяти ему потребуется на следующем шаге;
- сколько ему потребуется процессорного времени;
- какие внешние ресурсы ему предположительно потребуются.

Чем меньше будут эти предполагаемые требования, тем выше будет приоритет процесса.

Этот алгоритм хорошо зарекомендовал себя для интерактивных систем, т.е. в системах, где пользователь, запуская небольшую задачу, ожидает, что она быстро выполниться, и наоборот.

3.2 СИНХРОНИЗАЦИЯ И ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ

Синхронизация процессов

Под синхронизацией мы будем понимать процессы и средства, направленные на организацию коллективного выполнения процессами одной или нескольких задач. В ходе такого выполнения всегда требуется договариваться о единовременности выполнения или невыполнения некоторых действия, о последовательности выполнения действий и т.д., что и является основным назначением синхронизации.

Рассмотрим основные способы синхронизации процессов.

Критическая секция

Важным понятием синхронизации процессов является понятие "критическая секция" программы. Критическая секция - это часть программы, в которой осуществляется доступ к разделяемым данным. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо обеспечить, чтобы в каждый момент в критической секции, связанной с этим ресурсом, находился максимум один процесс. Этот прием называют взаимным исключением.

Используют блокирующие переменные. С каждым разделяемым ресурсом связывается двоичная переменная, которая принимает значение 1, если ресурс свободен (то есть ни один процесс не находится в данный момент в критической секции, связанной с данным процессом), и значение 0, если ресурс занят.

Если все процессы написаны с учётом значения этой блокирующей переменной, то взаимное исключение гарантируется. Операция проверки и установки блокирующей переменной должна быть неделимой.

Реализация критических секций с использованием блокирующих переменных имеет существенный недостаток: в течение времени, когда один процесс находится в критической секции, другой процесс, которому требуется тот же ресурс, будет выполнять рутинные действия по опросу блокирующей переменной, бесполезно тратя процессорное время. Для устранения таких ситуаций может быть использован так называемый аппарат событий. С помощью этого средства могут решаться не только проблемы взаимного исключения, но и более общие задачи синхронизации процессов. Используются системные функции которые условно можно условно назвать WAIT(x) и POST(x), где x - идентификатор некоторого события. Если ресурс занят, то процесс не выполняет циклический опрос, а вызывает системную функцию WAIT(D), здесь D обозначает событие, заключающееся в освобождении

ресурса D. Функция WAIT(D) переводит активный процесс в состояние ОЖИДАНИЕ и делает отметку в его дескрипторе о том, что процесс ожидает события D. Процесс, который в это время использует ресурс D, после выхода из критической секции выполняет системную функцию POST(D), в результате чего операционная система просматривает очередь ожидающих процессов и переводит процесс, ожидающий события D, в состояние ГОТОВНОСТЬ.

Обобщающее средство синхронизации процессов предложил Дейкстра, который ввел два новых примитива. В абстрактной форме эти примитивы, обозначаемые P и V, оперируют над целыми неотрицательными переменными, называемыми семафорами. Пусть S такой семафор. Операции определяются следующим образом:

V(S) : переменная S увеличивается на 1 одним неделимым действием; выборка, инкремент и запоминание не могут быть прерваны, и к S нет доступа другим процессам во время выполнения этой операции.

P(S) : уменьшение S на 1, если это возможно. Если S=0, то невозможно уменьшить S и остаться в области целых неотрицательных значений, в этом случае процесс, вызывающий P-операцию, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также является неделимой операцией.

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную. Операция P заключает в себе потенциальную возможность перехода процесса, который ее выполняет, в состояние ожидания, в то время как V-операция может при некоторых обстоятельствах активизировать другой процесс, приостановленный операцией P (сравните эти операции с системными функциями WAIT и POST).

Рассмотрим использование семафоров на классическом примере взаимодействия двух процессов, один из которых пишет данные в буферный пул, а другой

считывает их из буферного пула. Пусть буферный пул состоит из N буферов, каждый из которых может

содержать одну запись. Процесс "писатель" должен приостанавливаться, когда все буфера оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, процесс "читатель" приостанавливается, когда все буферы пусты, и активизируется при появлении хотя бы одной записи.

Барьер

Используется, когда надо быть уверенным в том, что несколько процессов завершили свое действие.

Функциональная спецификация барьера включает следующие операции:

- регистрация процесса на барьере;
- удаление процесса с барьера;
- установка признака достижения барьера определенным процессом.

Процессы, достигшие барьер, переводятся в состояние ожидания до тех пор, пока все зарегистрированные на барьере процессы не дойдут до него. В этом случае все процессы переводятся в состояние готовности.

Семафор

Семафор – контролирует доступность нескольких однотипных ресурсов, предоставляя счетчик, хранящий количество ресурсов свободных в настоящий момент. Говорят, что семафор открыт, если счетчик больше 0 и закрыт, если равен 0.

Функциональная спецификация семафора включает следующие операции:

- захватить семафор (если счетчик > 0, то он уменьшится на единицу, если счетчик = 0 – процесс находится в состоянии ожидания до тех пор, пока счетчик не стане равен 1 и, отнимая ее, продолжит работу);
- освободить семафор.

Взаимодействие процессов

Под взаимодействием процессов в нашем курсе мы будем понимать передачу от одного процесса другому данных. И хоть синхронизацию между процессами тоже можно рассматривать как обмен синхронизационной информацией, при взаимодействии мы будем говорить об обмене обсчитываемой информацией, такой как массивы значений, картинки, видео, тексты и т.д.

Перечислим основные способы такого взаимодействия.

Общая память

Общая память является самым быстрым способом передачи данных между процессами. Платой за это является невозможность использования её для обмена между разными процессами в сети.

Общая память образуется путем подключения различных процессов в рамках одной вычислительной системы к одним и тем физическим страницам виртуальной памяти (хотя адреса внутри процессов при этом могут не совпадать).

Этот способ требует строгой синхронизации данных. Типы данных должны совпадать по структуре. Должна совпадать последовательность полей структур данных и выравнивание (размер, с границы которого имеет право начинаться новое поле структуры) должно совпадать.

Сигналы

Один процесс передает сигналы другому (информация, которая передается процессу – это номер сигнала). Сигнал очень похож на прерывание: у него есть номер и он обязателен для исполнения процессом получателем.

Сигналы – это способ взаимодействия доступный только в операционных система UNIX.

Сообщения

Структура данных, которая может передаваться от одного процесса другому. В зависимости от операционной системы и типа сообщения могут быть строго стандартизированы. В сообщении нельзя, как правило, передавать указатель, потому что у процессов разные виртуальные пространства и адресный указатель одного пространства неравен адресному указателю другого процесса.

Каналы (pipe)

Это специального вида файл, открываемый между двумя процессами. Это файл одностороннего доступа с последовательным доступом, как правило, ограничен в объеме.

В разных системах каналы могут иметь разную реализацию и разные возможности. Так в UNIX-системах канала обеспечивают взаимодействие только между процессами в рамках одного компьютера, а в Windows-системах – это способ в том числе и межсетевого взаимодействия.

3.3 ВЗАИМОБЛОКИРОВКА ПРОЦЕССОВ (ТУПИКИ)

В современной литературе по теории ОС нет единого мнения по поводу того, что является первичным – проблемы взаимодействия процессов, или средства синхронизации и взаимодействия процессов. Первый подход подразумевает, что разработке средств взаимодействия процессов предшествовало осознание проблем

взаимодействия, а второй подход – что проблемы взаимодействия – это то, что было получено в результате того, что были разработаны именно такие средства взаимодействия процессов. В этом курсе лекций мы пойдём по первому пути, т.е. сначала будут изложены проблемы взаимодействия процессов, а потом методы взаимодействия и синхронизации процессов.

Процессам часто нужно взаимодействовать друг с другом, например, один процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает проблема синхронизации процессов, которая может решаться приостановкой и активизацией процессов, организацией очередей, блокированием и освобождением ресурсов.

Будем говорить, что процесс находится в состоянии тупика (clinch) или взаимоблокировки (deadlock), в случае, когда он ожидает одного или более ресурсов, которые заранее никогда не будут выделены или события, которое никогда не произойдёт. Например, в тупик может попасть процесс (поток) ждущий некоторого события, в случае если это событие устанавливается этим же процессом.

После ряда математических исследований в области теории игр было доказано, что для установления тупика достаточно и необходимо выполнение 4-х условий:

- В системе присутствуют монопольные ресурсы, т.е. ресурсы, которыми может пользоваться только один процесс в один момент времени.
- Процессы, уже владея некоторыми ресурсам, могут запрашивать выделение дополнительных ресурсов.
- ОС не может принудительно отобрать у процесса ресурс, выданный ему в монопольное использование, до тех пор, пока процесс сам его не освободит.
- В ОС существует цикл некоторых процессов, каждый из которых владеет каким-либо ресурсом, который запрашивается другим процессом, находящимся в этом цикле (рис 3.2).

Рисунок 3.2 – Пример тупика с циклической зависимостью процессов

Борьба с тупиками

Поскольку тупики были и остаются проблемой индустрии ИТ, были разработаны методы позволяющие в той или иной мере снизить их влияние на работоспособность программного обеспечения. Можно выделить три крупных группы таких методов: недопущение тупиков, их обход и восстановление работоспособности при их возникновении.

Недопущение тупиков

Хавендер и его коллеги доказали, что если нарушить хотя бы одно условие возникновения тупиковых ситуаций, то тупики не могут возникнуть в принципе. Были предложены 3 стратегии по изменению структуры ОС таким образом, чтобы нарушить любое из 3-х условий (последних) возникновения тупиковых ситуаций.

Стратегия №1. Предполагает условие: любой процесс запрашивает все требуемые ресурсы сразу и не продолжает выполнения до тех пор, пока это условие не будет выполнено. Эта стратегия позволяет нарушить второе условие возникновения, но приводит к крайне нерациональному использованию ресурсов вычислительной системы, т.к. занятые ресурсы могут оставаться неиспользуемыми продолжительное время.

Стратегия №2. Если процессу отказывается в выделении ресурса, то он освобождает все выделенные ему ранее ресурсы и перезапрашивает их заново. Таким образом, другие процессы могут захватить монопольные ресурсы, выделенные данному процессу ранее. Данная стратегия нарушает третье условие,

но удобство написания программ при этом страдает самым ужасным образом, т.к. программа превращается в кучу проверок и циклом только с целью «правильно» занять ресурсы.

Стратегия №3. Процесс выдаёт запросы на ресурсы только в том порядке, в котором они ассоциированы в ОС. При этом исключается перекрестный захват ресурсов. Данная стратегия позволяет обойти четвертое условие, но, к сожалению, практически неосуществима на современных вычислительных системах, где список устройств слишком большой и может меняться непосредственно в процессе работы компьютера и программы соответственно. Понятие ассоциации в таком динамическом мире сложно (если не невозможно) поддерживать.

Обход тупика

Этот метод похож на первый, но применяется в тех случаях, когда нельзя применить первый метод. Он является более мягким по отношению к первому методу. ОС учитывает и пытается предсказать ситуации возникновения тупиков и решить проблему до того, как она возникнет.

Обнаружение тупика

Основное назначение метода – обнаружить сам факт присутствия тупика и выявить те процессы и ресурсы, которые участвуют в возникновении тупика, чтобы потом его устранить.

Строится граф, узлы которого ресурсы и процессы, а рёбра – реальные или потенциальные взаимосвязи. Алгоритм заключается в следующем: Для данного графа говорят, что любой процесс может быть редуцирован, если он может освободить используемые ресурсы и получить запрошенные. Производится попытка редуцировать все процессы в графе. Если это возможно, то тупика нет. Нередуцированные процессы и ресурсы составляют цикл, участвующий в создании тупика.

Восстановление работоспособности

Одним из наиболее распространённых способов борьбы с тупиками является останов одного или более процессов, участвующих в тупиковой ситуации, с последующим его перезапуском.

ДЕ6: УПРАВЛЕНИЕ ПРОЦЕССАМИ В СИСТЕМАХ UNIX

3.4 РЕАЛИЗАЦИЯ МНОГОЗАДАЧНОСТИ В СИСТЕМАХ UNIX

В UNIX (и большинстве современных операционных систем, как-то Microsoft Windows, Mac OS X, FreeBSD и Linux) каждая задача представляется как процесс. UNIX способен выполнять много задач одновременно, потому что процессы

поочередно выполняются на центральном процессоре в течение очень непродолжительного промежутка времени.

Процесс - это нечто вроде контейнера, объединяющего выполняемое приложение, его переменные среды, состояние потоков ввода/вывода приложения и параметры процесса, включающие в себя его приоритет и степень использования ресурсов системы. Рисунок 3.3 иллюстрирует понятие процесса.

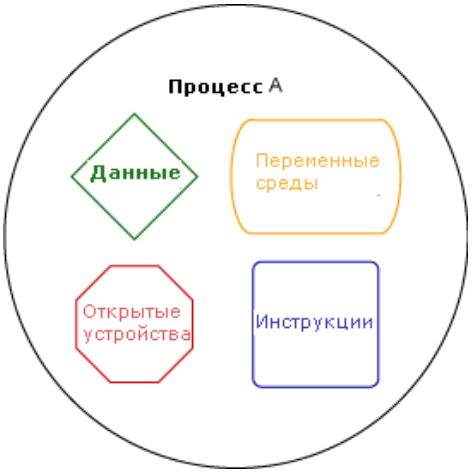


Рисунок 3.3 – Концептуальная модель процесса в UNIX

Можно представлять процесс как независимое государство со своими границами, ресурсами и валовым национальным продуктом.

Каждый процесс имеет своего владельца. Задачи, инициализируемые пользователем, например, его shell и командами, обычно принадлежат этому пользователю. Системные процессы могут принадлежать пользователям с особыми правами или системному администратору root. Например, для повышения безопасности Apache HTTP Server обычно принадлежит выделенному пользователю с именем www, который предоставляет Web-серверу только доступ к файлам, которые ему необходимы для работы.

Владелец процесса может меняться. У одного процесса не может быть одновременно двух владельцев.

Можно представлять процесс как независимое государство со своими границами, ресурсами и валовым национальным продуктом.

Каждый процесс имеет своего владельца. Задачи, инициализируемые пользователем, например, его shell и командами, обычно принадлежат этому пользователю. Системные процессы могут принадлежать пользователям с особыми правами или системному администратору root. Например, для повышения безопасности Apache HTTP Server обычно принадлежит выделенному пользователю с именем www, который предоставляет Web-серверу только доступ к файлам, которые ему необходимы для работы.

Владелец процесса может меняться. У одного процесса не может быть одновременно двух владельцев.

Наконец, каждый процесс имеет привилегии. Обычно привилегии процесса соответствуют статусу его владельца в ОС. Например, если пользователь не может получить доступ к какому-либо файлу через команду shell, то программы, которые он запустит при помощи этой оболочки, наследуют то же ограничение на доступ к этому файлу.

Правило наследования привилегий можно обойти, т.е. процесс может получить большие привилегии, чем его владелец, если запустить процесс командой, в которой помимо всех прочих действий активируется специальный бит setuid или setgid, как показано на примере ls.

Бит setuid можно задать при помощи chmod u+s. Права доступа, заданные при помощи setuid, выглядят так:

```
$ ls -l /usr/bin/top
-rwsr-xr-x     1 root  wheel     83088 Mar 20  2005 top
```

Бит setgid может быть задан при помощи chmod g+s:

```
$ ls -l /usr/bin/top
-r-xr-sr-x   1 root  tty  19388 Mar 20  2005 /usr/bin/wall
```

Процесс setuid-, например запускаемый процесс top, выполняется с привилегиями пользователя, которому принадлежит. Следовательно, если запустить top, то у пользователя будут привилегии администратора. Соответственно, процесс setgid выполняется с привилегиями группы владельцев файла.

Например, в Mac OS X, утилита wall, укороченное от "write all", сокращение от write all (отправляет сообщение на каждый физический или виртуальный терминал), задана с setgid tty (как показано выше). Когда пользователь зашел в систему и выбрал терминал, с которого будет вводить команды (терминал становится стандартным каналом ввода для shell), то он становится владельцем терминала, а tty становится группой владельцев. Ввиду того что wall выполняется с привилегиями группы tty, он может открывать и писать вывод на любой терминал.

В UNIX некоторые процессы выполняются с момента загрузки компьютера до его выключения, но большинство процессов имеет короткий жизненный цикл, ограничивающийся началом и окончанием выполнения задачи. Иногда процесс может быть принудительно завершен. Откуда берутся новые процессы?

Каждый новый процесс в UNIX является перевоплощением существующего процесса. Или, по другому, каждый новый процесс - давайте называть его дочерним - является клоном своего процесса-родителя хотя бы на мгновенье, пока дочерний процесс не начнет выполняться самостоятельно.

Для создания нового процесса используется системный вызов fork. В среде программирования нужно относиться к этому системному вызову как к вызову функции, возвращающей целое значение - идентификатор порожденного процесса, который затем может использоваться для управления (в ограниченном смысле)

порожденным процессом. Реально, все процессы системы UNIX, кроме начального, запускаемого при раскрутке системы, образуются при помощи системного вызова fork.

Вот что делает ядро системы при выполнении системного вызова fork:

- Выделяет память под описатель нового процесса в таблице описателей процессов.
- Назначает уникальный идентификатор процесса (PID) для вновь образованного процесса.
- Образует логическую копию процесса, выполняющего системный вызов fork, включая полное копирование содержимого виртуальной памяти процесса-предка во вновь создаваемую виртуальную память, а также копирование составляющих ядерного статического и динамического контекстов процесса-предка.
- Увеличивает счетчики открытия файлов (процесс-потомок автоматически наследует все открытые файлы своего родителя).
- Возвращает вновь образованный идентификатор процесса в точку возврата из системного вызова в процессе-предке и возвращает значение 0 в точке возврата в процессе-потомке.

3.4 СИСТЕМНЫЕ ФУНКЦИИ РАБОТЫ С МНОГОЗАДАЧНОСТЬЮ В ОПЕРАЦИОННЫХ СИСТЕМАХ UNIX

В данном разделе мы рассмотрим основные функции использования и управления многозадачностью в ОС семейства UNIX.

Функция «fork»

```
pid_t fork(void)
```

Описание

fork создает дочерний процесс, отличающийся от родительского только своим номером (PID) и номером родительского (PPID).

В среде Linux, fork реализован с использованием механизма copy-on-write, так что память выделяется системой только под те страницы памяти, значение которых изменяется по отношению к родительскому процессу.

Возвращаемое значение

В случае успеха, возвращает родителю PID дочернего (созданного) процесса, а дочернему возвращает ноль. В случае неудачи, возвращает -1, а в переменную errorno заносит более детальную информацию об ошибке.

Функция «wait»

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Описание

Функция wait приостанавливает выполнение процесса, ожидая завершения одного из дочерних процессов.

Функция waitpid аналогична функции wait за тем исключением, что можно детально указать завершения каких именно дочерних процессов мы дожидаемся. Это выполняется при помощи параметра pid функции. Допустимые значения параметра pid описаны в таблице 3.1.

Таблица 3.1 – Допустимые значения параметра pid функции wait

Значение параметра	Описание
меньше -1	Ожидаем завершения любого дочернего процесса, номер группы которого равен модулю величины параметра.
равен -1	Ожидаем завершения любого дочернего процесса.
равен 0 (нолю)	Ожидаем завершения любого дочернего процесса, номер группы которого номеру группы родительского процесса.
больше 0 (ноля)	Ожидаем завершения дочернего процесса с указанным номером.

Возвращаемое значение

В случае если дочерний процесс уже завершен, функция тут же возвращает управление, а ресурсы, занятые завершенным дочерним процессом освобождаются.

Функция «clone»

```
int clone(int (*fn)(void *), void *child_stack, int flags, void
*arg);
```

Описание

Поведение функции clone похоже на поведение функции fork. Однако, в отличие от fork, функция позволяет разделять некоторые части контекста исполнения с дочерними процессами. Основное назначение функции – реализация модели многопоточного программирования.

Возвращаемое значение

Аналогично функции fork.

Функции «getpid» и «getppid»

```
pid_t getpid(void);
pid_t getppid(void);
```

Возвращают номер процесса и номер родительского процесса соответственно.

Функция «signal»

```
sighandler_t signal(int signum, sighandler_t handler);
```

Описание

Вызов функции устанавливает в системе новый обработчик для сигнала с указанным номером. Обработчик сигнала, передаваемый через параметр handler может быть как пользовательской функцией, так и одним из предопределенных значений: SIG_IGN – игнорирование сигнала, SIG_DFL – обработчик сигнала по умолчанию.

Возвращаемое значение

В случае успеха возвращает указатель на предыдущий обработчик, установленный для этого сигнала в системе.

В случае ошибки – значение SIG_ERR.

Функция «kill»

```
int kill(pid_t pid, int sig);
```

Описание

Системный вызов kill используется для отправки процессу или группе процессов указанного сигнала. Параметр функции pid может принимать тот же диапазон значений с теми же значениями, что и функция waitpid.

Возвращаемое значение

В случае успеха возвращает 0 (ноль), в случае любой ошибки - значение -1 (и дополнительный код в переменной errno).

Функция «ftok»

```
key_t ftok(const char *pathname, int proj_id);
```

Описание

Используя уникальность файла переданного в параметре pathname и как минимум младшие 8 бит параметра proj_id, функция генерирует уникальное значение ключа.

Подобные ключи часто используются для идентификации различных объектов ядра операционной системы.

Возвращаемое значение

Значение ключа или -1 в случае ошибки.

Функция «shmget»

```
int shmget(key_t key, int size, int shmflg);
```

Описание

Возвращает идентификатор сегмента разделяемой памяти, создавая его в случае необходимости. Для того, чтобы разные программы могли использовать один и тот же сегмент общей памяти, используется уникальное значение ключа. Параметр shmflg может комбинироваться из следующих флагов:

- IPC_CREAT – создать новый сегмент общей памяти;
- IPC_EXCL – используется совместно с флагом IPC_CREAT и генерирует ошибку в случае, если сегмент с таким ключом уже существует;
- произвольные флаги прав доступа к сегменту данных.

Возвращаемое значение

Идентификатор сегмента общей памяти или -1 в случае ошибки.

ДЕ7: УПРАВЛЕНИЕ ПРОЦЕССАМИ В СИСТЕМАХ MICROSOFT WINDOWS

3.6 РЕАЛИЗАЦИЯ И УПРАВЛЕНИЕ МНОГОЗАДАЧНОЙ СРЕДОЙ В СИСТЕМАХ MICROSOFT WINDOWS

В современных полновесных реализациях Windows (Windows 2000, Windows XP, Windows 2003) планировщик ядра выделяет процессорное время потокам. Управление волокнами возложено на приложения пользователя: Windows предоставляет набор функций, с помощью которых приложение может управлять созданными волокнами. Фактически для волокон реализуется невытесняющая многозадачность средствами приложения; с точки зрения операционной системы, все волокна должны быть созданы в рамках потоков (один поток может быть "расщеплен" на множество волокон средствами приложения) и система никак не вмешивается в их планирование.

В Windows определен список событий, которые приводят к перепланированию потоков:

– создание и завершение потока;
– выделенный потоку квант исчерпан;
– поток вышел из состояния ожидания;
– поток перешел в состояние ожидания;
– изменен приоритет потока;
– изменена привязка к процессору.

В целях уменьшения затрат на планирование потоков несколько изменен граф состояний потока. На рис. 3.4 приведен "классический" вид графа состояний задачи, а на рис. 3.5 – граф состояний потока в Windows. Переход из состояния "готовность" в состояние "выполнение" сделан в два этапа - выбранный к выполнению поток подготавливается к выполнению и переводится в состояние "выбран"; эта подготовка может осуществляться до наступления момента перепланирования, и в нужный момент достаточно просто переключить контекст выполняющегося потока на выбранный.

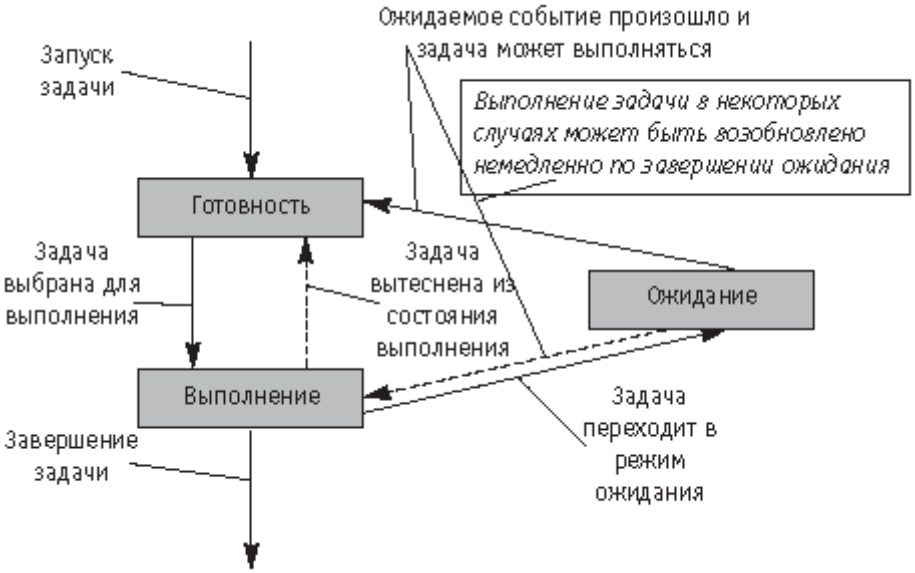


Рисунок 3.4 – Граф состояния процесса в ОС Windows

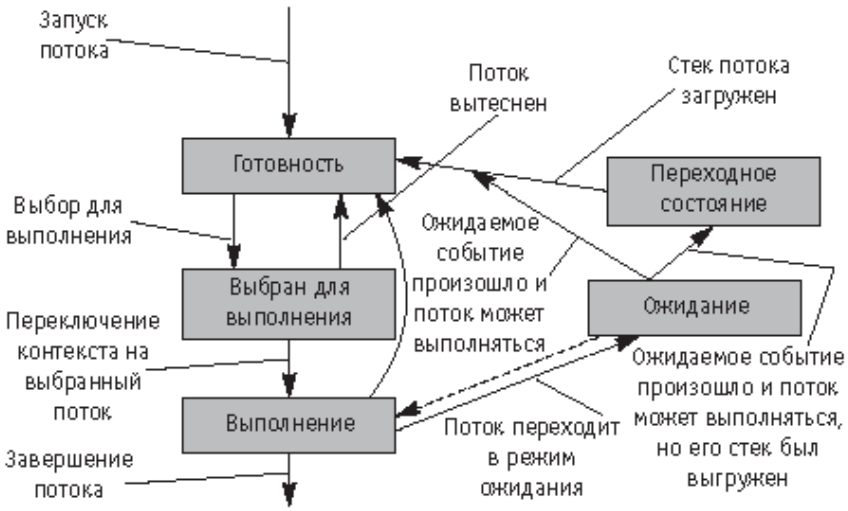


Рисунок 3.5 – Граф состояний потока в ОС Windows

Также в два этапа может происходить переход из состояния "ожидание" в "готовность": если ожидание было долгим, то стек потока может быть выгружен из оперативной памяти. В этом случае поток переводится в промежуточное состояние до завершения загрузки стека - в списке готовых к выполнению потоков находятся только те, которые можно начать выполнять без лишнего ожидания.

При выборе потока для выполнения учитываются приоритеты потоков (абсолютные приоритеты) - система начинает выполнять код потока с наибольшим приоритетом из числа готовых к исполнению.

Процесс выбора потока для выполнения усложняется в случае SMP систем, когда помимо приоритета готового к исполнению потока учитывается, на каком процессоре ранее выполнялся код данного потока.

В Windows выделяют понятие "идеального" процессора - им назначается процессор, на котором запускается приложение в первый раз. В дальнейшем система старается выполнять код потока именно на этом процессоре - для SMP систем это решение улучшает использование кэш-памяти, а для NUMA систем позволяет, по большей части, ограничиться использованием оперативной памяти, локальной для данного процессора. Заметим, что диспетчер памяти Windows при выделении памяти для запускаемого процесса старается учитывать доступность памяти для назначенного процессора в случае NUMA системы.

В многопроцессорной системе используется либо первый простаивающий процессор, либо, при необходимости вытеснения уже работающего потока, проверяются идеальный процессор, последний использовавшийся и процессор с наибольшим номером. Если на одном из них работает поток с меньшим приоритетом, то последний вытесняется и заменяется новым потоком; в противном случае выполнение потока откладывается (даже если в системе есть процессоры, занятые потоками с меньшим приоритетом).

Современные реализации Windows в рамках единого дерева кодов могут быть использованы для различных классов задач - от рабочих станций, обслуживающих преимущественно интерфейс пользователя, до серверных установок на многопроцессорных машинах. Чтобы можно было эффективно использовать одну ОС в столь разных классах систем, планировщик Windows динамически изменяет длительность квантов и приоритеты, назначаемые потокам. Администратор системы может в некоторой степени изменить поведение системы при назначении длительности квантов и приоритетов потоков.

Управление квантованием

Квантование потоков осуществляется по тикам системного таймера, продолжительность одного тика составляет обычно 10 или 15 мс, больший по продолжительности тик назначают многопроцессорным машинам. Каждый тик системного таймера соответствует 3 условным единицам; величина кванта может варьироваться от 2 до 12 тиков (от 6 до 36 единиц).

Параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\ Win32PrioritySeparation предназначен для управления квантованием. На рис. 3.6 дан формат этого параметра для Windows 2000-2003.

Рисунок 3.6 – Управление квантованием в Windows

Управление длительностью кванта связано с активностью процесса, которая определяется наличием интерфейса пользователя (GUI или консоль) и его активностью. Если процесс находится в фоновом режиме, то длительность назначенного ему кванта соответствует "нулевым" колонкам таблицы 3.2 (выделены серым цветом; т.е. длительности 6 или 12 - для переменной длины кванта или 18 и 36 - для фиксированной). Когда процесс становится активным, то ему назначается продолжительность квантов, исходя из значения двух младших бит параметра Win32PrioritySeparation в соответствии с приведенной таблицей.

Таблица 3.2 – Длительность квантов в ОС Windows

	Короткий квант			Длинный квант		
Значение младших 2-х бит параметра Win32PrioritySeparation	0	1	2	0	1	2
Переменная длительность	6	12	18	12	24	36
Фиксированная длительность	18	18	18	36	36	36

Еще один случай увеличения длительности кванта - процесс долгое время не получал процессорного времени (это может случиться, если все время есть активные процессы более высокого приоритета). В этой ситуации система раз в 3-4 секунды (в зависимости от продолжительности тика) назначает процессу повышенный приоритет и квант удвоенной длительности. По истечении этого кванта приоритет возвращается к прежнему значению и восстанавливается рекомендуемая длительность кванта.

Управление приоритетами

В Windows выделяется 32 уровня приоритетов. 0 соответствует самому низкому приоритету (с таким приоритетом работает только специальный поток обнуления страниц), 31 - самому высокому. Этот диапазон делится на три части:

Приоритет 0 - соответствует приоритету потока обнуления страниц.

Приоритеты с 1 по 15 - соответствуют динамическим уровням приоритетов. Большинство потоков работают именно в этом диапазоне приоритетов, и Windows может корректировать в некоторых случаях приоритеты потоков из этого диапазона.

Приоритеты с 16 по 31 - соответствуют приоритетам "реального времени". Этот уровень достаточно высок для того, чтобы поток, работающий с таким приоритетом, мог реально помешать нормальной работе других потоков в системе - например, помешать обрабатывать сообщения от клавиатуры и мыши. Windows самостоятельно не корректирует приоритеты этого диапазона.

Для некоторого упрощения управления приоритетами в Windows выделяют "классы приоритета" (priority class), которые задают базовый уровень приоритета, и "относительные приоритеты" потоков, которые корректируют указанный базовый уровень. Операционная система предоставляет набор функций для управления классами и относительными приоритетами потоков.

Планировщик операционной системы также может корректировать уровень приоритета (из диапазона 1-15), однако базовый уровень (т.е. класс) не может быть изменен. Такая коррекция приоритета выполняется в случае:

- Завершения операции ввода-вывода - в зависимости от устройства, приоритет повышается на 1 - 8 уровней.

- По окончании ожидания события или семафора - на один уровень.
- При пробуждении GUI потоков - на 2 уровня.
- По окончании ожидания потоком активного процесса (определяется по активности интерфейса) - на величину, указанную младшими 2 битами параметра Win32PrioritySeparation (см. управление длительностью кванта).

В случае коррекции приоритета по одной из перечисленных причин, повышенный приоритет начинает постепенно снижаться до начального уровня потока - с каждым тиком таймера на один уровень.

Еще один случай повышения приоритета (вместе с увеличением длительности кванта) - процесс долгое время не получал процессорного времени. В этой ситуации система раз в 3-4 секунды назначает процессу приоритет, равный 15, и квант удвоенной длительности. По истечении этого кванта приоритет возвращается к прежнему значению и восстанавливается рекомендуемая длительность кванта.

3.7 СИСТЕМНЫЕ ФУНКЦИИ РАБОТЫ С МНОГОЗАДАЧНОСТЬЮ В MICROSOFT WINDOWS

Рассматривая огромное количество функций по управлению многозадачностью, предоставляемых ОС Windows программисту, мы разделим их на условные четыре группы: управление процессами, управление потоками, синхронизация и взаимодействие.

Функции управления процессами

Функция «CreateProcess»

```
BOOL CreateProcess(
    PCTSTR pszApplicationName,
    PTSTR pszCommandLine,
    PSECURITY_ATTRIBUTES psaProcess,
    PSECURITY_ATTRIBUTES psaThiead,
    BOOL bInheritHandles,
    DWORD fdwCreate,
    PVOID pvEnvironment,
    PCTSTR pszCurDir,
    PSTARTUPINFO psiStartInfo, PPROCESS_INFORMATION ppiProcInfo);
```

Когда приложение вызывает CreateProcess, система создает новый объект ядра "процесс" с начальным значением счетчика числа его пользователей, равным 1. Этот объект – не сам процесс, а компактная структура данных, через которую операционная система управляет процессом. (Объект ядра "процесс" следует рассматривать как структуру данных со статистической информацией о процессе.) Затем система создает для нового процесса виртуальное адресное пространство и

загружает в него код и данные, как для исполняемого файла, так и для любых DLL (если таковые требуются).

Далее система формирует объект ядра "поток" (со счетчиком, равным 1) для первичного потоки нового процесса. Как и в первом случае, объект ядра "поток" – это компактная структура данных, через которую система управляет потоком. Первичный поток начинает с исполнения стартового кода из библиотеки C/C++, который в конечном счете вызывает функцию WinMain, wWinMain, main или wmain в Вашей про грамме. Если системе удастся создать новый процесс и его первичный поток, CreateProcess вернет TRUE.

Функция «ExitProcess»

```
VOID ExilProcess(UINT fuExitCode);
```

Эта функция завершает процесс и заносит в параметр fuExitCode код завершения процесса. Возвращаемого значения у ExitProcess нет, так как результат ее действия — завершение процесса. Если за вызовом этой функции в программе присутствует какой-нибудь код, он никогда не исполняется.

Когда входная функция (WinMain, wWinMain, main или wmairi) в Вашей программе возвращает управление, оно передастся стартовому коду из библиотеки C/C++, и тот проводит очистку всех ресурсов, выделенных им процессу, а затем обращается к функции ExitProcess, передавая ей значение, возвращенное входной функцией Вот почему воз врат управления входной функцией первичного потока приводит к завершению все го процесса. Обратите внимание, что при завершении процесса прекращается выполнение и всех других его потоков.

Функция «TerminateProcess»

```
BOOL TerminateProcess( HANDLE hProcess, UINT fuExitCode);
```

Главное отличие этой функции от ExitProcess в том, что ее может вызвать любой поток и завершить любой процесс. Параметр bProcess идентифицирует описатель завершаемого процесса, а в параметре fuExitCode возвращается код завершения процесса.

Пользуйтесь TerminateProcess лишь в том случае, когда иным способом завершить процесс не удается. Процесс не получает абсолютно никаких уведомлений о том, что он завершается, и приложение не может ни выполнить очистку, ни предотвратить свое неожиданное завершение (если оно, конечно, не использует механизмы защиты). При этом теряются все данные, которые процесс не успел переписать из памяти на диск.

Функции управления потоками

Функция «CreateThread»

```
HANDLE CreateThread(
```

```
    PSECURITY_ATTRIBUTES psa,
    DWORD cbStack,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD tdwCreate,
    PDWORD pdwThreadID);
```

Мы уже говорили, как при вызове функции CreateProcess появляется на свет первичный поток процесса. Если Вы хотите создать дополнительные потоки, нужно вызывать из первичного потока функцию CreateThread.

При каждом вызове этой функции система создает объект ядра "поток» Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке. Так что объект ядра "поток" — полный аналог объекта ядра "процесс".

Система выделяет память под стек потока из адресного пространства процесса. Новый поток выполняется в контексте того же процесса, что и родительский поток. Поэтому он получает доступ ко всем описателям объектов ядра, всей памяти и стекам всех потоков в процессе. За счет этого потоки в рамках одного процесса могут легко взаимодействовать друг с другом.

Функция «ExitThread»

```
VOID ExitThread(DWORD dwExitCode);
```

При вызове этой функции освобождаются все ресурсы операционной системы, выделенные дан ному потоку, но C/C++ - ресурсы (например, объекты, созданные из C++-классов) не очищаются. Именно поэтому лучше возвращать управление из функции потока, чем самому вызывать функцию ExitThread.

В параметр dwExitCode Вы помещаете значение, которое система рассматривает как код завершения потока. Возвращаемого значения у этой функции нет, потому что после ее вызова поток перестает существовать.

Функция «TerminateThread»

```
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

В отличие от ExitThread, которая уничтожает только вызывающий поток, эта функция завершает поток, указанный в параметре hThread. В параметр dwExitCode Вы помещаете значение, которое система рассматривает как код завершения потока. После того как поток будет уничтожен, счетчик пользователей его объекта ядра "по ток» уменьшится на 1.

Корректно написанное приложение не должно вызывать эту функцию, поскольку поток не получает никакого уведомления о завершении; из-за этого он не может вы полнить должную очистку ресурсов.

Функции «SuspendThread» и «ResumeThread»

```
DWORD ResumeThread(HANDLE hThread);
```

Если вызов ResumeThread прошел успешно, она возвращает предыдущее значение счетчика простоев данного потока; в ином случае – 0xFFFFFFFF.

Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен 3 раза, то и возобновлен он должен быть тоже 3 раза – лишь тогда система выделит ему процессорное время. Выполнение потока можно приостановить не только при его создании с флагом CREATE_SUSPENDED, но и вызовом SuspendThread.

```
DWORD SuspendThread(HANDLE hThread);
```

Любой поток может вызвать эту функцию и приостановить выполнение другого потока (конечно, если его описатель известен). Хоть об этом нигде и не говорится, приостановить свое выполнение поток способен сам, а возобновить себя без посторонней помощи – нет. Как и ResumeThread, функция SuspendThread возвращает предыдущее значение счетчика простоев данного потока. Поток можно приостанавливать не более чем MAXIMUM_SUSPEND_COUNT раз (в файле WinNT.h это значение определено как 127). Обратите внимание, что SuspendThread в режиме ядра работает асинхронно, но в пользовательском режиме не выполняется, пока поток остается в приостановленном состоянии.

Функции синхронизации процессов

Функция «WaitForSingleObject»

```
DWORD WaitForSingleObject(
    HANDLE hObject,
    DWORD dwMilliseconds);
```

Когда поток вызывает эту функцию, первый параметр, hObject, идентифицирует объект ядра, поддерживающий состояния «свободен-занят» (То есть любой объект, упомянутый в списке из предыдущего раздела.) Второй параметр, dwMilliseconds, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта.

Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем hProcess.

В случае если процесс готов ждать указанного объекта бесконечно долгое время, во втором параметре функции необходимо указать константу INFINITE. Именно эта константа обычно и передается функцию неопытными программистами, но Вы можете указать любое значение в миллисекундах.

> Использовать константу ожидания INFINITE следует с большой осторожностью, так как если ожидаемое событие никогда не произойдет (по любой причине) то поток, вызвавший функций WaitForSingleObject, окажется навсегда заблокированным в состоянии «Ожидание». То есть фактически в тупике.

Функция «WaitForMultipleObjects»

```
DWORD WaitForMultipleObjects(
    DWOHD dwCount,
    CONST HANDLE* phObjects,
    BOOL fWaitAll,
    DWORD dwMilliseconds);
```

Функция WaitForMultipleObjects аналогична WaitForSingleObject с тем исключением, что позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка.

Параметр dwCount определяет количество интересующих Вас объектов ядра Его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS (в заголовочных файлах Windows оно определено как 64). Параметр phObject — это указатель на массив описателей объектов ядра.

WaitForMultipleObjects приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр fWaitAll как раз и определяет, чего именно Вы хотите от функции. Если он равен TRUE, функция не даст потоку возобновить свою работу, пока нс освободятся все объекты.

Параметр dwMilliseconds идентичен одноименному параметру функции WaitFor SingleObject. Если Вы указываете конкретное время ожидания, то по его истечении функция в любом случае возвращает управление. И опять же, в этом параметре обычно передают INFINITE (будьте внимательны при написании кода, чтобы не создать ситуацию взаимной блокировки).

Возвращаемое значение функции WaitForMultipleObjects сообщает, почему возобновилось выполнение вызвавшего ее потока Значения WAIT_FAILED и WAIT_TIMEOUT никаких пояснений не требуют. Если Вы передали TRUE в параметре fWaitAll и все объекты перешли в свободное состояние, функция возвращает значение WAIT_OB JECT_0. Если fWaitAll приравнен FALSE, она возвращает управление, как только освобождается любой из объектов. Вы, по-видимому, захотите выяснить, какой именно объект освободился В этом случае возвращается значение от WAIT_OBJECT_0 до WAIT_OBJECT_0 + dwCount - 1. Иначе говоря, если возвращаемое значение не равно WAIT_TIMEOUT или WAIT_FAILED, вычтите из него значение WAlT_OBJECT_0, и Вы получите индекс в массиве описателей, на который указывает второй параметр функции WaitForMultipleObjects. Индекс подскажет Вам, какой объект перешел в незанятое состояние.

Функция «CreateEvent»

```
HANDLE CreateEvent(
    PSECURITY_ATTRIBUTES psa
     BOOL fManualReset
     BOOL fInitialState,
    PCTSTR pszName);
```

События – самая примитивная разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая — его состояние (свободен или занят).

События просто уведомляют об окончании какой-либо операции. Объекты-события бывают двух типов: со сбросом вручную (manual-reset events) и с автосбросом (auto-reset events). Первые позволяют возобновлять выполнение сразу нескольких ждущих потоков, вторые — только одного.

Объекты-события обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что тот может продолжить работу. Инициализирующий поток переводит объект "событие» в занятое состояние и приступает к своим операциям. Закончив, он сбрасывает событие в свободное состояние. Тогда другой поток, который ждал перехода события в свободное состояние, пробуждается и вновь становится планируемым.

Параметр fManualReset (булева переменная) сообщает системе, хотите Вы создать событие со сбросом вручную (TRUE) или с автосбросом (FALSE). Параметру fInitialState определяет начальное состояние события — свободное (TRUE) или занятое (FALSE). После того как система создает объект событие, CreateEvent возвращает описатель события, специфичный для конкретного процесса. Потоки из других процессов могут получить доступ к этому объекту:

1) вызовом CreateEvent с тем же параметром pszName;

2) наследованием описателя;

3) применением функции DuplicateHandle;

4) вызовом OpenEvent с передачей в параметре pszName имени события, совпадающего с указанным в аналогичном параметре функции CreateEvent.

Чтобы перевести событие его в свободное состояние можно воспользоваться следующим кодом:

```
    BOOL SetEventT(HANDLE hEvenet);
```

А чтобы поменять его на занятое:

```
    BOOL ResetEvent(HANDLE hEvent);
```

Для событий с автосбросом действует следующее правило. Когда его ожидание потоком успешно завершается, этот объект автоматически сбрасывается в занятое

состояние. Отсюда и произошло название таких объектов-событий. Для этого объекта обычно не требуется вызывать ResetEvent, поскольку система сама восстанавливает его состояние, а для событий со сбросом вручную никаких побочных эффектов успешного ожидания не предусмотрено.

Функция «CreateMutex»

```
HANDLE CreateMutex(
    PSECURITY_ATTRIBUTES psa,
    BOOL fIniLialOwner,
    PCTSTR pszName);
```

Для использования объекта-мьютекса один из процессов должен сначала создать его вызовом CreateMutex.

Параметр fInitialOwner определяет начальное состояние мьютекса. Если в нем передается FALSE (что обычно и бывает), объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока и счетчик рекурсии равны 0. Если же в нем передается TRUE, идентификатор потока, принадлежащий мьютексу, приравнивается идентификатору вызывающего потока, а счетчик рекурсии получает значение 1. Поскольку теперь идентификатор потока отличен от 0, мьютекс изначально находится в занятом состоянии.

Поток получает доступ к разделяемому ресурсу, вызывая одну из Wait-функций и передавая ей описатель мьютекса, который охраняет этот ресурс. Wait-функция проверяет у мьютекса идентификатор потока, если его значение не равно 0, мьютекс свободен, в ином случае оно принимает значение идентификатора вызывающего потока, и этот поток остается планируемым.

Если Wait-функция определяет, что у мьютекса идентификатор потока не равен 0 (мьютекс занят), вызывающий поток переходит в состояние ожидания. Система запоминает это и, когда идентификатор обнуляется, записывает в него идентификатор ждущего потока, а счетчику рекурсии присваивает значение 1, после чего ждущий поток вновь становится планируемым. Все проверки и изменения состояния объекта-мьютекса выполняются на уровне атомарного доступа.

Функция «ReleaseMutex»

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Эта функция уменьшает счетчик рекурсии в объекте-мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать ReleaseMutex столько раз, сколько необходимо для обнуления счетчика рекурсии Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнулится, и объект-мьютекс освободится. После этого система проверит, ожидают ли освобождения мьютекса какие-нибудь другие потоки. Если да, система

«по-честному» выберет один из ждущих потоков и передаст ему во владение объект-мьютекс.

4 ФАЙЛОВЫЕ СИСТЕМЫ

ДЕ8: НАЗНАЧЕНИЕ, ФУНКЦИИ И УСТРОЙСТВО ФАЙЛОВЫХ СИСТЕМ

4.1 ОСНОВНЫЕ СВЕДЕНИЯ О ФАЙЛОВЫХ СИСТЕМАХ

Файловые системы хранятся на дисках. Большинство дисков делятся на несколько разделов с независимой файловой системой на каждом разделе. Сектор 0 диска называется главной загрузочной записью (MBR, Master Boot Record) и используется для загрузки компьютера. В конце главной загрузочной записи содержится таблица разделов. В этой таблице хранятся начальные и конечные адреса (номера блоков) каждого раздела. Один из разделов помечен в таблице как активный. При загрузке компьютера BIOS считывает и исполняет MBR-запись, после чего загрузчик в MBR-записи определяет активный раздел диска, считывает его первый блок, называемый загрузочным, и исполняет его. Программа, находящаяся в загрузочном блоке, загружает операционную систему, содержащуюся в этом разделе.

Для единообразия каждый дисковый раздел начинается с загрузочного блока, даже если в нем не содержится загружаемой операционной системы. К тому же в этом разделе может быть в дальнейшем установлена операционная система, поэтому зарезервированный загрузочный блок оказывается полезным.

Во всем остальном строение раздела диска меняется от системы к системе. Часто файловые системы содержат некоторые из элементов, показанных на рис. 4.1. Один из таких элементов, называемый суперблоком, содержит ключевые параметры файловой системы и считывается в память при загрузке компьютера или при первом обращении к файловой системе. Типичная информация, хранящаяся в суперблоке, включает «магическое» число, позволяющее различать системные файлы, количество блоков в файловой системе, а также другую ключевую административную информацию.

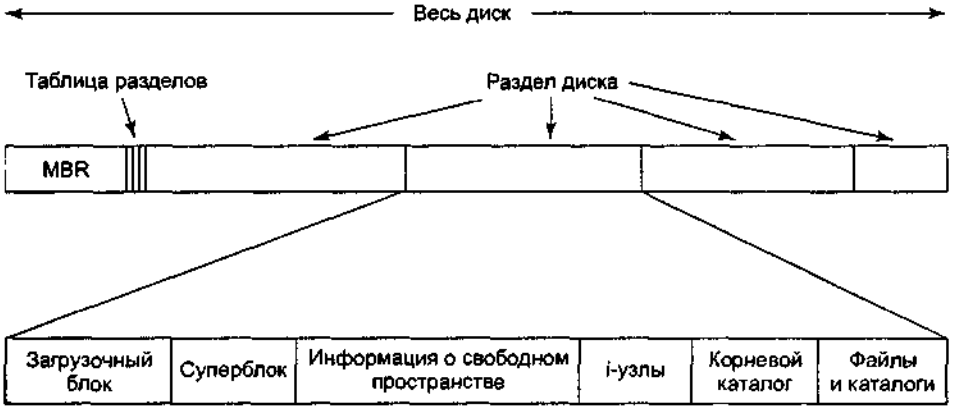


Рисунок 4.1 – Возможная структура файловой системы

Следом располагается информация о свободных блоках файловой системы, например в виде битового массива или списка указателей. За этими данными может следовать информация об i-узлах, представляющих собой массив структур данных, по одной структуре на файл, содержащих всю информацию о файлах. Следом может размещаться корневой каталог, содержащий вершину дерева файловой системы. Наконец, остальное место дискового раздела занимают все остальные каталоги и файлы.

Реализация файлов

Подавляющее большинство современных носителей информации являются блочными устройствами и позволяют читать или писать данные только в блоках кратных определенному минимальному размеру. При такой организации файлы с большой вероятностью будут занимать более одного блока данных и важным становится способ, которым файловая система (ФС) будет сохранять информацию и списке занимаемых файлом блоков.

Рассмотрим некоторые способы расположения файлов в дискретном блочном пространстве.

Непрерывные файлы

Простейшей схемой выделения файлам определенных блоков на диске является система, в которой файлы представляют собой непрерывные наборы соседних блоков диска. Тогда на диске, состоящем из блоков по 1 Кбайт, файл размером в 50 Кбайт будет занимать 50 последовательных блоков. При 2-килобайтных блоках такой файл займет 25 соседних блоков.

Пример непрерывных файлов показан на рис. 4.2, а. Здесь показаны первые 40 блоков диска, начиная с блока 0, слева. Вначале диск был пуст. Затем на диск, начиная с блока 0, был записан файл А длиной в четыре блока. После него был записан шестиблочный файл В, впритык к файлу А. Обратите внимание, что каждый файл начинается с нового блока, так что если длина файла Л была равна 3/4 блока, некоторое место в конце последнего блока файла пропадает. На рисунке всего показано семь файлов. Каждый следующий файл начинается с блока, следующего за последним блоком предыдущего файла. Затенение используется только для того, чтобы было легче различать отдельные файлы.

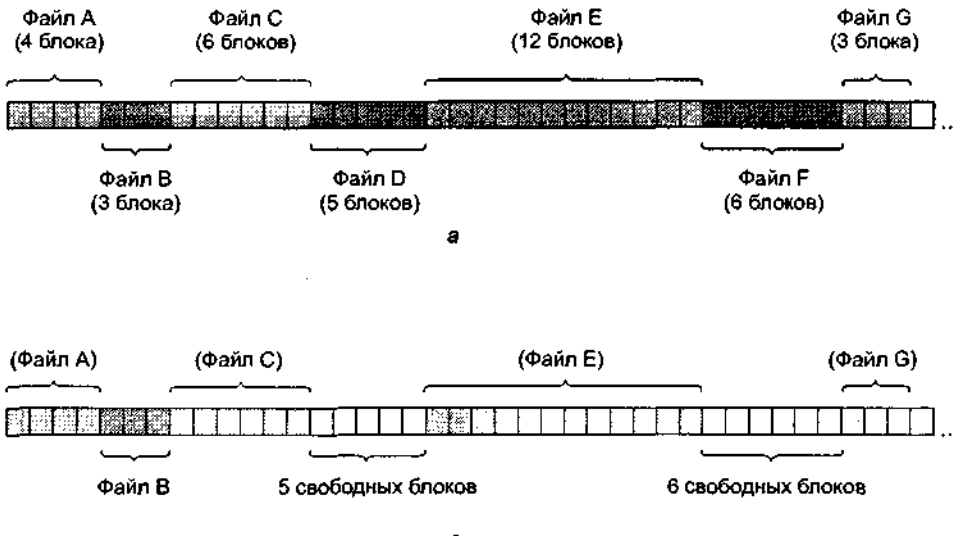


Рисунок 4.2 – Семь непрерывных файлов на диске (а); состояние диска после удаления двух файлов (б)

У непрерывных файлов есть два существенных преимущества. Во-первых, такую систему легко реализовать, так как системе, чтобы определить, какие блоки принадлежат тому или иному файлу, нужно следить всего лишь за двумя числами: номером первого блока файла и числом блоков в файле. Зная первый блок файла, любой другой его блок легко получить при помощи простой операции сложения.

Во-вторых, при работе с непрерывными файлами производительность просто превосходна, так как весь файл может быть прочитан с диска за одну операцию. Требуется только одна операция поиска (для первого блока). После этого более не нужно искать цилиндры и тратить время на ожидания вращения диска, поэтому данные могут считываться с максимальной скоростью, на которую способен диск. Таким образом, непрерывные файлы легко реализуются и обладают высокой производительностью.

К сожалению, у такого способа распределения дискового пространства имеется серьезный недостаток: со временем диск становится фрагментированным. Чтобы понять, как это происходит, рассмотрим рис. 4.2, б. Два файла, D и F, были удалены. Когда файл удаляется, его блоки освобождаются, оставляя промежутки свободных блоков на диске. По мере удаления файлов диск становится все более «дырявым».

Вначале эта фрагментация не представляет проблемы, так как каждый новый файл может быть записан в конец диска, вслед за предыдущим файлом. Однако, в конце концов, диск заполнится и либо потребуется специальная операция по уплотнению используемого пространства диска, либо надо будет изыскать способ использовать свободное пространство на месте удаленных файлов. Для повторного использования освободившегося пространства потребуется содержать список пустых участков, что в принципе выполнимо. Однако при создании нового файла будет необходимо знать его окончательный размер, чтобы выбрать для него участок подходящего размера.

Представьте себе последствия такой структуры. Пользователь запускает текстовый редактор или текстовой процессор, чтобы создать документ. Первое, что интересует программу, это сколько байтов будет в документе. На этот вопрос следует дать ответ, в противном случае программа не сможет работать. Если пользователь укажет слишком маленькое число, программа может закончиться аварийно, так как свободный участок диска окажется заполнен и будет негде разместить остальную часть файла. Если пользователь попытается обойти эту проблему, задав заведомо большой окончательный размер, например 100 Мбайт, может случиться, что редактор не сможет найти такой большой свободный участок и сообщит, что не может создать файл. Конечно, пользователь может поторговаться и снизить свои требования до 50 Мбайт и т. д. до тех пор, пока не найдется подходящий свободный участок. Тем не менее, такая схема вряд ли доставит удовольствие пользователям.

И все-таки есть ситуации, в которых непрерывные файлы могут применяться и в самом деле широко используются: на компакт-дисках. Здесь все размеры файлов известны заранее и не могут меняться при последующем использовании файловой системы CD-ROM. Наиболее распространенную файловую систему CD-ROM мы рассмотрим ниже в этой главе.

В ИТ индустрии история часто повторяется с появлением новых технологий. Файловые системы, состоящие из непрерывных файлов, применялись на магнитных дисках много лет назад благодаря их простоте и высокой производительности (удобство для пользователей почти не принималось тогда в расчет). Затем эта идея была позабыта из-за необходимости задавать окончательный размер файла при его создании. Но с появлением CD-ROM и DVD, а также других одноразовых оптических носителей о преимуществах непрерывных файлов вспомнили снова. Изучение старых систем и идей оказывается полезным, так как многие простые и ясные концепции тех систем находят применение в новых системах самым удивительным образом.

Связные списки

Второй метод размещения файлов состоит в представлении каждого файла в виде связного списка из блоков диска, как показано на рис. 4.3. Первое слово каждого блока используется как указатель на следующий блок. В остальной части блока хранятся данные.

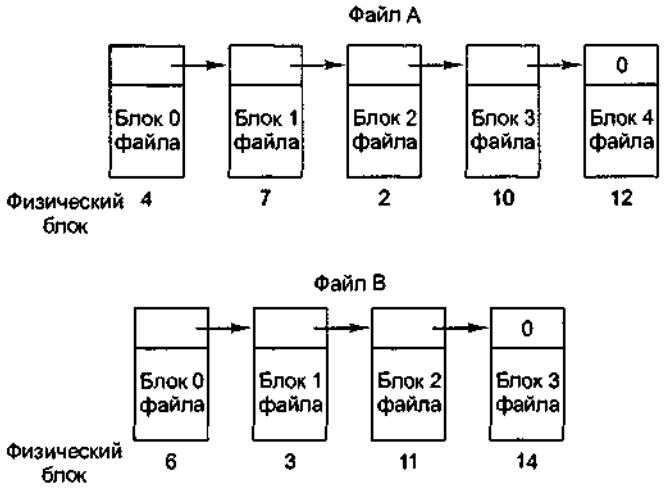


Рисунок 4.3 – Размещение файла в виде связного списка блоков диска

В отличие от систем с непрерывными файлами, такой метод позволяет использовать каждый блок диска. Нет потерь дискового пространства на фрагментацию (кроме потерь в последних блоках файла). Кроме того, в каталоге нужно хранить только адрес первого блока файла. Всю остальную информацию можно найти там.

С другой стороны, хотя последовательный доступ к такому файлу несложен, произвольный доступ будет довольно медленным. Чтобы получить доступ к блоку N, операционная система должна сначала прочитать первые N - 1 блоков по очереди. Очевидно, такая схема оказывается очень медленной.

Кроме того, размер блока уменьшается на несколько байтов, требуемых для хранения указателя. Хотя это и не смертельно, но размер блока, не являющийся степенью двух, будет менее эффективным, так как многие программы читают и пишут блоками по 512, 1024, 2048 и т. д. байтов. Если первые несколько байтов каждого блока будут заняты указателем на следующий блок, то для чтения блока полного размера придется считывать и объединять два соседних блока диска, для чего потребуется выполнение дополнительных операций.

Связный список при помощи таблицы в памяти

Оба недостатка предыдущей схемы организации файлов в виде связных списков могут быть устранены, если указатели на следующие блоки хранить не прямо в блоках, а в отдельной таблице, загружаемой в память. На рис. 4.4 показан внешний вид такой таблицы для файлов с рис. 4.3. На обоих рисунках показаны два файла. Файл А использует блоки диска 4, 7,2,10 и 12, а файл В использует блоки диска б, 3,11 и 14. С помощью таблицы, показанной на рис. 4.4, мы можем начать с блока 4 и следовать по цепочке до конца файла. То же может быть сделано для второго файла, если начать с блока 6. Обе цепочки завершаются специальным маркером (например -1), не являющимся допустимым номером блока. Такая таблица, загружаемая в оперативную память, называется FAT-таблицей (File Allocation Table — таблица размещения файлов).

Рисунок 4.4 – Таблица размещения файлов

Эта схема позволяет использовать для данных весь блок. Кроме того, случайный доступ при этом становится намного проще. Хотя для получения доступа к какому-либо блоку файла все равно понадобится проследовать по цепочке по всем ссылкам вплоть до ссылки на требуемый блок, однако в данном случае вся цепочка ссылок уже хранится в памяти, поэтому для следования по ней не требуются дополнительные дисковые операции. Как и в предыдущем случае, в каталоге достаточно хранить одно целое число (номер начального блока файла) для обеспечения доступа ко всему файлу.

Основной недостаток этого метода состоит в том, что вся таблица должна постоянно находиться в памяти. Для 20-гигабайтного диска с блоками размером 1 Кбайт потребовалась бы таблица из 20 млн. записей, по одной для каждого из 20 млн. блоков диска. Каждая запись должна состоять как минимум из трех байтов. Для ускорения поиска размер записей должен быть увеличен до 4 байт. Таким образом, таблица будет постоянно занимать 60 или 80 Мбайт оперативной памяти. Таблица, конечно, может быть размещена в виртуальной памяти, но и в этом случае ее размер оказывается чрезмерно большим, к тому же, постоянная выгрузка таблицы на диск и загрузка с диска существенно снизит производительность файловых операций.

I-узлы

Последний метод отслеживания принадлежности блоков диска файлам состоит в связывании с каждым файлом структуры данных, называемой i-узлом (index node — индекс-узел), содержащей атрибуты файла и адреса блоков файла. Простой пример i-узла показан на рис. 4.5. При наличии i-узла можно найти все блоки файла. Большое преимущество такой схемы перед хранящейся в памяти таблицей из связных списков заключается в том, что каждый конкретный i-узел должен находиться в памяти только тогда, когда соответствующий ему файл открыт. Если

каждый i-узел занимает n байт, а одновременно открыто может быть k файлов, то для массива i-узлов потребуется в памяти всего kn байтов.

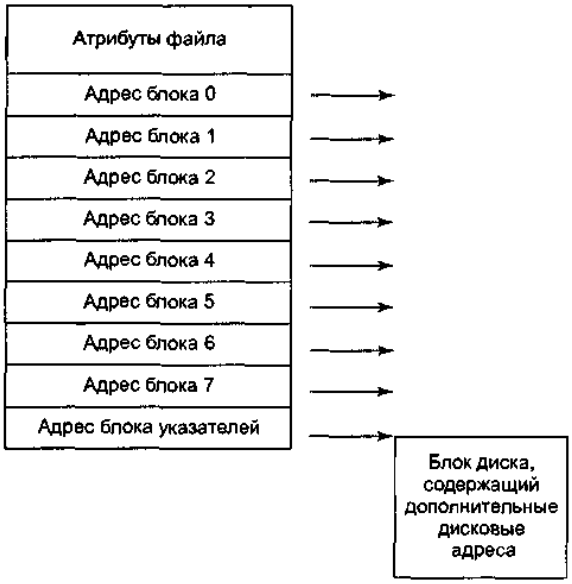


Рисунок 4.5 – Пример i-узла

Обычно этот размер значительно меньше, чем FAT-таблица, описанная в предыдущем разделе. Это легко объясняется. Размер таблицы, хранящей связный список всех блоков диска, пропорционален размеру самого диска. Для диска из n блоков потребуется n записей в таблице. Таким образом, размер таблицы линейно растет с ростом размера диска. Для схемы i-узлов, напротив, требуется массив в памяти с размером, пропорциональным максимальному количеству файлов, которые могут быть открыты одновременно. При этом не важно, будет ли размер диска 1 Гбайт, 10 Гбайт или 100 Гбайт.

С такой схемой связана проблема, заключающаяся в том, что при выделении каждому файлу фиксированного количества дисковых адресов этого количества может не хватить. Одно из решений заключается в резервировании последнего дискового адреса не для блока данных, а для следующего адресного блока. Более того, можно создавать целые цепочки и даже деревья адресных блоков. Мы снова вернемся к теме i-узлов, когда приступим к изучению системы UNIX позднее.

Реализация каталогов

Прежде чем прочитать файл, его следует открыть. При открытии файла операционная система использует поставляемое пользователем имя пути, чтобы найти запись в каталоге. Запись в каталоге содержит информацию, необходимую для нахождения блоков диска. В зависимости от системы это может быть дисковый адрес всего файла (для непрерывных файлов), номер первого блока файла (обе схемы связных списков) или номер i-узла. Во всех случаях основная функция системы каталогов состоит в преобразовании ASCII-имени в информацию, необходимую для нахождения данных.

С этой проблемой тесно связан вопрос размещения атрибутов файла. Каждая файловая система поддерживает различные атрибуты файла, такие как дату создания файла, имя владельца файла и т. д., и всю эту информацию нужно где-то хранить. Один из возможных вариантов состоит в хранении этих сведений прямо в записи каталога. Многие файловые системы именно так и поступают. Этот вариант показан на рис. 4.6, а. В этой простой схеме каталог состоит из списка элементов фиксированной длины по одному на файл, содержащих имена файлов, структуру атрибутов файла, а также один или несколько дисковых адресов, указывающих расположение файла на диске.

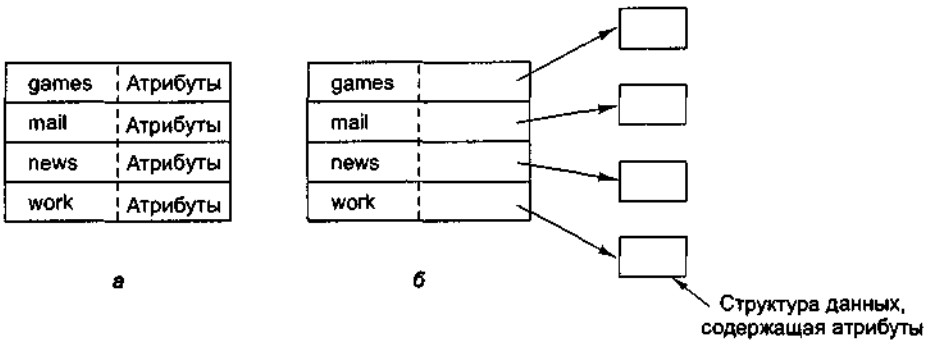


Рисунок 4.6 – Простой каталог, содержащий записи фиксированной длины с атрибутами и дисковыми адресами (а); каталог, в котором каждая запись является просто ссылкой на i-узел (б)

Системы, использующие i-узлы, могут хранить атрибуты в i-узлах, а не в записях каталога. В этом случае запись в каталоге может быть короче: просто имя файла и номер i-узла. Этот подход показан на рис. 4.6, б. Как мы увидим позднее, у этого метода есть определенные преимущества по сравнению с помещением атрибутов прямо в записи каталога. Два подхода, показанные на рис. 4.6, соответствуют системам MS-DOS и UNIX, о чем еще будет рассказано в этой главе.

До сих пор мы предполагали, что файлы имеют короткие имена фиксированной длины. В системе MS-DOS файл может иметь имя длиной от 1 до 8 символов, а также расширение длиной до 3 символов. В системе UNIX Version 7 имена файлов могут быть от одного до 14 символов, включая расширение. Однако почти всеми современными операционными системами поддерживаются более длинные имена файлов переменной длины. Как это может быть реализовано?

Простейший метод состоит в установке ограничения на длину имени файла, обычно 255 символов, и использовании одной из схем, показанных на рис. 4.6. Такой способ прост, но он расходует много места в каталоге, так как длинные имена обычно бывают далеко не у всех файлов. Следовательно, для более эффективного использования дискового пространства желательно использовать другую структуру.

Один из альтернативных подходов состоит в отказе от предположения о том, что все записи в каталоге должны иметь один и тот же размер. При таком подходе каждая запись в каталоге начинается с порции фиксированного размера, обычно

начинающейся с длины записи, за которой следуют данные в фиксированном формате — идентификатор владельца, дата создания, информация о защите и прочие атрибуты. Следом за заголовком фиксированной длины идет часть записи переменной длины, содержащая имя файла (рис. 4.7, а). На рисунке показаны три описателя файла, project-budget, personnel. Имя каждого файла завершается специальным символом (обычно 0), обозначенным на рисунке перечеркнутыми квадратиками. Чтобы каждая запись в каталоге могла начинаться с границы слова, имя каждого файла дополняется до целого числа слов байтами, показанными на рисунке затененными прямоугольниками.

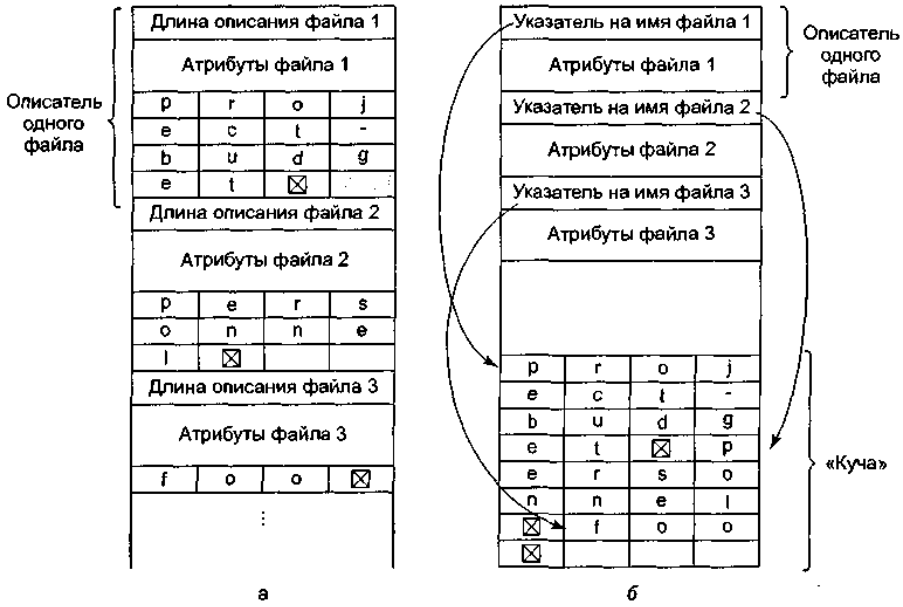


Рисунок 4.7 – Два варианта реализации длинных имен: прямо в записи каталога (а); в «куче» (б)

Недостаток этого метода состоит в том, что при удалении файла в каталоге остается промежуток переменной длины, в который описатель следующего файла может не поместиться. Эта проблема аналогична проблеме хранения на диске непрерывных файлов, хотя уплотнить каталог значительно легче, чем весь диск. Другая проблема связана с тем, что каталоговые записи переменной длины могут занимать сразу две страницы памяти. При чтении такой каталоговой записи может возникнуть прерывание из-за отсутствия в оперативной памяти следующей страницы.

Другой метод реализации длинных имен файлов заключается в том, чтобы сделать все записи каталога фиксированной (равной) длины и хранить в них только указатели на имена, а сами имена хранить отдельно в «куче», в конце каталога, как показано на рис. 4.7, б. Преимущество этого метода состоит в том, что при удалении файла освободившееся место в каталоге точно подойдет для нового описателя файла. Тем не менее «кучу» придется все так же «разгребать», и при чтении длинного имени из нее также может возникнуть прерывание из-за отсутствия страницы памяти. Однако имена файлов уже не должны начинаться с границы слов, поэтому символы-заполнители не потребуются.

Во всех рассмотренных нами пока схемах при поиске файла каталоги просматриваются линейно сверху вниз. Для очень больших каталогов, содержащих много тысяч файлов, такой поиск может занять довольно много времени. Один из способов ускорить поиск файла состоит в использовании хэш-таблицы в каждом каталоге. Пусть размер такой таблицы будет равен n. При добавлении в каталог нового файла его имя должно хэшироваться в число от 0 до n. В качестве хэш-функции может использоваться, например, взятие остатка от деления имени файла на n. В качестве альтернативы можно делить не само имя, а сумму слов, его образующих. Возможны и другие варианты.

В любом случае исследуется элемент таблицы, соответствующий полученному хэш-коду. Если элемент не используется, туда помещается указатель на описатель файла. (Описатели файлов размещаются вслед за хэш-таблицей.) Если же элемент таблицы уже занят, то создается связный список, объединяющий все описатели файлов с одинаковым хэш-кодом.

Поиск файла осуществляется аналогично. Имя файла хэшируется. По хэш-коду определяется элемент таблицы. Затем проверяются все описатели файла из связного списка и сравниваются с искомым именем файла обычным способом. Если имени файла в связном списке нет, это означает, что файла нет в каталоге.

Преимуществом использования хэш-таблицы является ускоренный в несколько раз поиск файла. Недостаток этого метода состоит в более сложном администрировании каталога. Применять этот метод стоит только в тех системах, в которых ожидается, что каталоги будут содержать сотни и тысячи файлов.

Принципиально отличный способ ускорения процесса поиска файлов в больших каталогах заключается в кэшировании результатов поиска. Прежде чем начать поиск файла, проверяется, нет ли его имени в кэше. Если файловая система недавно уже искала этот файл, его имя окажется в кэше и повторная операция поиска будет выполнена очень быстро. Конечно, кэширование поможет только в том случае, если файловая система много раз обращается к небольшому количеству файлов.

4.2 НЕКОТОРЫЕ ВОПРОСЫ ОПТИМИЗАЦИИ ФАЙЛОВЫХ СИСТЕМ

Доступ к диску производится значительно медленнее, чем к оперативной памяти. Чтение слова из памяти может занять около 10 не. Чтение с жесткого диска может выполняться со скоростью 10 Мбайт/с, что в сорок раз медленнее, но к этому следует добавить 5-10 мс на поиск нужного цилиндра и задержку вращения диска. Если требуется прочитать или записать всего одно слово, то оперативная память оказывается примерно в миллион раз быстрее жесткого диска. Поэтому во многих файловых системах применяются различные методы оптимизации, увеличивающие производительность. В данном разделе мы рассмотрим три из них.

Кэширование

Для минимизации количества обращений к диску применяется блочный кэш или буферный кэш. (Термин «кэш» происходит от французского слова cacher, что значит «скрывать»), В данном контексте кэшем называется набор блоков, логически принадлежащих диску, но хранящихся в оперативной памяти по соображениям производительности.

Существуют различные алгоритмы управления кэшем. Обычная практика заключается в перехвате всех запросов чтения к диску и проверке наличия требующихся блоков в кэше. Если блок присутствует в кэше, то запрос чтения блока может быть удовлетворен без обращения к диску. В противном случае блок сначала считывается с диска в кэш, а оттуда копируется по нужному адресу памяти. Последующие обращения к тому же блоку могут удовлетворяться из кэша.

Работа кэша показана на рис. 4.8. Поскольку в кэше хранится большое количество (часто тысячи) блоков, требуется некий быстрый способ определения наличия или отсутствия блока в кэше. Обычно для этого используется хэширование номера устройства и дискового адреса (номера блока) и поиск результата в хэш-таблице. Все блоки с одинаковыми хэш-кодами сцепляются вместе в связный список.

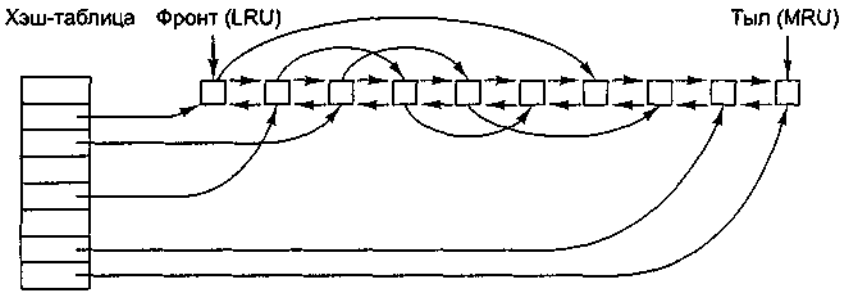


Рисунок 4.8 – Структура данных блочного кэша

Когда требуется загрузить блок в заполненный до предела кэш, какой-либо другой блок должен быть удален из кэша (и записан на диск, если он был модифицирован в кэше). Эта ситуация очень похожа на страничную организацию памяти, и к ней применимы все обычные алгоритмы замены, описанные в главе 3, такие как FIFO (First in First Out — первым прибыл — первым обслужен), «вторая попытка» и LRU (Least Recently Used — с наиболее давним использованием). Одно приятное отличие кэширования от страничной организации памяти состоит в том, что обращения к кэшу производятся относительно нечасто, что позволяет хранить все блоки в точном LRU-порядке со связными списками.

На рис. 4.8 мы видим, что в дополнение к цепям, начинающимся в хэш-таблице, используется также и двунаправленный список, в котором содержатся номера всех блоков в порядке их использования. При этом самый старый блок помещается в начало списка, а самый новый блок — в его конец. При обращении к блоку блок может перемещаться со своей текущей позиции в конец двунаправленного списка. Таким образом, может поддерживаться точный LRU-порядок.

К сожалению, здесь есть одна загвоздка. Теперь, когда мы можем реализовать точное выполнение алгоритма LRU, оказывается, что алгоритм LRU является нежелательным. Вызвано это тем, что буквальное применение алгоритма LRU снижает надежность файловой системы и угрожает ее непротиворечивости (обсуждавшейся в предыдущем разделе). Если в кэш считывается и модифицируется критический блок, например блок i-узла, но не записывается тут же на диск, то компьютерный сбой может привести к тому, что файловая система окажется в противоречивом состоянии. Если блок i-узла поместить в конец цепочки LRU, то может пройти довольно много времени, прежде чем этот блок попадет в ее начало и будет записан на диск.

Более того, к некоторым блокам, таким как блоки i-узлов, программы редко обращаются дважды в течение короткого интервала времени. Исходя из этих соображений, мы приходим к модифицированной схеме LRU, принимая во внимание два следующих фактора:

1. Насколько велика вероятность того, что данный блок скоро снова понадобится?

2. Важен ли данный блок для непротиворечивости файловой системы?

Для ответа на каждый из этих вопросов блоки можно разделить на такие категории, как блоки i-узлов, косвенные блоки, блоки каталогов, блоки, полные данных, и блоки, частично заполненные данными. Блоки, которые, вероятно, не потребуются снова в ближайшее время, помещаются в начало списка LRU, чтобы занимаемые ими буферы могли вскоре освободиться. Блоки, вероятность повторного использования которых в ближайшее время высока (например, записываемые блоки, частично заполненные данными), помещаются в конец списка LRU, что позволяет им оставаться в кэше более долгое время.

Второй вопрос не связан с первым. Если блок представляет важность для непротиворечивости файловой системы (обычно это все блоки, кроме блоков данных) и такой блок модифицируется, то его следует немедленно сохранить на диске независимо от его положения в списке LRU. Быстро записывая критические блоки, мы значительно снижаем вероятность того, что сбой компьютера повредит файловую систему. Пользователь вряд ли будет рад потере одного из своих файлов из-за сбоя компьютера. Еще более он огорчится, если при этом испорченной окажется вся файловая система.

Даже при принятии всех перечисленных выше мер предосторожности по поддержанию в рабочем состоянии файловой системы слишком долгое хранение в кэше блоков с данными является нежелательным. Представьте себе пользователя, работающего на персональном компьютере над написанием книги. Даже если наш писатель периодически велит текстовому редактору сохранять редактируемый файл на диске, есть большая вероятность, что все блоки останутся в кэше. Если произойдет сбой, структура файловой системы не пострадает, но работа целого дня работы будет потеряна.

Эта ситуация случается не слишком часто, если только с очень невезучими пользователями. Для решения данной проблемы обычно применяется два метода. В системе UNIX есть системный вызов sync, принуждающий сохранение всех модифицированных блоков кэша на диске. При загрузке операционной системы запускается фоновая задача, обычно называющаяся update, вся работа которой заключается в периодическом (обычно через каждые 30 с) обращении к системному вызову sync. В результате при любом сбое будет потеряно не более 30 с работы.

В системе MS-DOS используется другой подход, состоящий в том, что каждый модифицированный блок записывается на диск сразу же. Кэш, в котором все модифицированные блоки немедленно записываются на диск, называются сквозным кэшем или кэшем со сквозной записью. При использовании сквозного кэша количество обращений ввода-вывода к диску больше, чем при применении обычного кэша. Чтобы лучше понять разницу в этих двух подходах, представьте себе программу, записывающую блок размером в 1 Кбайт по одному символу. Система UNIX будет собирать все символы в кэше и записывать этот блок на диск каждые 30 с или когда блок будет удален из кэша. Система MS-DOS будет обращаться к диску при каждом записываемом символе. Конечно, большинством программ применяется внутренняя буферизация, поэтому обычно они обращаются к системному вызову write не с одним символом, а с целыми строками или большими единицами данных.

Результатом различия стратегий кэширования оказывается тот факт, что простое удаление (гибкого) диска из системы UNIX, без выполнения системного вызова sync, почти всегда приведет к потере данных и часто также к повреждению файловой системы. В MS-DOS такой проблемы не возникает. Такое различие в стратегиях связано с тем, что система UNIX разрабатывалась в среде, в которой все диски были жесткими и постоянными, тогда как система MS-DOS изначально предназначалась для работы с гибкими дисками. Когда жесткие диски стали нормой, более эффективный метод, применяющийся в UNIX, стал нормой и теперь также используется в Windows для жестких дисков.

Опережающее чтение блока

Второй метод увеличения производительности файловой системы состоит в попытке получить блоки диска в кэш прежде, чем они потребуются. В частности, многие файлы считываются последовательно. Когда файловая система получает запрос на чтение блока k файла, она выполняет его, но после этого сразу проверяет, есть ли в кэше блок k + 1. Если этого блока в кэше нет, файловая система читает его в надежде, что к тому моменту, когда он понадобится, этот блок уже будет считан в кэш. В крайнем случае, он уже будет на пути туда.

Конечно, такая стратегия работает только для тех файлов, которые считываются последовательно. Если обращения к блокам файла производятся в случайном

порядке, опережающее чтение не помогает. В самом деле, не хотелось бы обременять диск считыванием ненужных блоков и удалением потенциально полезных блоков из кэша (возможно, тем самым еще более обременяя диск необходимостью записывать эти блоки на диск, если они модифицированы). Чтобы определить, следует ли использовать опережающее чтение блоков, файловая система может вести учет доступа к блокам каждого открытого файла. Например, для каждого открытого файла один бит может означать «режим последовательного доступа» или «режим произвольного доступа». Вначале каждому открываемому файлу в соответствии с принципом презумпции невиновности назначается режим последовательного доступа. Однако при перемещении указателя в файле этот бит сбрасывается. Если к этому файлу опять будут обращаться с запросами последовательного чтения, бит будет установлен снова. Таким образом, файловая система может строить догадки о том, следует ли ей выполнять операции опережающего чтения или нет. Если она и будет ошибаться время от времени, то ничего страшного не произойдет, просто будет потрачен впустую некоторый процент пропускной способности диска.

Снижение времени перемещения блока головок

Кэширование и опережающее чтение являются не единственными способами увеличения производительности системы. Другой важный метод состоит в уменьшении затрат времени на перемещение блока головок. Достигается это помещением блоков, к которым высока вероятность доступа в течение короткого интервала времени, близко друг к другу, желательно на одном цилиндре. Когда записывается выходной файл, файловая система должна зарезервировать место для чтения таких блоков за одну операцию. Если свободные блоки учитываются в битовом массиве, а весь битовый массив помещается в оперативной памяти, то довольно легко выбрать свободный блок как можно ближе к предыдущему блоку. В случае, когда свободные блоки хранятся в списке, часть которого в оперативной памяти, а часть на диске, сделать это значительно труднее.

Однако даже при использовании списка свободных блоков может быть выполнена определенная кластеризация блоков. Хитрость заключается в том, чтобы учитывать место на диске не в блоках, а в группах последовательных блоков. Если сектор состоит из 512 байт, система может использовать блоки размером в 1 Кбайт (2 сектора), но выделять пространство на диске в единицах по 2 блока (4 сектора). Это не то же самое, что использование 2-килобайтных дисковых блоков, так как кэш по-прежнему будет использовать килобайтные блоки и дисковые операции чтения и записи будут по-прежнему работать с килобайтными блоками. Однако при последовательном чтении файла количество операций поиска цилиндра уменьшится вдвое, что значительно увеличит производительность. Вариация этой же темы состоит в попытке системы учесть позицию блока в цилиндре.

Еще один фактор, снижающий производительность файловых систем, связан с тем, что при использовании i-узлов или чего-либо эквивалентного им, особенно при чтении коротких файлов, требуется два обращения к диску вместо одного: одно для i-узла и одно для блока данных. Обычное размещение i-узлов на диске показано на рис. 4.9, а. Здесь все i-узлы располагаются в начале диска, так что среднее расстояние между i-узлом и его блоками будет составлять около половины количества цилиндров, то есть при доступе практически к каждому файлу потребуются значительные перемещения блока головок.

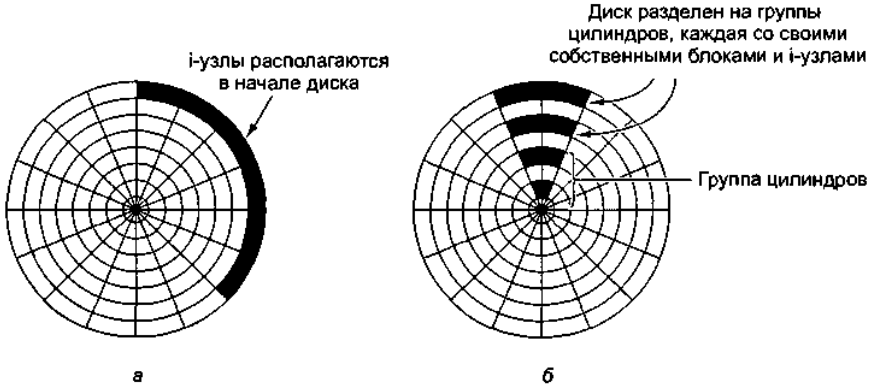


Рисунок 4.9 – i-узлы, размещенные в начале диска (а); диск, разделенный на группы цилиндров, каждая со своими собственными блоками и i-узлами (б)

Один из способов увеличения производительности состоит в помещении i-узлов в середину диска, уменьшая, таким образом, среднее расстояние перемещения блоков головок в два раза. Другая идея, показанная на рис. 4.9, б, заключается в разбиении диска на группы цилиндров, каждая со своими i-узлами, блоками и списком свободных блоков. Когда создается новый файл, может быть выбран любой i-узел, но предпринимается попытка найти блок в той же группе цилиндров, что и i-узел. Если эта попытка заканчивается неудачей, используется блок в соседней группе цилиндров.

ДЕ9: КОНКРЕТНЫЕ ПРИМЕРЫ РЕАЛИЗАЦИИ ФАЙЛОВЫХ СИСТЕМ

4.3 ФАЙЛОВЫЕ СИСТЕМЫ FAT

Наиболее распространенная файловая система. Модификации этой файловой системы (VFAT,FAT32) до сих пор служат основной файловой системой в Windows 9x, а для дискет даже Windows NT не предлагает ничего другого. Оригинальная версия FAT, сохранившаяся практически неизменной от MS-DOS 2.0 до MS-DOS 6.22 крайне проста: вся информация о файле хранится в каталоге, и для доступа к нему используется имя файла, построенное по так называемой “формуле 8.3”.

Большие и маленькие буквы в именах файлов не различаются: при всех операциях с файлами используются большие буквы. У каждого файла хранится

время модификации с точностью до 2 секунд, а также хранится информация о четырёх видах атрибутов – Read Only, Archive, Hidden, System.

Кэширование дисковых операций в FAT реализовано очень эффективно. Это связано с тем, что FAT16 имеет очень мало данных, отвечающих за организацию файловой системы. Из служебных областей можно выделить только саму область FAT, которая не может превышать 128 Кбайт (!) - эта область отвечает и за поиск фрагментов файлов, и за поиск свободного места на томе. Каталоги системы FAT также очень компактны. Общий объем памяти, необходимый для предельно эффективной работы с FAT-ом, может колебаться от сотни килобайт и до мегабайта-другого - при условии огромного числа и размера каталогов, с которыми ведется работа.

На быстродействие FAT очень сильно влияет размер физической кэш-памяти, установленной на жёстком диске.

VFAT

Файловая система VFAT впервые появилась в Windows NT, а широкое распространение получила после выхода Windows 95. Это усовершенствованная версия FAT, в которой разрешены длинные имена файлов.

Появившиеся в VFAT длинные имена сделали работу с файлами более удобной, однако породили ряд проблем. Во-первых, VFAT сохраняет в именах разницу между большими и маленькими буквами, но для доступа к файлам разрешает использовать любые их комбинации. Во-вторых, что более существенно, у каждого файла в VFAT есть два имени – длинное и короткое. При попытке создать новый файл, имя которого совпадает с коротким именем существующего файла, будет выдано сообщение о том, что такой файл уже есть.

Если в VFAT дать файлу имя, удовлетворяющее ограничениям FAT и состоящее из символов стандартного набора ASCII, то Windows 95 и Windows NT будут считать, что он имеет только короткое имя. При этом с точки зрения Windows 95 такое имя будет состоять только из больших букв, а сточки зрения Windows NT – только из маленьких.

FAT32

FAT32, введённая в Windows 95 OSR2 и поддерживаемая в Windows 98 отличается от VFAT лишь количественными параметрами: она допускает меньший размер кластеров и больший размер дисков.

В связи с появление этих возможностей на первый план выходит один недостаток, связанный с потерей быстродействия в случае высокой фрагментации. FAT32, из-за большой области самой таблицы размещения будет испытывать огромные трудности, если фрагменты файла разбросаны по всему диску. Дело в

том, что FAT (File Allocation Table, таблица размещения файлов) представляет собой мини-образ диска, куда включен каждый его кластер. Для доступа к фрагменту файла в системе FAT16 и FAT32 приходится обращаться к соответствующей частичке FAT. Если файл, к примеру, расположен в трех фрагментах - в начале диска, в середине, и в конце - то в системе FAT нам придется обратиться к фрагменту FAT также в его начале, в середине и в конце. В системе FAT16, где максимальный размер области FAT составляет 128 Кбайт, это не составит проблемы - вся область FAT просто хранится в памяти, или же считывается с диска целиком за один проход и буферизируется. FAT32 же, напротив, имеет типичный размер области FAT порядка сотен килобайт, а на больших дисках - даже несколько мегабайт. Если файл расположен в разных частях диска - это вынуждает систему совершать движения головок винчестера столько раз, сколько групп фрагментов в разных областях имеет файл, а это очень и очень сильно замедляет процесс поиска фрагментов файла.

Важным параметром быстродействия является время создания файла. Особенно этот показатель важен при записи аудио/видеоинформации в реальном времени. Для определения того, свободен ли данный кластер или нет, системы на основе FAT должны просмотреть одну запись FAT, соответствующую этому кластеру. Размер одной записи FAT16 составляет 16 бит, одной записи FAT32 - 32 бита. Для поиска свободного места на диске может потребоваться просмотреть почти всего FAT - это 128 Кбайт (максимум) для FAT16 и до нескольких мегабайт (!) - в FAT32. Для того, чтобы не превращать поиск свободного места в катастрофу (для FAT32), операционной системе приходится идти на различные ухищрения.

FAT16 и FAT32 имеют очень компактные каталоги, размер каждой записи которых предельно мал. Более того, из-за сложившейся исторически системы хранения длинных имен файлов (более 11 символов), в каталогах систем FAT используется не очень эффективная и на первый взгляд неудачная, но зато очень экономная структура хранения длинных имен файлов. Работа с каталогами FAT производится достаточно быстро, так как в подавляющем числе случаев каталог (файл данных каталога) не фрагментирован и находится на диске в одном месте.

Единственная проблема, которая может существенно понизить скорость работы каталогов FAT - большое количество файлов в одном каталоге (порядка тысячи или более). Система хранения данных - линейный массив - не позволяет организовать эффективный поиск файлов в таком каталоге, и для нахождения данного файла приходится перебирать большой объем данных (в среднем - половину файла каталога).

С кэшированием дисковых операций в FAT32 дела обстоят значительно хуже, чем с FAT16. Это связанно с тем, что сама область FAT может иметь более внушительные размеры. На томах порядка 5 - 10 Гбайт область FAT может занимать объем в несколько Мбайт, и это уже очень внушительный объем, надежно кэшировать который не представляется возможным. Тем не менее, область FAT, а

вернее те фрагменты, которые отвечают за местоположение рабочих файлов, в подавляющем большинстве систем находятся в памяти машины - на это расходуется порядка нескольких Мбайт оперативной памяти.

Быстродействие системы FAT32 можно довольно существенно повысить, увеличив размер кластера. Если в NTFS размер кластера почти не влияет на размер и характер данных системных областей, то в системе FAT увеличивая кластер в два раза, мы сокращаем область FAT в те же два раза. Сокращение области FAT в несколько раз даст заметное увеличение быстродействия, так как объем системных данных файловой системы сильно сократиться - уменьшается и время, затрачиваемое на чтение данных о расположении файлов, и объем оперативной памяти, необходимый для буферизирования этой информации. Типичный объем кластера для систем FAT32 составляет тоже 4 Кбайт, и увеличение его до 8 или даже до 16 Кбайт - особенно для больших (десяток и более гигабайт) дисков - достаточно разумный шаг.

4.4 ФАЙЛОВАЯ СИСТЕМА EXT2

Производители жестких дисков обычно поставляют свои изделия отформатированными на низком уровне. Это означает, что все дисковое пространство с помощью специальных меток разбито на "сектора", размером 512 байт. Такой диск (или дисковый раздел) должен быть подготовлен для использования в определенной операционной системе. В MS-DOS или Windows процедура подготовки называется форматированием, а в Linux — созданием файловой системы. Создание файловой системы ext2fs заключается в создании в разделе диска определенной логической структуры. Эта структура строится следующим образом.

Во-первых, на диске выделяется загрузочная область. Загрузочная область создается в любой файловой системе. На первичном разделе она содержит загрузочную запись — фрагмент кода, который инициирует процесс загрузки операционной системы при запуске. На других разделах эта область не используется. Все остальное пространство на диске делится на блоки. Блок может иметь размер от 1, 2 или 4 килобайта. Блок является адресуемой единицей дискового пространства. Выделение места файлам осуществляется целыми блоками, поэтому при выборе размера блока приходится идти на компромисс. Большой размер блока, как правило, сокращает число обращений к диску при чтении или записи файла, но зато увеличивает долю нерационально используемого пространства, особенно при наличии большого числа файлов маленького размера.

Блоки, в свою очередь, объединяются в группы блоков (рис. 4.10). Группы блоков в файловой системе и блоки внутри группы нумеруются последовательно, начиная с 1. Первый блок на диске имеет номер 1 и принадлежит группе с номером 1. Общее число блоков на диске (в разделе диска) является делителем объема диска, выраженного в секторах. А число групп блоков не обязано делить число блоков,

потому что последняя группа блоков может быть не полной. Начало каждой группы блоков имеет адрес, который может быть получен как ((номер_группы — 1)* (число_блоков_в_группе)).

Загрузочная запись	Группа блоков 1	Группа Блоков 2	...	Группа блоков n

Рисунок 4.10 – Структура дискового раздела в ext2fs

Каждая группа блоков имеет одинаковое строение, изображенное на рис. 4.11.

Супер-блок	Описание группы блоков (Group Descriptors)	Битовая карта блоков (Block Bitmap)	Битовая карта индексных дескрипторов (Inode Bitmap)	Таблица индексных дескрипторов(Inode Table)	Область блоков данных

Рисунок 4.11 – Внутренняя структура группы блоков

Такая структура служит повышению производительности файловой системы за счет того, что сокращается расстояние между таблицей индексных дескрипторов и блоками данных, а, следовательно, сокращается время поиска нужного места головками в процессе операций записи/считывания файла.

Первый элемент каждой группы блоков (суперблок) одинаков для всех групп, а все остальные — индивидуальны для каждой группы. Суперблок хранится в первом блоке каждой группы блоков. Суперблок является начальной точкой файловой системы. Он имеет размер 1024 байта и всегда располагается по смещению 1024 байта от начала файловой системы. Наличие нескольких копий суперблока объясняется чрезвычайной важностью этого элемента файловой системы. Дубликаты суперблока используются при восстановлении файловой системы после сбоев.

Информация, хранимая в суперблоке, используется для организации доступа к остальным данным на диске. В суперблоке определяется размер файловой системы, максимальное число файлов в разделе, объем свободного пространства и содержится информация о том, где искать незанятые участки. При запуске ОС суперблок считывается в память, и все изменения файловой системы вначале находят отображение в копии суперблока, находящейся в ОП, и записываются на диск только периодически. Это позволяет повысить производительность системы,

так как многие пользователи и процессы постоянно обновляют файлы. С другой стороны, при выключении системы суперблок обязательно должен быть записан на диск, что не позволяет выключать компьютер простым выключением питания. В противном случае, при следующей загрузке информация, записанная в суперблоке, окажется не соответствующей реальному состоянию файловой системы.

Структура сперблока приведена в таблице 4.1.

Таблица 4.1 – Структура суперблока

Название поля	Тип	Комментарий
s_inodes_count	ULONG	Число индексных дескрипторов в файловой системе
s_blocks_count	ULONG	Число блоков в файловой системе
s_r_blocks_count	ULONG	Число блоков, зарезервированных для суперпользователя
s_free_blocks_count	ULONG	Счетчик числа свободных блоков
s_free_inodes_count	ULONG	Счетчик числа свободных индексных дескрипторов
s_first_data_block	ULONG	Первый блок, который содержит данные. В зависимости от размера блока, это поле может быть равно 0 или 1.
s_log_block_size	ULONG	Индикатор размера логического блока: 0 = 1 Кб; 1 = 2 Кб; 2 = 4 Кб.
s_log_frag_size	LONG	Индикатор размера фрагментов (кажется, понятие фрагмента в настоящее время не используется)
s_blocks_per_group	ULONG	Число блоков в каждой группе блоков
s_frags_per_group	ULONG	Число фрагментов в каждой группе блоков
s_inodes_per_group	ULONG	Число индексных дескрипторов (inodes) в каждой группе блоков
s_mtime	ULONG	Время, когда в последний раз была смонтирована файловая система.

Название поля	Тип	Комментарий
s_wtime	ULONG	Время, когда в последний раз производилась запись в файловую систему
s_mnt_count	USHORT	Счетчик числа монтирований файловой системы. Если этот счетчик достигает значения, указанного в следующем поле (s_max_mnt_count), файловая система должна быть проверена (это делается при перезапуске), а счетчик обнуляется.
s_max_mnt_count	SHORT	Число, определяющее, сколько раз может быть смонтирована файловая система
s_magic	USHORT	"Магическое число" (0xEF53), указывающее, что файловая система принадлежит к типу ex2fs
s_state	USHORT	Флаги, указывающее текущее состояние файловой системы (является ли она чистой (clean) и т.п.)
s_errors	USHORT	Флаги, задающие процедуры обработки сообщений об ошибках (что делать, если найдены ошибки).
s_pad	USHORT	Заполнение
s_lastcheck	ULONG	Время последней проверки файловой системы
s_checkinterval	ULONG	Максимальный период времени между проверками файловой системы
s_creator_os	ULONG	Указание на тип ОС, в которой создана файловая система
s_rev_level	ULONG	Версия (revision level) файловой системы.
s_reserved	ULONG[235]	Заполнение до 1024 байт

Вслед за суперблоком расположено описание группы блоков (Group Descriptors). Это описание представляет собой массив, имеющий структуру, приведенную в таблице 4.2.

Таблица 4.2 – Структура описания группы блоков

Название поля	Тип	Назначение
bg_block_bitmap	ULONG	Адрес блока, содержащего битовую карту блоков (block bitmap) данной группы
bg_inode_bitmap	ULONG	Адрес блока, содержащего битовую карту индексных дескрипторов (inode bitmap) данной группы
bg_inode_table	ULONG	Адрес блока, содержащего таблицу индексных дескрипторов (inode table) данной группы
bg_free_blocks_count	USHORT	Счетчик числа свободных блоков в данной группе
bg_free_inodes_count	USHORT	Число свободных индексных дескрипторов в данной группе
bg_used_dirs_count	USHORT	Число индексных дескрипторов в данной группе, которые являются каталогами
bg_pad	USHORT	Заполнение
bg_reserved	ULONG[3]	Заполнение

Размер описания группы блоков можно вычислить как

*(размер_группы_блоков_в_ext2 * число_групп) / размер_блока,*

при необходимости округляя вверх.

Информация, которая хранится в описании группы, используется для того, чтобы найти битовые карты блоков и индексных дескрипторов, а также таблицу индексных дескрипторов. Не забывайте, что блоки и группы блоков нумеруются, начиная с 1.

Битовая карта блоков (block bitmap) — это структура, каждый бит которой показывает, отведен ли соответствующий ему блок какому-либо файлу. Если бит равен 1, то блок занят. Эта карта служит для поиска свободных блоков в тех случаях, когда надо выделить место под файл, Битовая карта блоков занимает число блоков, равное (число_блоков_в_группе / 8) / размер_блока (при необходимости округляем).

Битовая карта индексных дескрипторов выполняет аналогичную функцию по отношению к таблице индексных дескрипторов: показывает, какие именно дескрипторы заняты.

Следующая область в структуре группы блоков служит для хранения таблицы индексных дескрипторов файлов. И, наконец, все оставшееся место в группе блоков отводится для хранения собственно файлов.

Индексные дескрипторы файлов

Каждому файлу на диске соответствует один и только один индексный дескриптор файла, который идентифицируется своим порядковым номером — индексом файла. Это означает, что число файлов, которые могут быть созданы в файловой системе, ограничено числом индексных дескрипторов, которое либо явно задается при создании файловой системы, либо вычисляется исходя из физического объема дискового раздела.

Строение индексного дескриптора файла приведено в таблице 4.3.

Таблица 4.3 – Структура индексного дескриптора

Название поля	Тип	Описание
i_mode	USHORT	Тип и права доступа к данному файлу
i_uid	USHORT	Идентификатор владельца файла (Owner Uid)
i_size	ULONG	Размер файла в байтах
i_atime	ULONG	Время последнего обращения к файлу (Access time)
i_ctime	ULONG	Время создания файла
i_mtime	ULONG	Время последней модификации файла
i_dtime	ULONG	Время удаления файла
i_gid	USHORT	Идентификатор группы (GID)
i_links_count	USHORT	Счетчик числа связей (Links count)
i_blocks	ULONG	Число блоков, занимаемых файлом
i_flags	ULONG	Флаги файла (File flags)

Название поля	Тип	Описание
i_reserved1	ULONG	Зарезервировано для ОС
i_block	ULONG[15]	Указатели на блоки, в которых записаны данные файла (это поле подробно описано в *разд. 16.4*)
i_version	ULONG	Версия файла (для NFS)
i_file_acl	ULONG	ACL файла
i_dir_acl	ULONG	ACL каталога
i_faddr	ULONG	Адрес фрагмента (Fragment address)
i_frag	UCHAR	Номер фрагмента (Fragment number)
i_fsize	UCHAR	Размер фрагмента (Fragment size)
i_pad1	USHORT	Заполнение
i_reserved2	ULONG[2]	Зарезервировано

Поле типа и прав доступа к файлу представляет собой двухбайтовое слово, каждый бит которого служит флагом, индицирующим отношение файла к определенному типу или установку одного конкретного права на файл (таблица 4.4).

Таблица 4.4 – Структура поля, задающего тип и права доступа

Идентификатор	Значение	Назначение флага (поля)
S_IFMT	F000	Маска для типа файла
S_IFSOCK	A000	Доменное гнездо (socket)
S_IFLNK	C000	Символическая ссылка
S_IFREG	8000	Обычный (regular) файл
S_IFBLK	6000	Блок-ориентированное устройство
S_IFDIR	4000	Каталог
S_IFCHR	2000	Байт-ориентированное (символьное) устройство
S_IFIFO	1000	Именованный канал (fifo)

S_ISUID	0800	SUID — бит смены владельца
S_ISGID	0400	SGID — бит смены группы
S_ISVTX	0200	Бит сохранения задачи (sticky bit)
S_IRWXU	01C0	Маска прав владельца файла
S_IRUSR	0100	Право на чтение
S_IWUSR	0080	Право на запись
S_IXUSR	0040	Право на выполнение
S_IRWXG	0038	Маска прав группы
S_IRGRP	0020	Право на чтение
S_IWGRP	0010	Право на запись
S_IXGRP	0008	Право на выполнение
S_IRWXO	0007	Маска прав остальных пользователей
S_IROTH	0004	Право на чтение
S_IWOTH	0002	Право на запись
S_IXOTH	0001	Право на выполнение

Среди индексных дескрипторов имеется несколько дескрипторов, которые зарезервированы для специальных целей и играют особую роль в файловой системе (таблица 4.5).

Таблица 4.5 – Особые индексные дескрипторы

Идентификатор	Значение	Описание
EXT2_BAD_INO	1	Индексный дескриптор, в котором перечислены адреса дефектных блоков на диске (Bad blocks inode)
EXT2_ROOT_INO	2	Индексный дескриптор корневого каталога файловой системы (Root inode)
EXT2_ACL_IDX_INO	3	ACL inode

EXT2_ACL_DATA_INO	4	ACL inode
EXT2_BOOT_LOADER_INO	5	Индексный дескриптор загрузчика (Boot loader inode)
EXT2_UNDEL_DIR_INO	6	Инлексный дескриптор каталога для удаленных файлов (Undelete directory inode)
EXT2_FIRST_INO	11	Первый незарезервированный индексный дескриптор

Самый важный дескриптор в этом списке — дескриптор корневого каталога. Этот дескриптор указывает на корневой каталог, который, подобно всем каталогам, представляет собой связанный список, состоящий из записей переменной длины. Каждая запись имеет следующую структуру (табл. 4.6):

Таблица 4.6 – Структура дескриптора, описывающего корневой каталог

Название поля	Тип	Описание
Inode	ULONG	Номер индексного дескриптора (индекс) файла
Rec_len	USHORT	Длина этой записи
Name_len	USHORT	Длина имени файла
Name	CHAR[0]	Имя файла

Использование записей переменной длины позволяет использовать длинные имена файлов без пустой траты дискового пространства. Отдельная запись в каталоге не может пересекать границу блока (т. е. должна быть расположена целиком внутри одного блока). Поэтому, если очередная запись не помещается целиком в данном блоке, она переносится в следующий блок, а предыдущая запись продолжается таким образом, чтобы она заполнила блок до конца.

Система адресации данных

Система адресации данных — это одна из самых существенных составных частей файловой системы. Именно система адресации позволяет находить нужный файл среди множества как пустых, так и занятых блоков на диске. В ext2fs система адресации реализуется полем i_block индексного дескриптора файла.

Поле i_block в индексном дескрипторе файла представляет собой массив из 15 адресов блоков. Первые 12 адресов в этом массиве (EXT2_NDIR_BLOCKS [12]) представляют собой прямые ссылки (адреса) на номера блоков, в которых хранятся

данные из файла. Следующий адрес в этом массиве (EXT2_IND_BLOCK) является косвенной ссылкой, т. е. адресом блока, в котором хранится список адресов следующих блоков с данными из этого файла. В этом блоке могут быть записаны адреса (размер_блока / размер_ULONG) блоков с данными файла.

Следующий адрес в поле i_block индексного дескриптора (EXT2_DIND_BLOCK) указывает на блок двойной косвенной адресации (double indirect block). Этот блок содержит список адресов блоков, которые в свою очередь содержат списки адресов следующих блоков данных того файла, который задается данным индексным дескриптором.

И, наконец, последний адрес (EXT2_TIND_BLOCK) в поле i_block индексного дескриптора задает адрес блока тройной косвенной адресации, т. е. блока со списком адресов блоков, которые являются блоками двойной косвенной адресации.

Теперь вы знаете, как устроены индексные дескрипторы файлов, т. е. можете представить, как в файловой системе ext2fs осуществляется запись в файл и чтение из файла.

4.5 ФАЙЛОВАЯ СИСТЕМА NTFS

Раздел NTFS, теоретически, может быть почти какого угодно размера. Максимальный размер раздела NTFS в данный момент ограничен лишь размерами жестких дисков.

Как и любая другая система, NTFS делит все полезное место на кластеры - блоки данных, используемые единовременно. NTFS поддерживает почти любые размеры кластеров - от 512 байт до 64 Кбайт, неким стандартом же считается кластер размером 4 Кбайт.

Файловая система NTFS представляет собой выдающееся достижение структуризации: каждый элемент системы представляет собой файл - даже служебная информация. Самый главный файл на NTFS называется MFT, или Master File Table - общая таблица файлов. Именно он размещается в MFT зоне и представляет собой централизованный каталог всех остальных файлов диска, и, как не парадоксально, себя самого. MFT поделен на записи фиксированного размера (обычно 1 Кбайт), и каждая запись соответствует какому либо файлу (в общем смысле этого слова). Первые 16 файлов носят служебный характер и недоступны операционной системе - они называются метафайлами, причем самый первый метафайл - сам MFT. Эти первые 16 элементов MFT - единственная часть диска, имеющая фиксированное положение. Интересно, что вторая копия первых трех записей, для надежности - они очень важны - хранится ровно посередине диска. Остальной MFT-файл может располагаться, как и любой другой файл, в произвольных местах диска - восстановить его положение можно с помощью его самого, "зацепившись" за самую основу - за первый элемент MFT.

Первые 16 файлов NTFS (метафайлы) носят служебный характер. Каждый из них отвечает за какой-либо аспект работы системы. Преимущество настолько модульного подхода заключается в поразительной гибкости - например, на FAT-е физическое повреждение в самой области FAT фатально для функционирования всего диска, а NTFS может сместить, даже фрагментировать по диску, все свои служебные области, обойдя любые неисправности поверхности - кроме первых 16 элементов MFT.

NTFS - отказоустойчивая система, которая вполне может привести себя в корректное состояние при практически любых реальных сбоях. Любая современная файловая система основана на таком понятии, как транзакция - действие, совершаемое целиком и корректно или не совершаемое вообще. У NTFS просто не бывает промежуточных (ошибочных или некорректных) состояний - квант изменения данных не может быть поделен на до и после сбоя, принося разрушения и путаницу - он либо совершен, либо отменен.

По поводу быстродействия можно сказать, что NTFS способна обеспечить быстрый поиск фрагментов файлов, поскольку вся информация хранится в нескольких очень компактных записях (типичный размер - несколько килобайт). Если файл очень сильно фрагментирован (содержит большое число фрагментов) - NTFS придется использовать много записей, что часто заставит хранить их в разных местах. Лишние движения головок при поиске этих данных, в таком случае, приведут к сильному замедлению процесса поиска данных о местоположении файла.

Создание файла на NTFS происходит гораздо быстрее, чем на FAT16/FAT32. Это связанно с тем, что NTFS имеет битовую карту свободного места, одному кластеру соответствует 1 бит. Для поиска свободного места на диске приходится оценивать объемы в десятки раз меньшие, чем в системах FAT и FAT32.

В NTFS файловые операции в каталогах содержащих большое кол-во файлов производится значительно быстрее, чем в FAT и FAT32 это связанно с тем, что NTFS использует гораздо более эффективный способ адресации - бинарное дерево. Эта организация позволяет эффективно работать с каталогами любого размера - каталогам NTFS не страшно увеличение количества файлов в одном каталоге и до десятков тысяч.

Стоит заметить, однако, что сам каталог NTFS представляет собой гораздо менее компактную структуру, нежели каталог FAT - это связано с гораздо большим (в несколько раз) размером одной записи каталога. Данное обстоятельство приводит к тому, что каталоги на томе NTFS в подавляющем числе случаев сильно фрагментированы. Размер типичного каталога на FAT-е укладывается в один кластер, тогда как сотня файлов (и даже меньше) в каталоге на NTFS уже приводит к размеру файла каталога, превышающему типичный размер одного кластера. Это, в свою очередь, почти гарантирует фрагментацию файла каталога, что, к сожалению, довольно часто сводит на нет все преимущества гораздо более эффективной организации самих данных.

Преимущества каталогов NTFS становятся реальными и неоспоримыми только в том случае, если в одно каталоге присутствуют тысячи файлов - в этом случае быстродействие компенсирует фрагментированность самого каталога и трудности с физическим обращением к данным (в первый раз - далее каталог кэшируется). Напряженная работа с каталогами, содержащими порядка тысячи и более файлов, проходит на NTFS буквально в несколько раз быстрее, а иногда выигрыш в скорости по сравнению с FAT и FAT32 достигает десятков раз.

В плане кэширования NTFS имеет большие требования к памяти, необходимой для работы системы. Прежде всего, кэширование сильно затрудняет большие размеры каталогов. Размер одних только каталогов, с которыми активно ведет работу система, может запросто доходить до нескольких Мбайт и даже десятков Мбайт! Есть также необходимость кэшировать карту свободного места тома (сотни Кбайт) и записи MFT для файлов, с которыми осуществляется работа (в типичной системе - по 1 Кбайт на каждый файл). К счастью, NTFS имеет удачную систему хранения данных, которая не приводит к увеличению каких-либо фиксированных областей при увеличении объема диска. Количество данных, с которым оперирует система на основе NTFS, практически не зависит от объема тома, и основной вклад в объемы данных, которые необходимо кэшировать, вносят каталоги. Тем не менее, уже этого вполне достаточно для того, чтобы только минимальный объем данных, необходимых для кэширования базовых областей NTFS, доходил до 5 - 8 Мбайт.

К сожалению, можно с уверенностью сказать: NTFS теряет огромное количество своего теоретического быстродействия из-за недостаточного кэширования. На системах, имеющих менее 64 Мбайт памяти, NTFS просто не может оказаться быстрее FAT16 или FAT32. Единственное исключение из этого правила - диски FAT32, имеющие объем десятки Гбайт (авторы некоторых публикаций не рекомендуют использовать диски FAT32 объемом свыше 30 Гбайт). В остальных же случаях - системы с менее чем 64 мегабайтами памяти просто обязаны работать с FAT32 быстрее.

Типичный в настоящее время объем памяти в 64 Мбайта, к сожалению, также не дает возможности организовать эффективную работу с NTFS. На малых и средних дисках (до 10 Гбайт) в типичных системах FAT32 работает, немного быстрее. Единственное, что можно сказать по поводу быстродействия систем с таким объемом оперативной памяти - системы, работающие с FAT32, будут гораздо сильнее страдать от фрагментации, чем системы на NTFS. Но если хотя бы изредка дефрагментировать диски, то FAT32, с точки зрения быстродействия, является предпочтительным вариантом. Многие люди, тем не менее, выбирают в таких системах NTFS - просто из-за того, что это даст некоторые довольно важные преимущества, тогда как типичная потеря быстродействия не очень велика.

Системы с более чем 64 Мбайтами, а особенно - со 128 Мбайт и более памяти, смогут уверенно кэшировать абсолютно всё, что необходимо для работы систем, и

вот на таких компьютерах NTFS, скорее всего, покажет более высокое быстродействие из-за более продуманной организации данных.

На быстродействие файловых систем также влияет скорость работы самого жёсткого диска. Специальный режим UDMA может дать наиболее большой выигрыш в быстродействии именно на NTFS. Это связанно с тем, что NTFS производит отложенную запись гораздо большего числа данных.

Типичный размер кластера для NTFS - 4 Кбайта. Оптимальным с точки зрения быстродействия, по крайней мере, для средних и больших файлов, считается (самой Microsoft) размер 16 Кбайт. Увеличивать размер далее неразумно из-за слишком больших расходов на неэффективность хранения данных и из-за мизерного дальнейшего увеличения быстродействия.

Структура раздела - общий взгляд

Как и любая другая система, NTFS делит все полезное место на кластеры - блоки данных, используемые единовременно. NTFS поддерживает почти любые размеры кластеров - от 512 байт до 64 Кбайт, неким стандартом же считается кластер размером 4 Кбайт. Никаких аномалий кластерной структуры NTFS не имеет, поэтому на эту, в общем-то, довольно банальную тему, сказать особо нечего.

Диск NTFS условно делится на две части. Первые 12% диска отводятся под так называемую MFT зону - пространство, в которое растет метафайл MFT (об этом ниже). Запись каких-либо данных в эту область невозможна. MFT-зона всегда держится пустой - это делается для того, чтобы самый главный, служебный файл (MFT) не фрагментировался при своем росте. Остальные 88% диска представляют собой обычное пространство для хранения файлов.

Свободное место диска, однако, включает в себя всё физически свободное место - незаполненные куски MFT-зоны туда тоже включаются. Механизм использования MFT-зоны таков: когда файлы уже нельзя записывать в обычное пространство, MFT-зона просто сокращается (в текущих версиях операционных систем ровно в два раза), освобождая таким образом место для записи файлов. При освобождении места в обычной области MFT зона может снова расширится. При этом не исключена ситуация, когда в этой зоне остались и обычные файлы: никакой аномалии тут нет. Что ж, система старалась оставить её свободной, но ничего не получилось. Жизнь продолжается... Метафайл MFT все-таки может фрагментироваться, хоть это и было бы нежелательно.

MFT и его структура

Файловая система NTFS представляет собой выдающееся достижение структуризации: каждый элемент системы представляет собой файл - даже служебная информация. Самый главный файл на NTFS называется MFT, или

Master File Table - общая таблица файлов. Именно он размещается в MFT зоне и представляет собой централизованный каталог всех остальных файлов диска, и, как не парадоксально, себя самого. MFT поделен на записи фиксированного размера (обычно 1 Кбайт), и каждая запись соответствует какому либо файлу (в общем смысле этого слова). Первые 16 файлов носят служебный характер и недоступны операционной системе - они называются метафайлами, причем самый первый метафайл - сам MFT. Эти первые 16 элементов MFT - единственная часть диска, имеющая фиксированное положение. Интересно, что вторая копия первых трех записей, для надежности (они очень важны) хранится ровно посередине диска. Остальной MFT-файл может располагаться, как и любой другой файл, в произвольных местах диска - восстановить его положение можно с помощью его самого, "зацепившись" за самую основу - за первый элемент MFT.

Метафайлы

Первые 16 файлов NTFS (метафайлы) носят служебный характер. Каждый из них отвечает за какой-либо аспект работы системы. Преимущество настолько модульного подхода заключается в поразительной гибкости - например, на FAT-е физическое повреждение в самой области FAT фатально для функционирования всего диска, а NTFS может сместить, даже фрагментировать по диску, все свои служебные области, обойдя любые неисправности поверхности - кроме первых 16 элементов MFT.

Метафайлы находятся корневом каталоге NTFS диска - они начинаются с символа имени "$", хотя получить какую-либо информацию о них стандартными средствами сложно. Любопытно, что и для этих файлов указан вполне реальный размер - можно узнать, например, сколько операционная система тратит на каталогизацию всего вашего диска, посмотрев размер файла $MFT. В таблице 4.7 приведены используемые в данный момент метафайлы и их назначение.

Таблица 4.7 – Метафайлы ФС NTFS

Имя метафайла	Описание (назначение)
$mft	Сам файла $mft (самоописание).
$mftmirr	Копия первых 16 записей MFT, размещенная посередине тома.
$logfile	Файл поддержки журналирования (см. ниже).
$volume	Служебная информация тома – метка тома, версия файловой системы, т.д.
$attrdef	Список стандартных обязательных атрибутов (в смысле ФС NTFS) файла на томе.
$.	Ссылка на корневой каталог.
$bitmap	Файл, содержащий битовую карту свободных/занятых блоков на томе.

$boot	Загрузочный раздел тома (если том загрузочный).
$quota	Файл, в котором записаны права пользователей на использование дискового пространства (начал работать лишь в NT5).
$upcase	Файл-таблица соответствия заглавных и прописных букв в имен файлов на текущем томе. Нужен в основном потому, что в NTFS имена файлов записываются в Unicode, что составляет 65 тысяч различных символов, искать большие и малые эквиваленты которых очень нетривиально.

Файлы и потоки

Итак, у системы есть файлы - и ничего кроме файлов. Что включает в себя это понятие на NTFS?

Прежде всего, обязательный элемент - запись в MFT, ведь, как было сказано ранее, все файлы диска упоминаются в MFT. В этом месте хранится вся информация о файле, за исключением собственно данных. Имя файла, размер, положение на диске отдельных фрагментов, и т.д. Если для информации не хватает одной записи MFT, то используются несколько, причем не обязательно подряд.

Опциональный элемент - потоки данных файла. Может показаться странным определение "опциональный", но, тем не менее, ничего странного тут нет. Во-первых, файл может не иметь данных - в таком случае на него не расходуется свободное место самого диска. Во-вторых, файл может иметь не очень большой размер. Тогда идет в ход довольно удачное решение: данные файла хранятся прямо в MFT, в оставшемся от основных данных месте в пределах одной записи MFT. Файлы, занимающие сотни байт, обычно не имеют своего "физического" воплощения в основной файловой области - все данные такого файла хранятся в одном месте - в MFT.

Довольно интересно обстоит дело и с данными файла. Каждый файл на NTFS, в общем-то, имеет несколько абстрактное строение - у него нет как таковых данных, а есть потоки (streams). Один из потоков и носит привычный нам смысл - данные файла. Но большинство атрибутов файла - тоже потоки! Таким образом, получается, что базовая сущность у файла только одна - номер в MFT, а всё остальное опционально. Данная абстракция может использоваться для создания довольно удобных вещей - например, файлу можно "прилепить" еще один поток, записав в него любые данные - например, информацию об авторе и содержании файла, как это сделано в Windows 2000 (самая правая закладка в свойствах файла, просматриваемых из проводника). Интересно, что эти дополнительные потоки не видны стандартными средствами: наблюдаемый размер файла - это лишь размер основного потока, который содержит традиционные данные. Можно, к примеру, иметь файл нулевой длинны, при стирании которого освободится 1 Гбайт

свободного места - просто потому, что какая-нибудь хитрая программа или технология прилепила к нему дополнительный поток (альтернативные данные) гигабайтового размера. Но на самом деле в текущий момент потоки практически не используются, так что опасаться подобных ситуаций не следует, хотя гипотетически они возможны. Просто имейте в виду, что файл на NTFS - это более глубокое и глобальное понятие, чем можно себе вообразить просто просматривая каталоги диска. Ну и напоследок: имя файла может содержать любые символы, включая полый набор национальных алфавитов, так как данные представлены в Unicode - 16-битном представлении, которое дает 65535 разных символов. Максимальная длина имени файла - 255 символов.

Каталоги

Каталог на NTFS представляет собой специфический файл, хранящий ссылки на другие файлы и каталоги, создавая иерархическое строение данных на диске. Файл каталога поделен на блоки, каждый из которых содержит имя файла, базовые атрибуты и ссылку на элемент MFT, который уже предоставляет полную информацию об элементе каталога. Внутренняя структура каталога представляет собой бинарное дерево. Вот что это означает: для поиска файла с данным именем в линейном каталоге, таком, например, как у FAT-а, операционной системе приходится просматривать все элементы каталога, пока она не найдет нужный. Бинарное же дерево располагает имена файлов таким образом, чтобы поиск файла осуществлялся более быстрым способом - с помощью получения двухзначных ответов на вопросы о положении файла. Вопрос, на который бинарное дерево способно дать ответ, таков: в какой группе, относительно данного элемента, находится искомое имя - выше или ниже? Мы начинаем с такого вопроса к среднему элементу, и каждый ответ сужает зону поиска в среднем в два раза. Файлы, скажем, просто отсортированы по алфавиту, и ответ на вопрос осуществляется очевидным способом - сравнением начальных букв. Область поиска, суженная в два раза, начинает исследоваться аналогичным образом, начиная опять же со среднего элемента.

Вывод - для поиска одного файла среди 1000, например, FAT придется осуществить в среднем 500 сравнений (наиболее вероятно, что файл будет найден на середине поиска), а системе на основе дерева - всего около 10-ти (2^10 = 1024). Экономия времени поиска налицо. Не стоит, однако думать, что в традиционных системах (FAT) всё так запущено: во-первых, поддержание списка файлов в виде бинарного дерева довольно трудоемко, а во-вторых - даже FAT в исполнении современной системы (Windows2000 или Windows98) использует сходную оптимизацию поиска. Это просто еще один факт в вашу копилку знаний. Хочется также развеять распространенное заблуждение (которое я сам разделял совсем еще недавно) о том, что добавлять файл в каталог в виде дерева труднее, чем в линейный каталог: это достаточно сравнимые по времени операции - дело в том, что для того, чтобы

добавить файл в каталог, нужно сначала убедится, что файла с таким именем там еще нет :) - и вот тут-то в линейной системе у нас будут трудности с поиском файла, описанные выше, которые с лихвой компенсируют саму простоту добавления файла в каталог.

Какую информацию можно получить, просто прочитав файл каталога? Ровно то, что выдает команда dir. Для выполнения простейшей навигации по диску не нужно лазить в MFT за каждым файлом, надо лишь читать самую общую информацию о файлах из файлов каталогов. Главный каталог диска - корневой - ничем не отличается об обычных каталогов, кроме специальной ссылки на него из начала метафайла MFT.

Журналирование

NTFS - отказоустойчивая система, которая вполне может привести себя в корректное состояние при практически любых реальных сбоях. Любая современная файловая система основана на таком понятии, как транзакция - действие, совершаемое целиком и корректно или не совершаемое вообще. У NTFS просто не бывает промежуточных (ошибочных или некорректных) состояний - квант изменения данных не может быть поделен на до и после сбоя, принося разрушения и путаницу - он либо совершен, либо отменен.

Пример 1: осуществляется запись данных на диск. Вдруг выясняется, что в то место, куда мы только что решили записать очередную порцию данных, писать не удалось - физическое повреждение поверхности. Поведение NTFS в этом случае довольно логично: транзакция записи откатывается целиком - система осознает, что запись не произведена. Место помечается как сбойное, а данные записываются в другое место - начинается новая транзакция.

Пример 2: более сложный случай - идет запись данных на диск. Вдруг, бах - отключается питание и система перезагружается. На какой фазе остановилась запись, где есть данные, а где чушь? На помощь приходит другой механизм системы - журнал транзакций. Дело в том, что система, осознав свое желание писать на диск, пометила в метафайле $LogFile это свое состояние. При перезагрузке это файл изучается на предмет наличия незавершенных транзакций, которые были прерваны аварией и результат которых непредсказуем - все эти транзакции отменяются: место, в которое осуществлялась запись, помечается снова как свободное, индексы и элементы MFT приводятся в с состояние, в котором они были до сбоя, и система в целом остается стабильна. Ну а если ошибка произошла при записи в журнал? Тоже ничего страшного: транзакция либо еще и не начиналась (идет только попытка записать намерения её произвести), либо уже закончилась - то есть идет попытка записать, что транзакция на самом деле уже выполнена. В последнем случае при следующей загрузке система сама вполне разберется, что на самом деле всё и так записано корректно, и не обратит внимания на "незаконченную" транзакцию.

И все-таки помните, что журналирование - не абсолютная панацея, а лишь средство существенно сократить число ошибок и сбоев системы. Вряд ли рядовой пользователь NTFS хоть когда-нибудь заметит ошибку системы или вынужден будет запускать chkdsk - опыт показывает, что NTFS восстанавливается в полностью корректное состояние даже при сбоях в очень загруженные дисковой активностью моменты. Вы можете даже оптимизировать диск и в самый разгар этого процесса нажать reset - вероятность потерь данных даже в этом случае будет очень низка. Важно понимать, однако, что система восстановления NTFS гарантирует корректность файловой системы, а не ваших данных. Если вы производили запись на диск и получили аварию - ваши данные могут и не записаться. Чудес не бывает.

Сжатие

Файлы NTFS имеют один довольно полезный атрибут - "сжатый". Дело в том, что NTFS имеет встроенную поддержку сжатия дисков - то, для чего раньше приходилось использовать Stacker или DoubleSpace. Любой файл или каталог в индивидуальном порядке может хранится на диске в сжатом виде - этот процесс совершенно прозрачен для приложений. Сжатие файлов имеет очень высокую скорость и только одно большое отрицательное свойство - огромная виртуальная фрагментация сжатых файлов, которая, правда, никому особо не мешает. Сжатие осуществляется блоками по 16 кластеров и использует так называемые "виртуальные кластеры" - опять же предельно гибкое решение, позволяющее добиться интересных эффектов - например, половина файла может быть сжата, а половина - нет. Это достигается благодаря тому, что хранение информации о компрессированности определенных фрагментов очень похоже на обычную фрагментацию файлов: например, типичная запись физической раскладки для реального, несжатого, файла:

кластеры файла с 1 по 43-й хранятся в кластерах диска начиная с 400-го

кластеры файла с 44 по 52-й хранятся в кластерах диска начиная с 8530-го

...

Физическая раскладка типичного сжатого файла:

кластеры файла с 1 по 9-й хранятся в кластерах диска начиная с 400-го

кластеры файла с 10 по 16-й нигде не хранятся

кластеры файла с 17 по 18-й хранятся в кластерах диска начиная с 409-го

кластеры файла с 19 по 36-й нигде не хранятся

....

Видно, что сжатый файл имеет "виртуальные" кластеры, реальной информации в которых нет. Как только система видит такие виртуальные кластеры, она тут же понимает, что данные предыдущего блока, кратного 16-ти, должны быть разжаты, а получившиеся данные как раз заполнят виртуальные кластеры - вот, по сути, и весь алгоритм.

Безопасность

NTFS содержит множество средств разграничения прав объектов - есть мнение, что это самая совершенная файловая система из всех ныне существующих. В теории это, без сомнения, так, но в текущих реализациях, к сожалению, система прав достаточно далека от идеала и представляет собой хоть и жесткий, но не всегда логичный набор характеристик. Права, назначаемые любому объекту и однозначно соблюдаемые системой, эволюционируют - крупные изменения и дополнения прав осуществлялись уже несколько раз и к Windows 2000 все-таки они пришли к достаточно разумному набору.

Права файловой системы NTFS неразрывно связаны с самой системой - то есть они, вообще говоря, необязательны к соблюдению другой системой, если ей дать физический доступ к диску. Для предотвращения физического доступа в Windows2000 (NT5) всё же ввели стандартную возможность - об этом см. ниже. Система прав в своем текущем состоянии достаточно сложна, и я сомневаюсь, что смогу сказать широкому читателю что-нибудь интересное и полезное ему в обычной жизни. Если вас интересует эта тема - вы найдете множество книг по сетевой архитектуре NT, в которых это описано более чем подробно.

На этом описание строение файловой системы можно закончить, осталось описать лишь некоторое количество просто практичных или оригинальных вещей.

Hard Links

Эта штука была в NTFS с незапамятных времен, но использовалась очень редко - и тем не менее: Hard Link - это когда один и тот же файл имеет два имени (несколько указателей файла-каталога или разных каталогов указывают на одну и ту же MFT запись). Допустим, один и тот же файл имеет имена 1.txt и 2.txt: если пользователь сотрет файл 1, останется файл 2. Если сотрет 2 - останется файл 1, то есть, оба имени, с момента создания, совершенно равноправны. Файл физически стирается лишь тогда, когда будет удалено его последнее имя.

Symbolic Links (NT5)

Гораздо более практичная возможность, позволяющая делать виртуальные каталоги - ровно так же, как и виртуальные диски командой subst в DOSe. Применения достаточно разнообразны: во-первых, упрощение системы каталогов.

Если вам не нравится каталог Documents and settings\Administrator\Documents, вы можете прилинковать его в корневой каталог - система будет по прежнему общаться с каталогом с дремучим путем, а вы - с гораздо более коротким именем, полностью ему эквивалентным. Для создания таких связей можно воспользоваться программой junction (junction.zip (15 Kb), 36 кб), которую написал известный специалист Mark Russinovich (http://www.sysinternals.com). Программа работает только в NT5 (Windows 2000), как и сама возможность. Для удаления связи можно воспользоваться стандартной командой rd.

> ВНИМАНИЕ! Попытка удаления связи с помощью проводника или других файловых менеджеров, не понимающих виртуальную природу каталога (например, FAR), приведет к удалению данных, на которые ссылается ссылка! Будьте осторожны.

Шифрование (NT5)

Полезная возможность для людей, которые беспокоятся за свои секреты - каждый файл или каталог может также быть зашифрован, что не даст возможность прочесть его другой инсталляцией NT. В сочетании со стандартным и практически непрошибаемым паролем на загрузку самой системы, эта возможность обеспечивает достаточную для большинства применений безопасность избранных вами важных данных.

4.6 ФАЙЛОВАЯ СИСТЕМА CDFS

В 1988 году был принят Международный стандарт ISO 9660, описывающий файловые системы для CD-ROM. Практически каждый продающийся сегодня диск соответствует этому стандарту, иногда согласуясь с его расширениями, которые будут обсуждаться ниже. Одно из назначений этого стандарта заключается в том, чтобы любой диск мог быть прочитан на любом компьютере, независимо от используемого байтового порядка и операционной системы. Как следствие, на файловую систему были наложены определенные ограничения, которые должны были позволить читать эти диски даже самым слабым из использовавшихся тогда операционных систем (таким как MS-DOS).

У CD-ROM нет концентрических цилиндров, как у магнитных дисков. Вместо этого они содержат непрерывную спираль, на которой последовательно размещены все биты (хотя поиск поперек спирали также возможен). Биты вдоль спирали разделены на логические блоки (также называемые логическими секторами) по 2352 байт. Некоторые из этих байтов используются для преамбул, коррекции ошибок и других накладных расходов. Полезная нагрузка в каждом блоке составляет 2048 байт. Аудиодиски содержат специальные разделительные участки между композициями, а также специальные заголовки и концевики для каждой фонограммы, не используемые в CD-ROM, содержащих другие данные. Часто

позиция блока в спирали указывается в минутах и секундах. Она может быть преобразована в линейный номер блока, так как каждая секунда содержит 75 блоков.

Стандарт ISO 9660 также поддерживает наборы размером до 216-1 CD-ROM. Сами CD-ROM также могут быть разделены на отдельные логические тома (разделы). Однако ниже будет обсуждаться стандарт ISO 9660 для одного CD-ROM, не разделенного на тома.

Каждый CD-ROM начинается с 16 блоков, чья функция не определяется стандартом ISO 9660. Производитель CD-ROM может использовать эту область для размещения загрузчика операционной системы или для другой цели. Следом располагается один блок, содержащий основной описатель тома, в котором хранится некоторая общая информация о CD-ROM. Среди данных, содержащихся в этом блоке, идентификатор системы (32 байт), идентификатор тома (32 байт) идентификатор издателя (128 байт), и идентификатор лица, подготовившего данные (128 байт). Производитель диска может заполнить эти поля произвольным образом, с условием, что он будет использовать только символы верхнего регистра, цифры и очень ограниченное количество знаков препинания, чтобы гарантировать совместимость с различными платформами.

Основной описатель тома также содержит имена трех файлов, в которых могут храниться краткий обзор, уведомление об авторских правах и библиографическая информация соответственно. Кроме того, в этом блоке также содержатся определенные ключевые числа, включающие размер логического блока (как правило, 2048, однако в определенных случаях могут использоваться блоки большего размера, например 4096, 8192 и других степеней двух), количество блоков на CD-ROM, а также дата создания и дата окончания срока службы диска. Наконец, основной описатель тома также содержит описатель корневого каталога, что позволяет найти этот каталог на CD-ROM (то есть определить номер блока, содержащего начало каталога). Начиная с этого каталога, можно определить местонахождение всей остальной файловой системы.

Корневой каталог и все остальные каталоги могут содержать переменное количество записей, в последней из которых установлен специальный бит, помечающий эту запись как последнюю. Сами каталоговые записи также могут иметь переменную длину. Каждая запись содержит от 10 до 12 полей, некоторые из них содержат текст формата ASCII, а другие являются числовыми двоичными полями. Двоичные поля кодируются дважды, один раз в формате, используемом в процессорах типа Pentium (сначала младшие байты, затем старшие), и один раз в формате, используемом в процессоре SPARC (сначала старшие байты, затем младшие). Следовательно, 16-разрядное число занимает 4 байт, а 32-разрядное число 8 байт. Такое избыточное кодирование было использовано при разработке стандарта, чтобы никого не обидеть. Если бы стандарт учитывал только один из способов хранения двоичного числа, тогда сотрудники компаний, в которых

применяется другой способ, посчитали бы, что их отнесли к гражданам второго сорта и не приняли бы стандарт. Таким образом, эмоциональное содержание CD-ROM может быть точно измерено в килобайтах потерянного пространства.

Формат каталоговой записи стандарта ISO 9660 показан на рис. 4.12. Поскольку каталоговые записи могут быть переменной длины, первое поле записи представляет собой байт, содержащий длину записи. Во избежание любых двусмысленностей стандартом определено, что старший бит этого байта располагается слева.

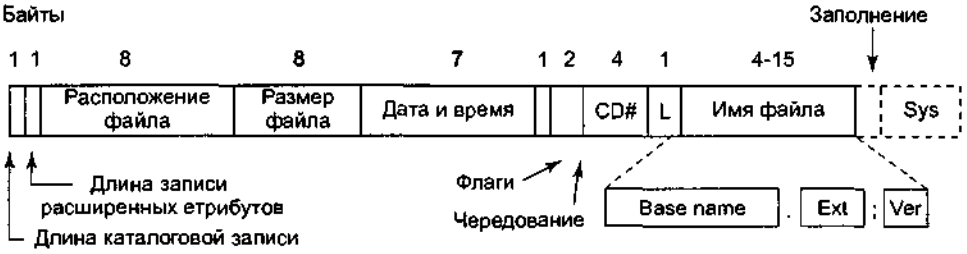


Рисунок 4.12 – Каталоговая запись стандарта ISO 9660

Записи каталогов могут иметь расширенные атрибуты. Если для каталоговой записи используется это свойство, тогда второй байт содержит длину записи расширенных атрибутов.

Следом располагается номер начального блока файла. Файлы хранятся на диске в виде непрерывных последовательностей блоков, так что размещение файла на диске однозначно определяется начальным блоком и размером, значение которого содержится в следующем поле.
В следующем поле хранятся дата и время записи CD-ROM1. Значения года, месяца, дня, часа, минуты, секунды и временной зоны хранятся в отдельных байтах. Годы отсчитываются от 1900, что означает, что CD-ROM будут страдать от проблемы 2156 года, так как следом за 2155 годом для них наступит 1900 год. Возникновение этой проблемы можно было отложить на 88 лет, приняв за точку отсчета 1988 (год принятия стандарта). Если бы это было сделано, проблему можно было бы отложить до 2244 года.

Поле Flags (флаги) содержит несколько различных управляющих битов, один из которых позволяет скрывать запись при отображении каталога (свойство, взятое из MS-DOS), другой разрешает использование расширенных атрибутов, а третий помечает последнюю запись в каталоге. Мы не станем рассматривать здесь остальные биты этого поля. Следующее поле описывает особенности чередования частей файла на диске. Это свойство не используется в простейшей версии стандарта ISO 9660, поэтому оно не будет обсуждаться в данной книге.

Еще одно поле указывает местоположение файла на CD-ROM. Стандарт допускает возможность расположения файла на другом CD-ROM набора. Таким образом, можно создать на одном CD-ROM главный каталог, содержащий все файлы всех остальных CD-ROM набора.

Поле, отмеченное на рис. 4.12 символом L, содержит длину имени файла в байтах. За ним следует само имя файла, состоящее из базового имени (base name на рисунке), точки, расширения, точки с запятой и версии файла в двоичном формате (один или два байта). В базовом имени и расширении могут использоваться прописные символы, цифры от 0 до 9 и символ подчеркивания. Все остальные символы запрещены, чтобы гарантировать, что каждый компьютер сможет работать со всеми файлами на диске. Базовое имя может быть длиной до восьми символов; расширение — до трех символов. Такой выбор был продиктован необходимостью совместимости с системой MS-DOS. Имя файла может встречаться несколько раз, но с различными номерами версий.

Последние два поля не всегда присутствуют. Поле Padding (заполнение) используется для выравнивания размера каталоговой записи до четного количества байтов, чтобы выровнять записи в каталоге по 2-байтовым границам. Если требуется выравнивание, используется нулевой байт. Наконец, функция и размер последнего поля System use (Sys на рисунке) никак не определяются стандартом. В стандарте указывается лишь, что это поле должно состоять из четного числа байтов. В различных операционных системах это поле используется различным образом. Например, Macintosh хранит в этом поле флаги Finder.

Все записи каталога, кроме первых двух, располагаются в алфавитном порядке. Первая запись представляет собой описатель самого каталога. Вторая запись является ссылкой на родительский каталог. В этом смысле эти записи аналогичны каталоговым записям «.» и «..» в UNIX.

Количество каталоговых записей не ограничено. Однако существует ограничение глубины вложенности каталогов. Максимальная глубина вложенности каталогов равна восьми.

Стандартом ISO 9660 определены так называемые три уровня. На уровне 1 применяются самые жесткие ограничения. Имена файлов ограничиваются уже описанной выше схемой 8 + 3, а имена каталогов могут состоять из восьми символов и не могут иметь расширений. Кроме того, уровень 1 требует, чтобы все файлы были непрерывными. Использование этого уровня обеспечивает совместимость CD-ROM с самым широким спектром систем.

Уровень 2 ослабляет ограничение на длину имени. Он позволяет файлам и каталогам иметь имена до 31 символа, но из того же набора символов.

На уровне 3 используются те же ограничения имен, что и на уровне 2, но ослабляется жесткость требования непрерывности файлов. На этом уровне файл может состоять из нескольких разделов, каждый из которых представляет собой непрерывную последовательность блоков. Одна и та же последовательность блоков может несколько раз встречаться в одном файле и даже входить в несколько различных файлов. Такая организация файловой системы позволяет экономить место на диске.

Рок-Ридж расширения

Как было показано, стандарт ISO 9660 содержит много различных ограничений. Вскоре после выхода этого стандарта пользователи из UNIX-сообщества начали работу над его расширением, чтобы файловая система UNIX могла быть представлена на CD-ROM. Эти расширения получили название Рок-Ридж (Rock Ridge) по городу из фильма Джина Уайлдера Blazing Saddles (Огненные седла), вероятно, потому, что этот фильм нравился одному из членов комитета.

Расширение использует поле System use, чтобы CD-ROM формата Рок-Ридж мог читаться на любом компьютере. Все остальные поля соответствуют требованиям стандарта ISO 9660. Система, не знакомая с расширениями Рок-Ридж, просто игнорирует их и видит нормальный CD-ROM.

Расширения содержат следующие поля:

1. PX - Атрибуты POSIX.
2. PN — Старший и младший номера устройств.
3. SL — Символьная связь.
4. NM — Альтернативное имя.
4. CL — Расположение дочернего узла.
6. PL — Расположение дочернего узла.
7. RE — Перераспределение.
8. TF — Временные штампы.

Поле PX содержит стандартные биты разрешений rwxrwxtwx системы UNIX для владельца, группы и всех остальных. Оно также содержит остальные биты слова состояния, такие как SETUID, SETGID и т. п.

Чтобы необработанные устройства могли быть представлены на CD-ROM, вводится поле PN. Оно содержит старший и младший номера устройств, ассоциированных с файлом. Таким образом, содержимое каталога /dev может быть записано на CD-ROM и позднее правильно воссоздано на другой системе.

Поле SL используется для символьных связей. Оно позволяет файлу из одной файловой системы ссылаться на файл из другой файловой системы.

Вероятно, наиболее важным является поле NM. С его помощью можно указать для файла второе имя. Этого имени не касаются ограничения стандарта ISO 9660, что позволяет указывать произвольные имена файлов системы UNIX на CD-ROM.

Следующие три поля используются вместе, чтобы обойти ограничения стандарта ISO 9660 на глубину вложенности каталогов. С их помощью можно указать, куда в дереве иерархии должен быть перемещен тот или иной каталог.

Наконец, поле TF содержит три временных штампа, включаемые в каждый i-узел системы UNIX, а именно: время создания файла, последнего изменения файла и

последнего доступа к файлу. Все вместе эти расширения позволяют скопировать файловую систему UNIX на CD-ROM, а затем корректно восстановить ее на другой машине.

Расширения Joliet

Сообщество UNIX было не единственной группой, которой требовалось расширение стандарта ISO 9660. Корпорация Microsoft также пришла к выводу, что стандарт ISO 9660 содержит слишком много ограничений (хотя большинство ограничений было вызвано в первую очередь требованием совместимости с файловой системой MS-DOS фирмы Microsoft). Поэтому корпорация Microsoft разработала некоторые расширения, названные Joliet. Они должны были позволить копировать на CD-ROM-диск и восстанавливать с него файловую систему Windows подобно тому, как расширения Рок-Ридж позволяли работать с файловой системой UNIX. Теоретически все программы, работающие в операционной системе Windows и использующие CD-ROM, включая программы записи на CD-R, поддерживают расширение Joliet.

Основными расширениями, содержащимися в Joliet, являются:

1. Длинные имена файлов.
2. Набор символов Unicode.
3. Глубина вложенности каталогов, превышающая восемь уровней.
4. Имена каталогов с расширениями.

Первое расширение позволяет использовать имена файлов длиной до 64 символов. Второе расширение разрешает использовать для имен файлов символы Unicode. Это расширение важно для программного обеспечения, предназначенного для распространения в странах, в которых не используется латинский алфавит, таких как Япония, Израиль или Греция'. Поскольку символы Unicode занимают два байта, максимальное имя файла в расширении Joliet занимает 128 байт.

Как и Рок-Ридж, расширение Joliet устраняет ограничение на глубину вложенности каталогов. Каталоги могут вкладываться друг в друга на любую требуемую глубину. Наконец, у имен каталогов могут быть расширения. Неясно, почему было включено такое расширение стандарта, поскольку каталоги в файловой системе Windows практически никогда не используют расширений, но, возможно, однажды они потребуются.