Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

К. С. Курочка, Д. А. Литвинов

ПРИНЦИПЫ ОРГАНИЗАЦИИ ВНЕШНЕЙ И ВНУТРЕННЕЙ ПАМЯТИ В ОПЕРАЦИОННЫХ СИСТЕМАХ

**Пособие
по курсам «Операционные системы»
и «Основы мультипроцессорной
и мультипрограммной обработки данных»
для студентов специальности 1-40 01 02
«Информационные системы и технологии
(по направлениям)»
дневной формы обучения**

Гомель 2010

УДК 004.451(075.8)
ББК 32.973-018.2я73
К93

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем ГГТУ им. П. О. Сухого
(*протокол № 9 от 10.05.2010 г.*)

Рецензент: зав. каф. «Промышленная электроника» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *Ю. В. Крышнев*

Курочка, К. С.

К93 Принципы организации внешней и внутренней памяти в операционных системах : пособие по курсам «Операционные системы» и «Основы мультипроцессорной и мультипрограммной обработки данных» для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» днев. формы обучения / К. С. Курочка, Д. А. Литвинов. – Гомель : ГГТУ им. П. О. Сухого, 2010. – 91 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: http://lib.gstu.local. – Загл. с титул. экрана.

Пособие содержит основные теоретические сведения по существующим системам управления оперативной памятью в различных операционных системах.

Для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения.

.

УДК 004.451(075.8)
ББК 32.973-018.2я73

ВВЕДЕНИЕ

Любое программное обеспечение в процессе своей работы активно использует оперативную память, неоднократно выполняет запросы к операционной системе на выделение, освобождение и перераспределение оперативной памяти. Поэтому разработка качественного и эффективного программного обеспечения невозможно без знания архитектурных особенностей обработки данных запросов операционной системой.

Практическое пособие содержит краткие теоретические и практические сведения, необходимые для освоения эффективных приёмов работы с оперативной памятью в мультипроцессном программном обеспечении различных операционных систем.

1. Организация памяти компьютера.

Главная задача компьютерной системы – выполнять программы. Программы вместе с данными, к которым они имеют доступ, в процессе выполнения должны (по крайней мере частично) находиться в оперативной памяти. Операционной системе приходится решать задачу распределения памяти между пользовательскими процессами и компонентами ОС. Эта деятельность называется управлением памятью. Таким образом, память (storage, memory) является важнейшим ресурсом, требующим тщательного управления. В недавнем прошлом память была самым дорогим ресурсом.

Часть ОС, которая отвечает за управление памятью, называется менеджером памяти.

1.1. Физическая организация памяти компьютера

Запоминающие устройства компьютера разделяют, как минимум, на два уровня: основную (главную, оперативную, физическую) и вторичную (внешнюю) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно ос-

новная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Эту схему можно дополнить еще несколькими промежуточными уровнями, как показано на рис. 1.1. Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости.

Рис. 1.1. Иерархия памяти

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

1.2. Локальность

Оказывается, при таком способе организации по мере снижения скорости доступа к уровню памяти снижается также и частота обращений к нему.

Ключевую роль здесь играет свойство реальных программ, в течение ограниченного отрезка времени способных работать с небольшим набором адресов памяти. Это эмпирически наблюдаемое свойство известно как принцип локальности или локализации обращений.

Свойство локальности (соседние в пространстве и времени объекты характеризуются похожими свойствами) присуще не только функционированию ОС, но и природе вообще. В случае ОС свойство локальности объяснимо, если учесть, как пишутся программы и как хранятся данные, то есть обычно в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных. Эту часть кода и данных удается разместить в памяти с быстрым доступом. В результате реальное время доступа к памяти определяется временем доступа к верхним уровням, что и обусловливает эффективность использования иерархической схемы.

Надо сказать, что описываемая организация вычислительной системы во многом имитирует деятельность человеческого мозга при переработке информации. Действительно, решая конкретную проблему, человек работает с небольшим объемом информации, храня не относящиеся к делу сведения в своей памяти или во внешней памяти (например, в книгах).

Кэш процессора обычно является частью аппаратуры, поэтому менеджер памяти ОС занимается распределением информации главным образом в основной и внешней памяти компьютера. В некоторых схемах потоки между оперативной и внешней памятью регулируются программистом (см. например, далее оверлейные структуры), однако это связано с затратами времени программиста, так что подобную деятельность стараются возложить на ОС.

Адреса в основной памяти, характеризующие реальное расположение данных в физической памяти, называются физическими адресами. Набор физических адресов, с которым работает программа, называют физическим адресным пространством.

1.3. Логическая память

Аппаратная организация памяти в виде линейного набора ячеек не соответствует представлениям программиста о том, как организовано хранение программ и данных. Большинство программ представляет собой набор модулей, созданных независимо друг от друга. Иногда все модули, входящие в состав процесса, располагаются в памяти один за другим, образуя линейное пространство адресов. Однако чаще модули помещаются в разные области памяти и используются поразному.

Схема управления памятью, поддерживающая этот взгляд пользователя на то, как хранятся программы и данные, называется сегментацией. Сегмент – область памяти определенного назначения, внутри которой поддерживается линейная адресация. Сегменты содержат процедуры, массивы, стек или скалярные величины, но обычно не содержат информацию смешанного типа.

По-видимому, вначале сегменты памяти появились в связи с необходимостью обобществления процессами фрагментов программного кода (текстовый редактор, тригонометрические библиотеки и т. д.), без чего каждый процесс должен был хранить в своем адресном пространстве дублирующую информацию. Эти отдельные участки памяти, хранящие информацию, которую система отображает в память нескольких процессов, получили название сегментов. Память, таким образом, перестала быть линейной и превратилась в двумерную. Адрес состоит из двух компонентов: номер сегмента, смещение внутри сегмента. Далее оказалось удобным размещать в разных сегментах различные компоненты процесса (код программы, данные, стек и т. д.). Попутно выяснилось, что можно контролировать характер работы с конкретным сегментом, приписав ему атрибуты, например права доступа или типы операций, которые разрешается производить с данными, хранящимися в сегменте.

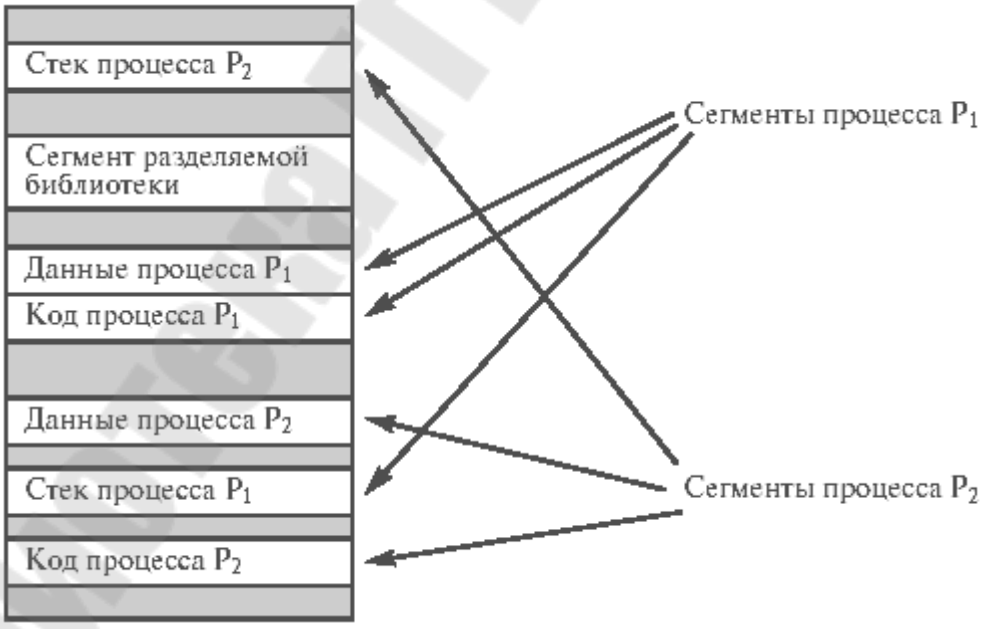

Рис. 1.2. Расположение сегментов процессов в памяти компьютера

Некоторые сегменты, описывающие адресное пространство процесса, показаны на рис. 1.2.

Большинство современных ОС поддерживают сегментную организацию памяти. В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием.

Адреса, к которым обращается процесс, таким образом, отличаются от адресов, реально существующих в оперативной памяти. В каждом конкретном случае используемые программой адреса могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические. Компилятор связывает эти символические адреса с перемещаемыми адресами (такими, как n байт от начала модуля). Подобный адрес, сгенерированный программой, обычно называют логическим (в системах с виртуальной памятью он часто называется виртуальным) адресом. Совокупность всех логических адресов называется логическим (виртуальным) адресным пространством.

1.4. Связывание адресов

Логические и физические адресные пространства ни по организации, ни по размеру не соответствуют друг другу. Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, 2^{32}) и в современных системах значительно превышает размер физического адресного пространства. Следовательно, процессор и ОС должны быть способны отобразить ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти. Такое отображение адресов называют трансляцией (привязкой) адреса или связыванием адресов (см. рис. 1.3).

Связывание логического адреса, порожденного оператором программы, с физическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Таким образом, привязка инструкций и данных к памяти в принципе может быть сделана на следующих шагах.

Этап компиляции (Compile time). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда непосредственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код. В качестве примера можно привести .com программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.

Этап загрузки (Load time). Если информация о размещении программы на стадии компиляции отсутствует, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.

Этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, связывание откладывается до стадии выполнения. Здесь желательно наличие специализированного оборудования, например регистров перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Большинство современных ОС осуществляет трансляцию адресов на этапе выполнения, используя для этого специальный аппаратный механизм.

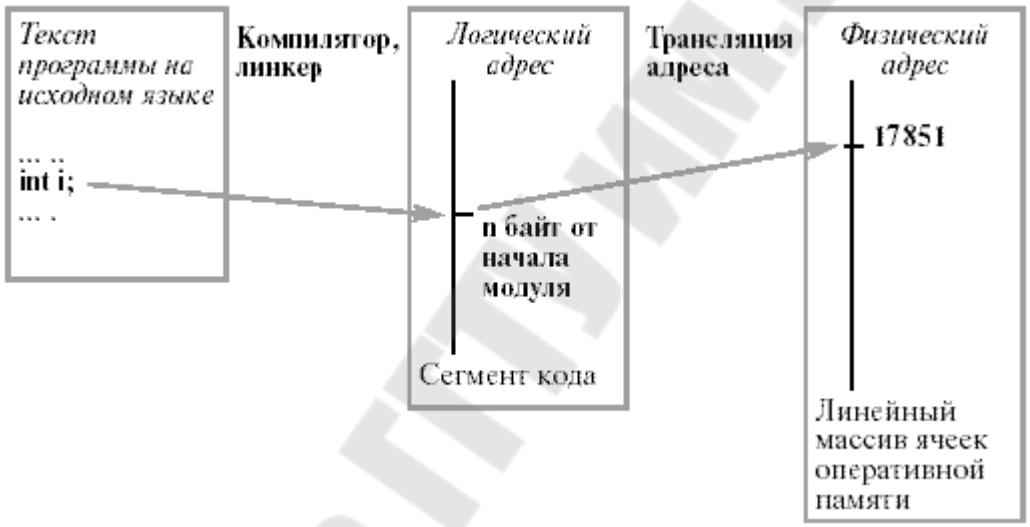

Рис. 1.3. Формирование логического адреса и связывание логического адреса с физическим

1.5. ФУНКЦИИ СИСТЕМЫ УПРАВЛЕНИЯ ПАМЯТЬЮ

Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:

- отображение адресного пространства процесса на конкретные области физической памяти;
- распределение памяти между конкурирующими процессами;
- контроль доступа к адресным пространствам процессов;

– выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
– учет свободной и занятой памяти.

2. Простейшие схемы управления памятью

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в основной памяти, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти. Затем появился "простой свопинг" (система по-прежнему размещает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю и заменяет его в основной памяти образом другого процесса). Такого рода схемы имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

2.1. Схема с фиксированными разделами

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции.

Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов (см. рис. 2.1).

Эта схема была реализована в IBM OS/360 (MFT), DEC RSX-11 и ряде других систем.

Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов.

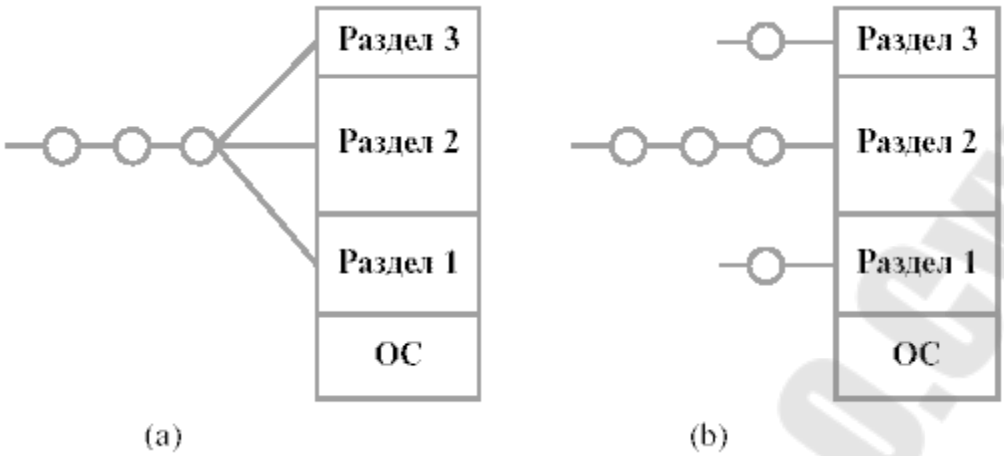

Рис. 2.1. Схема с фиксированными разделами: (a) – с общей очередью процессов, (b) – с отдельными очередями процессов

Очевидный недостаток этой схемы – число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от внутренней фрагментации – потери части памяти, выделенной процессу, но не используемой им. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

2.2. Один процесс в памяти

Частный случай схемы с фиксированными разделами – работа менеджера памяти однозадачной ОС. В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС – в верхней части памяти, в нижней или в средней. Причем часть ОС может быть в ROM (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение, – расположение вектора прерываний, который обычно локализован в нижней части памяти, поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS.

Защита адресного пространства ОС от пользовательской программы может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

2.3. Оверлейная структура

Так как размер логического адресного пространства процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная идея – держать в памяти только те инструкции программы, которые нужны в данный момент.
Потребность в таком способе загрузки появляется, если логическое адресное пространство системы мало, например 1 Мбайт (MS-DOS) или даже всего 64 Кбайта (PDP-11), а программа относительно велика. На современных 32-разрядных системах, где виртуальное адресное пространство измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами (см. раздел "Виртуальная память").

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .odl), описывающим дерево вызовов внутри программы.

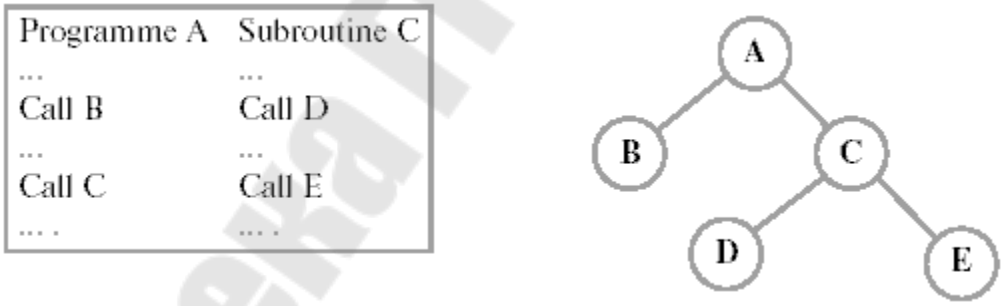

Рис. 2.2. Организация структуры с перекрытием. Можно поочередно загружать в память ветви A-B, A-C-D и A-C-E программы

Для примера, приведенного на рис. 2.2, текст этого файла может выглядеть так:
A-(B,C)

C-(D,E)

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к физической памяти происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. ОС при этом лишь делает несколько больше операций ввода-вывода. Типовое решение – порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

Тщательное проектирование оверлейной структуры отнимает много времени и требует знания устройства программы, ее кода, данных и языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с небольшим логическим адресным пространством. Как мы увидим в дальнейшем, проблема оверлейных сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что возможность организации структур с перекрытиями во многом обусловлена свойством локальности, которое позволяет хранить в памяти только ту информацию, которая необходима в конкретный момент вычислений.

2.4. Динамическое распределение. Свопинг

Имея дело с пакетными системами, можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) – перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging) и будет рассмотрена ниже.

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста.

Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

2.5. Схема с переменными разделами

В принципе, система свопинга может базироваться на фиксированных разделах. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера (рис. 2.3). Смежные свободные участки могут быть объединены.

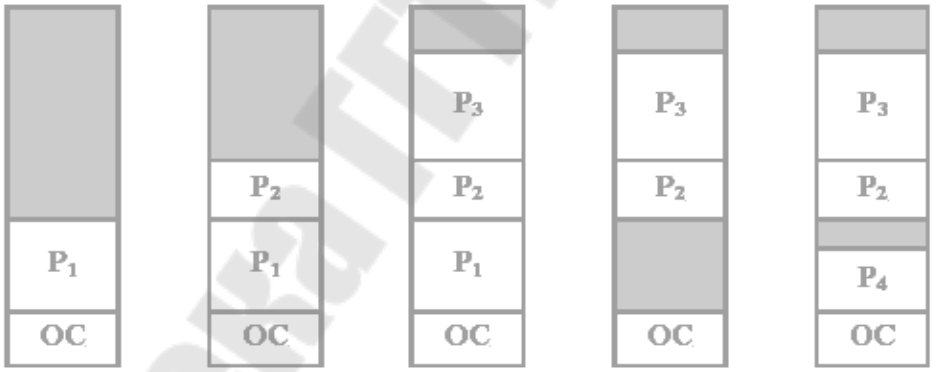

Рис. 2.3. Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

В какой раздел помещать процесс? Наиболее распространены три стратегии:

- стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел;
- стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места;

– стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща внешняя фрагментация – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации. Любопытно, что метод наиболее подходящего может оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем 1/3 памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы внешней фрагментации – организовать сжатие, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале фрагментация после сжатия должна отсутствовать. Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

2.6. Страничная память

Описанные выше схемы недостаточно эффективно используют память, поэтому в современных схемах управления памятью не принято размещать процесс в оперативной памяти одним непрерывным блоком.

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют физическими страницами или страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе – упорядоченная пара (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой таблицу страниц, которая хранится в оперативной памяти. Иногда говорят, что таблица страниц – это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на рис. 2.4. Если выполняемый процесс обращается к логическому адресу v = (p,d), механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

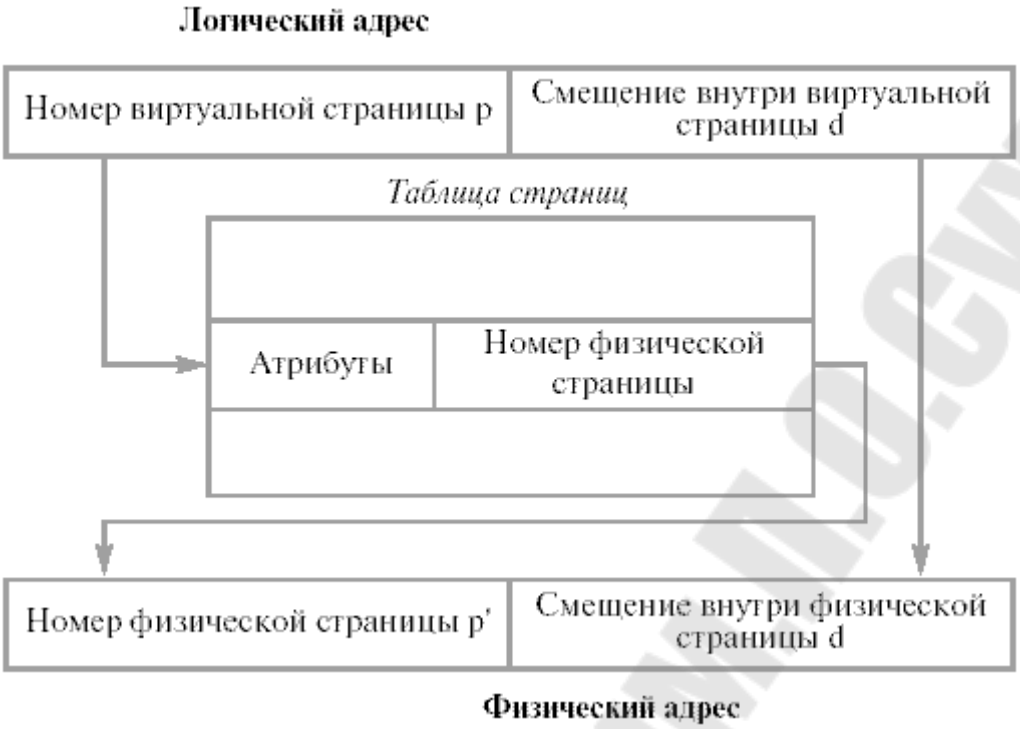

Рис. 2.4. Связь логического и физического адресов при страничной организации памяти

Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память – единое непрерывное пространство, содержащее только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При пере-

ключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

2.7. Сегментная и сегментно-страничная организация памяти

Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. Сегменты, в отличие от страниц, могут иметь переменный размер. Идея сегментации изложена во введении. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера сегмента и смещения внутри сегмента. Подчеркнем, что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек...).

Программисты, пишущие на языках низкого уровня, должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое адресное пространство – набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия). В отличие от страничной схемы, где пользователь задает только один адрес, который разбивается на номер страницы и смещение прозрачным для программиста образом, в сегментной схеме пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением.

Каждый сегмент – линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью процессора (при 32-разрядной адресации это 2^{32} байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.

Логический адрес – упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программа обращается к этим дескрипторам по номерам-селекторам. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и т. д. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.

Аппаратная поддержка сегментов распространена мало (главным образом на процессорах Intel). В большинстве ОС сегментация реализуется на уровне, не зависящем от аппаратуры.

Хранить в памяти сегменты большого размера целиком так же неудобно, как и хранить процесс непрерывным блоком. Напрашивается идея разбиения сегментов на страницы. При сегментно-страничной организации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей: номера сегмента логической памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

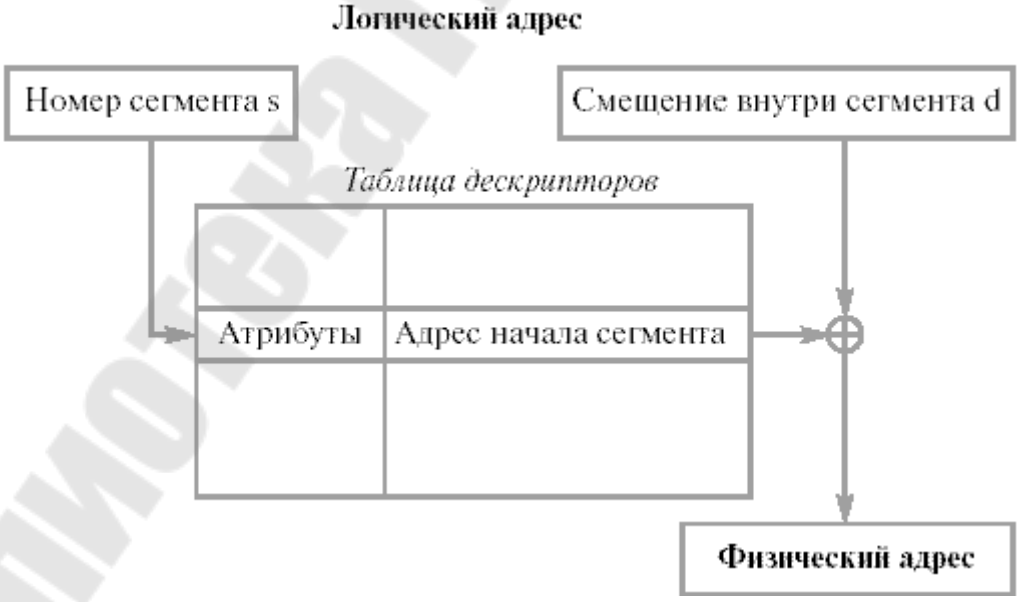

Рис. 2.5. Преобразование логического адреса при сегментной организации памяти

Сегментно-страничная и страничная организация памяти позволяет легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных.

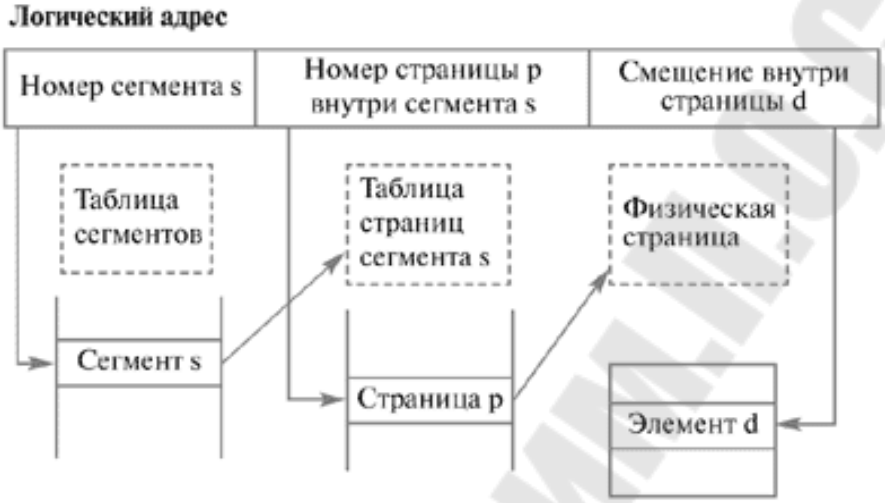

Рис. 2.6. Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

3. Виртуальная память.

3.1. Понятие виртуальной памяти

Разработчикам программного обеспечения часто приходится решать проблему размещения в памяти больших программ, размер которых превышает объем доступной оперативной памяти. Один из вариантов решения данной проблемы – организация структур с перекрытием – рассмотрен в предыдущей лекции. При этом предполагалось активное участие программиста в процессе формирования перекрывающихся частей программы. Развитие архитектуры компьютеров и расширение возможностей операционной системы по управлению памятью позволило переложить решение этой задачи на компьютер. Одним из главных достижений стало появление виртуальной памяти (virtual memory). Впервые она была реализована в 1959 г. на компьютере «Атлас», разработанном в Манчестерском университете.

Суть концепции виртуальной памяти заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах виртуальной памяти у процес-

са создается иллюзия того, что вся необходимая ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц.

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. Принцип локальности обеспечивает этой схеме нужную эффективность.

Возможность выполнения программы, находящейся в памяти лишь частично, имеет ряд вполне очевидных преимуществ. Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти. Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы. Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы «видимости» практически неограниченной (характерный размер для 32-разрядных архитектур 2^{32} = 4 Гбайт) адресуемой пользовательской памяти (логическое адресное пространство) при наличии основной памяти существенно меньших размеров (физическое адресное пространство) – очень важный аспект.

Но введение виртуальной памяти позволяет решать другую, не менее важную задачу – обеспечение контроля доступа к отдельным сегментам памяти и, в частности, защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ. Каждый процесс работает со своими виртуальными адресами, трансляцию которых в физические выполняет аппаратура компьютера. Таким обра-

зом, пользовательский процесс лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.

Например, 16-разрядный компьютер PDP-11/70 с 64 Кбайт логической памяти мог иметь до 2 Мбайт оперативной памяти. Операционная система этого компьютера тем не менее поддерживала виртуальную память, которая обеспечивала защиту и перераспределение основной памяти между пользовательскими процессами.

Напомним, что в системах с виртуальной памятью те адреса, которые генерирует программа (логические адреса), называются виртуальными, и они формируют виртуальное адресное пространство. Термин «виртуальная память» означает, что программист имеет дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Хотя известны и чисто программные реализации виртуальной памяти, это направление получило наиболее широкое развитие после соответствующей аппаратной поддержки.

Следует отметить, что оборудование компьютера принимает участие в трансляции адреса практически во всех схемах управления памятью. Но в случае виртуальной памяти это становится более сложным вследствие разрывности отображения и многомерности логического адресного пространства. Может быть, наиболее существенным вкладом аппаратуры в реализацию описываемой схемы является автоматическая генерация исключительных ситуаций при отсутствии в памяти нужных страниц (page fault).

Любая из трех ранее рассмотренных схем управления памятью – страничной, сегментной и сегментно-страничной – пригодна для организации виртуальной памяти. Чаще всего используется сегментно-страничная модель, которая является синтезом страничной модели и идеи сегментации. Причем для тех архитектур, в которых сегменты не поддерживаются аппаратно, их реализация – задача архитектурно-независимого компонента менеджера памяти. Сегментная организация в чистом виде встречается редко.

3.2 Архитектурные средства поддержки виртуальной памяти

Очевидно, что невозможно создать полностью машинно-независимый компонент управления виртуальной памятью. С другой стороны, имеются существенные части программного обеспечения,

связанного с управлением виртуальной памятью, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных ОС является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом входит в аппаратно-зависимую часть подсистемы управления виртуальной памятью.

В самом распространенном случае необходимо отобразить большое виртуальное адресное пространство в физическое адресное пространство существенно меньшего размера. Пользовательский процесс или ОС должны иметь возможность осуществить запись по виртуальному адресу, а задача ОС – сделать так, чтобы записанная информация оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю память). В случае виртуальной памяти система отображения адресных пространств помимо трансляции адресов должна предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в данный момент находятся в физической памяти и где именно размещаются.

3.3 Страничная виртуальная память

Как и в случае простой страничной организации, страничная виртуальная память и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы виртуальный адрес преобразуется в упорядоченную пару (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, таблица страниц принимает специфический вид (см. раздел «Структура таблицы страниц»), структура записей становится более сложной, среди атрибутов страницы появляются биты присутствия, модификации и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая страничное нарушение (page fault) или страничный отказ. Обработка страничного нарушения заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных кадров на диск выгружается редко используемая страница. Проблемы замещения страниц и обработки страничных нарушений рассматриваются в следующей лекции.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

В большинстве современных компьютеров со страничной организацией в основной памяти хранится лишь часть таблицы страниц, а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается, как будет показано ниже, за счет использования сверхбыстродействующей памяти, размещенной в кэше процессора.

3.4 Сегментно-страничная организации виртуальной памяти

Как и в случае простой сегментации, в схемах виртуальной памяти сегмент – это линейная последовательность адресов, начинающаяся с 0. При организации виртуальной памяти размер сегмента может быть велик, например может превышать размер оперативной памяти. Повторяя все ранее приведенные рассуждения о размещении в памяти больших программ, приходим к разбиению сегментов на страницы и необходимости поддержки своей таблицы страниц для каждого сегмента. На практике, однако, появления в системе большого количества таблиц страниц стараются избежать, организуя неперекрывающиеся сегменты в одном виртуальном пространстве, для описания которого хватает одной таблицы страниц. Таким образом, одна таблица страниц отводится для всего процесса. Например, в популярных ОС

Linux и Windows 2000 все сегменты процесса, а также область памяти ядра ограничены виртуальным адресным пространством объемом 4 Гбайт. При этом ядро ОС располагается по фиксированным виртуальным адресам вне зависимости от выполняемого процесса.

3.5 Структура таблицы страниц

Организация таблицы страниц – один из ключевых элементов отображения адресов в страничной и сегментно-страничной схемах. Рассмотрим структуру таблицы страниц для случая страничной организации более подробно.

Итак, виртуальный адрес состоит из виртуального номера страницы и смещения. Номер записи в таблице страниц соответствует номеру виртуальной страницы. Размер записи колеблется от системы к системе, но чаще всего он составляет 32 бита. Из этой записи в таблице страниц находится номер кадра для данной виртуальной страницы, затем прибавляется смещение и формируется физический адрес. Помимо этого запись в таблице страниц содержит информацию об атрибутах страницы. Это биты присутствия и защиты (например, 0 – read/write, 1 – read only...). Также могут быть указаны: бит модификации, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость перезаписи страницы на диск; бит ссылки, который помогает выделить малоиспользуемые страницы; бит, разрешающий кэширование, и другие управляющие биты. Заметим, что адреса страниц на диске не являются частью таблицы страниц.

Основную проблему для эффективной реализации таблицы страниц создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодня являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства размером 4 Гбайт (для 64-разрядных компьютеров эта величина равна 2^{64} байт). Кроме того, существует проблема скорости отображения, которая решается за счет использования так называемой ассоциативной памяти.

Подсчитаем примерный размер таблицы страниц. В 32-битном адресном пространстве при размере страницы 4 Кбайт (Intel) получаем $2^{32}/2^{12}=2^{20}$, то есть приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно

миллион строк (entry), причем запись в строке состоит из нескольких байтов. Заметим, что каждый процесс нуждается в своей таблице страниц (а в случае сегментно-страничной схемы желательно иметь по одной таблице страниц на каждый сегмент).

Понятно, что количество памяти, отводимое таблицам страниц, не может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты таблицы страниц. В силу свойства локальности число таких фрагментов относительно невелико. Выполнить разбиение таблицы страниц на части можно по-разному. Наиболее распространенный способ разбиения – организация так называемой многоуровневой таблицы страниц. Для примера рассмотрим двухуровневую таблицу с размером страниц 4 Кбайт, реализованную в 32-разрядной архитектуре Intel.

Таблица, состоящая из 2^{20} строк, разбивается на 2^{10} таблиц второго уровня по 2^{10} строк. Эти таблицы второго уровня объединены в общую структуру при помощи одной таблицы первого уровня, состоящей из 2^{10} строк. 32-разрядный адрес делится на 10-разрядное поле p1, 10-разрядное поле p2 и 12-разрядное смещение d. Поле p1 указывает на нужную строку в таблице первого уровня, поле p2 – второго, а поле d локализует нужный байт внутри указанного страничного кадра (см. рис. 3.1).

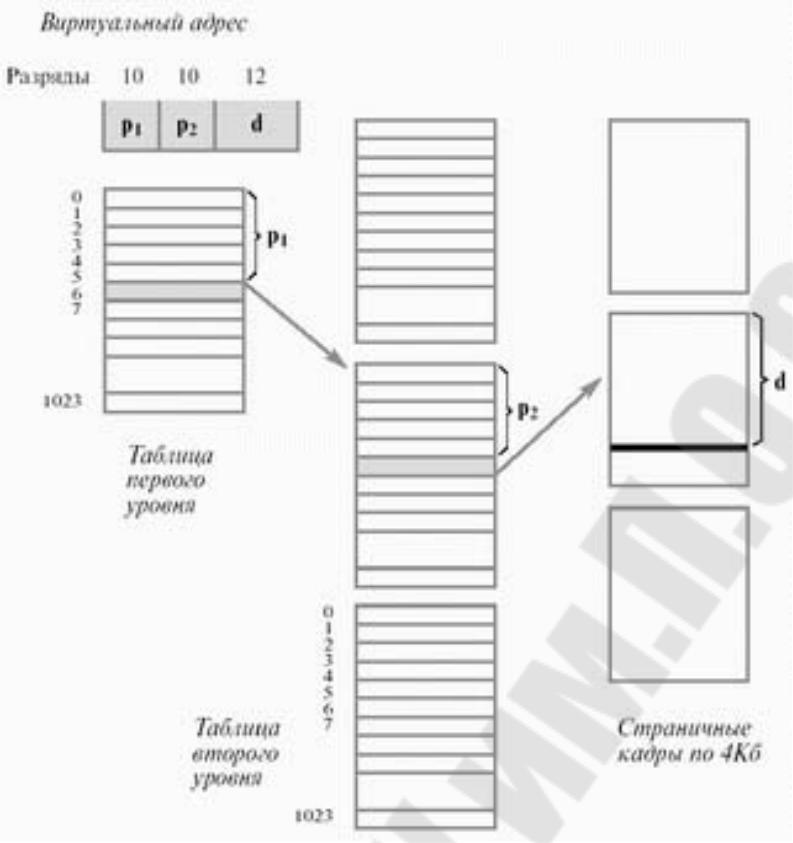

Рис. 3.1. Пример двухуровневой таблицы страниц

При помощи всего лишь одной таблицы второго уровня можно охватить 4 Мбайт (4 Кбайт х 1024) оперативной памяти. Таким образом, для размещения процесса с большим объемом занимаемой памяти достаточно иметь в оперативной памяти одну таблицу первого уровня и несколько таблиц второго уровня. Очевидно, что суммарное количество строк в этих таблицах много меньше 2^{20}. Такой подход естественным образом обобщается на три и более уровней таблицы.

Наличие нескольких уровней, естественно, снижает производительность менеджера памяти. Несмотря на то что размеры таблиц на каждом уровне подобраны так, чтобы таблица помещалась целиком внутри одной страницы, обращение к каждому уровню – это отдельное обращение к памяти. Таким образом, трансляция адреса может потребовать нескольких обращений к памяти.

Количество уровней в таблице страниц зависит от конкретных особенностей архитектуры. Можно привести примеры реализации одноуровневого (DEC PDP-11), двухуровневого (Intel, DEC VAX), трехуровневого (Sun SPARC, DEC Alpha) пейджинга, а также пейджинга с заданным количеством уровней (Motorola). Функционирование RISC-процессора MIPS R2000 осуществляется вообще без табли-

цы страниц. Здесь поиск нужной страницы, если эта страница отсутствует в ассоциативной памяти, должна взять на себя ОС (так называемый zero level paging).

3.6 Ассоциативная память

Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти, поэтому занимает много времени. В некоторых случаях такая задержка недопустима. Проблема ускорения поиска решается на уровне архитектуры компьютера.

В соответствии со свойством локальности большинство программ в течение некоторого промежутка времени обращаются к небольшому количеству страниц, поэтому активно используется только небольшая часть таблицы страниц.

Естественное решение проблемы ускорения – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц, то есть иметь небольшую, быструю кэш-память, хранящую необходимую на данный момент часть таблицы страниц. Это устройство называется ассоциативной памятью, иногда также употребляют термин буфер поиска трансляции (translation lookaside buffer – TLB).

Одна запись таблицы в ассоциативной памяти (один вход) содержит информацию об одной виртуальной странице: ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц.

Так как ассоциативная память содержит только некоторые из записей таблицы страниц, каждая запись в TLB должна включать поле с номером виртуальной страницы. Память называется ассоциативной, потому что в ней происходит одновременное сравнение номера отображаемой виртуальной страницы с соответствующим полем во всех строках этой небольшой таблицы. Поэтому данный вид памяти достаточно дорого стоит. В строке, поле виртуальной страницы которой совпало с искомым значением, находится номер страничного кадра. Обычное число записей в TLB от 8 до 4096. Рост количества записей в ассоциативной памяти должен осуществляться с учетом таких факторов, как размер кэша основной памяти и количества обращений к памяти при выполнении одной команды.

3.7 Менеджер памяти при наличии ассоциативной памяти.

Вначале информация об отображении виртуальной страницы в физическую отыскивается в ассоциативной памяти. Если нужная запись найдена – все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется.

Если нужная запись в ассоциативной памяти отсутствует, отображение осуществляется через таблицу страниц. Происходит замена одной из записей в ассоциативной памяти найденной записью из таблицы страниц. Здесь мы сталкиваемся с традиционной для любого кэша проблемой замещения (а именно какую из записей в кэше необходимо изменить). Конструкция ассоциативной памяти должна организовывать записи таким образом, чтобы можно было принять решение о том, какая из старых записей должна быть удалена при внесении новых.

Число удачных поисков номера страницы в ассоциативной памяти по отношению к общему числу поисков называется hit (совпадение) ratio (пропорция, отношение). Иногда также используется термин «процент попаданий в кэш». Таким образом, hit ratio – часть ссылок, которая может быть сделана с использованием ассоциативной памяти. Обращение к одним и тем же страницам повышает hit ratio. Чем больше hit ratio, тем меньше среднее время доступа к данным, находящимся в оперативной памяти.

Предположим, например, что для определения адреса в случае кэш-промаха через таблицу страниц необходимо 100 нс, а для определения адреса в случае кэш-попадания через ассоциативную память – 20 нс. С 90% hit ratio среднее время определения адреса – 0,9x20+0,1x100 = 28 нс.

Вполне приемлемая производительность современных ОС доказывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

Необходимо обратить внимание на следующий факт. При переключении контекста процессов нужно добиться того, чтобы новый процесс «не видел» в ассоциативной памяти информацию, относящуюся к предыдущему процессу, например очищать ее. Таким образом, использование ассоциативной памяти увеличивает время переключения контекста.

Рассмотренная двухуровневая (ассоциативная память + таблица страниц) схема преобразования адреса является ярким примером иерархии памяти, основанной на использовании принципа локальности.

3.8 Инвертированная таблица страниц

Несмотря на многоуровневую организацию, хранение нескольких таблиц страниц большого размера по-прежнему представляют собой проблему. Ее значение особенно актуально для 64-разрядных архитектур, где число виртуальных страниц очень велико. Вариантом решения является применение инвертированной таблицы страниц (inverted page table). Этот подход применяется на машинах PowerPC, некоторых рабочих станциях Hewlett-Packard, IBM RT, IBM AS/400 и ряде других.

В этой таблице содержится по одной записи на каждый страничный кадр физической памяти. Существенно, что достаточно одной таблицы для всех процессов. Таким образом, для хранения функции отображения требуется фиксированная часть основной памяти, независимо от разрядности архитектуры, размера и количества процессов. Например, для компьютера Pentium с 256 Мбайт оперативной памяти нужна таблица размером 64 Кбайт строк.

Несмотря на экономию оперативной памяти, применение инвертированной таблицы имеет существенный минус – записи в ней (как и в ассоциативной памяти) не отсортированы по возрастанию номеров виртуальных страниц, что усложняет трансляцию адреса. Один из способов решения данной проблемы – использование хеш-таблицы виртуальных адресов. При этом часть виртуального адреса, представляющая собой номер страницы, отображается в хеш-таблицу с использованием функции хеширования. Каждой странице физической памяти здесь соответствует одна запись в хеш-таблице и инвертированной таблице страниц. Виртуальные адреса, имеющие одно значение хеш-функции, сцепляются друг с другом. Обычно длина цепочки не превышает двух записей.

3.9 Размер страницы

Разработчики ОС для существующих машин редко имеют возможность влиять на размер страницы. Однако для вновь создаваемых компьютеров решение относительно оптимального размера страницы

является актуальным. Как и следовало ожидать, нет одного наилучшего размера. Скорее есть набор факторов, влияющих на размер. Обычно размер страницы – это степень двойки от 2^9 до 2^{14} байт.

Чем больше размер страницы, тем меньше будет размер структур данных, обслуживающих преобразование адресов, но тем больше будут потери, связанные с тем, что память можно выделять только постранично.

Как следует выбирать размер страницы? Во-первых, нужно учитывать размер таблицы страниц, здесь желателен большой размер страницы (страниц меньше, соответственно и таблица страниц меньше). С другой стороны, память лучше утилизируется с маленьким размером страницы. В среднем половина последней страницы процесса пропадает. Необходимо также учитывать объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Проблема не имеет идеального решения. Историческая тенденция состоит в увеличении размера страницы.

Как правило, размер страниц задается аппаратно, например в DEC PDP-11 – 8 Кбайт, в DEC VAX – 512 байт, в других архитектурах, таких как Motorola 68030, размер страниц может быть задан программно. Учитывая все обстоятельства, в ряде архитектур возникают множественные размеры страниц, например в Pentium размер страницы колеблется от 4 Кбайт до 8 Кбайт. Тем не менее большинство коммерческих ОС ввиду сложности перехода на множественный размер страниц поддерживают только один размер страниц.

4 Управление виртуальной памятью

4.1 Исключительные ситуации при работе с памятью

Отображение виртуального адреса в физический осуществляется при помощи таблицы страниц. Для каждой виртуальной страницы запись в таблице страниц содержит номер соответствующего страничного кадра в оперативной памяти, а также атрибуты страницы для контроля обращений к памяти.

Что же происходит, когда нужной страницы в памяти нет или операция обращения к памяти недопустима? Естественно, что операционная система должна быть как-то оповещена о происшедшем. Обычно для этого используется механизм исключительных ситуаций (exceptions). При попытке выполнить подобное обращение к вирту-

альной странице возникает исключительная ситуация "страничное нарушение" (page fault), приводящая к вызову специальной последовательности команд для обработки конкретного вида страничного нарушения.

Страничное нарушение может происходить в самых разных случаях: при отсутствии страницы в оперативной памяти, при попытке записи в страницу с атрибутом "только чтение" или при попытке чтения или записи страницы с атрибутом "только выполнение". В любом из этих случаев вызывается обработчик страничного нарушения, являющийся частью операционной системы. Ему обычно передается причина возникновения исключительной ситуации и виртуальный адрес, обращение к которому вызвало нарушение. Нас будет интересовать конкретный вариант страничного нарушения - обращение к отсутствующей странице, поскольку именно его обработка во многом определяет производительность страничной системы. Когда программа обращается к виртуальной странице, отсутствующей в основной памяти, операционная система должна выделить страницу основной памяти, переместить в нее копию виртуальной страницы из внешней памяти и модифицировать соответствующий элемент таблицы страниц.

Повышение производительности вычислительной системы может быть достигнуто за счет уменьшения частоты страничных нарушений, а также за счет увеличения скорости их обработки. Время эффективного доступа к отсутствующей в оперативной памяти странице складывается из:

1) обслуживания исключительной ситуации (page fault);
2) чтения (подкачки) страницы из вторичной памяти (иногда, при недостатке места в основной памяти, необходимо вытолкнуть одну из страниц из основной памяти во вторичную, то есть осуществить замещение страницы);
3) возобновления выполнения процесса, вызвавшего данный page fault.

Для решения первой и третьей задач ОС выполняет до нескольких сот машинных инструкций в течение нескольких десятков микросекунд. Время подкачки страницы близко к нескольким десяткам миллисекунд. Проведенные исследования показывают, что вероятности page fault 5x10-7 оказывается достаточно, чтобы снизить производительность страничной схемы управления памятью на 10%. Таким образом, уменьшение частоты page faults является одной из ключевых задач

системы управления памятью. Ее решение обычно связано с правильным выбором алгоритма замещения страниц.

4.2 Стратегии управления страничной памятью

Программное обеспечение подсистемы управления памятью связано с реализацией следующих стратегий: стратегия выборки (fetch policy), стратегия размещения (placement policy), стратегия замещения (replacement policy).

Стратегия выборки (fetch policy) - в какой момент следует переписать страницу из вторичной памяти в первичную. Существует два основных варианта выборки - по запросу и с упреждением. Алгоритм выборки по запросу вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

Алгоритм выборки с упреждением осуществляет опережающее чтение, то есть кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (обычно соседние страницы располагаются во внешней памяти последовательно и могут быть считаны за одно обращение к диску). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе со значительными объемами данных или кода; кроме того, оптимизируется работа с диском.

Стратегия размещения (placement policy) - в какой участок первичной памяти поместить поступающую страницу. В системах со страничной организацией все просто - в любой свободный страничный кадр. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

Стратегия замещения (replacement policy) - какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Разумная стратегия замещения, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту страничных нарушений. Замещение должно происходить с учетом выделенного каждому процессу количества кадров. Кроме того, нужно решить, должна ли замещаемая страница принадлежать процессу,

который инициировал замещение, или она должна быть выбрана среди всех кадров основной памяти.

4.3 Алгоритмы замещения страниц

Итак, наиболее ответственным действием менеджера памяти является выделение кадра оперативной памяти для размещения в ней виртуальной страницы, находящейся во внешней памяти. Напомним, что мы рассматриваем ситуацию, когда размер виртуальной памяти для каждого процесса может существенно превосходить размер основной памяти. Это означает, что при выделении страницы основной памяти с большой вероятностью не удастся найти свободный страничный кадр. В этом случае операционная система в соответствии с заложенными в нее критериями должна:

- найти некоторую занятую страницу основной памяти;
- переместить в случае надобности ее содержимое во внешнюю память;
- переписать в этот страничный кадр содержимое нужной виртуальной страницы из внешней памяти;
- должным образом модифицировать необходимый элемент соответствующей таблицы страниц;
- продолжить выполнение процесса, которому эта виртуальная страница понадобилась.

Заметим, что при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения может быть оптимизирован за счет использования бита модификации (один из атрибутов страницы в таблице страниц). Бит модификации устанавливается компьютером, если хотя бы один байт был записан на страницу. При выборе кандидата на замещение проверяется бит модификации. Если бит не установлен, нет необходимости переписывать данную страницу на диск, ее копия на диске уже имеется. Подобный метод также применяется к read-only-страницам, они никогда не модифицируются. Эта схема уменьшает время обработки page fault.

Существует большое количество разнообразных алгоритмов замещения страниц. Все они делятся на локальные и глобальные. Локальные алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему стра-

ницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов. Глобальный же алгоритм замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют ряд недостатков. Во-первых, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе одновременно использует большое количество страниц памяти, то все остальные приложения будут в результате ощущать сильное замедление из-за недостатка кадров памяти для своей работы. Во-вторых, некорректно работающее приложение может подорвать работу всей системы (если, конечно, в системе не предусмотрено ограничение на размер памяти, выделяемой процессу), пытаясь захватить больше памяти. Поэтому в многозадачной системе иногда приходится использовать более сложные локальные алгоритмы. Применение локальных алгоритмов требует хранения в операционной системе списка физических кадров, выделенных каждому процессу. Этот список страниц иногда называют резидентным множеством процесса. В одном из следующих разделов рассмотрен вариант алгоритма подкачки, основанный на приведении резидентного множества в соответствие так называемому рабочему набору процесса.

Эффективность алгоритма обычно оценивается на конкретной последовательности ссылок к памяти, для которой подсчитывается число возникающих page faults. Эта последовательность называется строкой обращений (reference string). Мы можем генерировать строку обращений искусственным образом при помощи датчика случайных чисел или трассируя конкретную систему. Последний метод дает слишком много ссылок, для уменьшения числа которых можно сделать две вещи: для конкретного размера страниц можно запоминать только их номера, а не адреса, на которые идет ссылка; несколько подряд идущих ссылок на одну страницу можно фиксировать один раз.

Большинство процессоров имеют простейшие аппаратные средства, позволяющие собирать некоторую статистику обращений к памяти. Эти средства обычно включают два специальных флага на каждый элемент таблицы страниц. Флаг ссылки (reference бит) автоматически устанавливается, когда происходит любое обращение к этой странице, а уже рассмотренный выше флаг изменения (modify бит) устанавливается, если производится запись в эту страницу. Операци-

онная система периодически проверяет установку таких флагов, для того чтобы выделить активно используемые страницы, после чего значения этих флагов сбрасываются.

4.4 Алгоритм FIFO. Выталкивание первой пришедшей страницы

Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто созданием очереди страниц, в конец которой страницы попадают, когда загружаются в физическую память, а из начала берутся, когда требуется освободить память. Для замещения выбирается старейшая страница. К сожалению, эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц, например страниц кода текстового процессора при редактировании файла. Заметим, что при замещении активных страниц все работает корректно, но page fault происходит немедленно.

4.5 Аномалия Билэди (Belady)

На первый взгляд кажется очевидным, что чем больше в памяти страничных кадров, тем реже будут иметь место page faults. Удивительно, но это не всегда так.

	0	1	2	3	0	1	4	0	1	2	3	4
Самая старая страница	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Самая новая страница			0	1	2	3	0	0	0	1	4	4
	p	p	p	p	p	p	p			p	p	9 *page faults*

(a)

	0	1	2	3	0	1	4	0	1	2	3	4
Самая старая страница	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Самая новая страница				0	0	0	1	2	3	4	0	1
	p	p	p	p			p	p	p	p	p	p 10 *page faults*

(b)

Рис. 4.1. Аномалия Билэди: (a) - FIFO с тремя страничными кадрами; (b) - FIFO с четырьмя страничными кадрами

Как установил Билэди с коллегами, определенные последовательности обращений к страницам в действительности приводят к увеличению числа страничных нарушений при увеличении кадров, выделенных процессу. Это явление носит название "аномалии Билэди" или "аномалии FIFO".

Система с тремя кадрами (9 faults) оказывается более производительной, чем с четырьмя кадрами (10 faults), для строки обращений к памяти 012301401234 при выборе стратегии FIFO.

Аномалию Билэди следует считать скорее курьезом, чем фактором, требующим серьезного отношения, который иллюстрирует сложность ОС, где интуитивный подход не всегда приемлем.

4.6 Оптимальный алгоритм (OPT)

Одним из последствий открытия аномалии Билэди стал поиск оптимального алгоритма, который при заданной строке обращений имел бы минимальную частоту page faults среди всех других алгоритмов. Такой алгоритм был найден. Он прост: замещай страницу, которая не будет использоваться в течение самого длительного периода времени.

Каждая страница должна быть помечена числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Выталкиваться должна страница, для которой это число наибольшее.

Этот алгоритм легко описать, но реализовать невозможно. ОС не знает, к какой странице будет следующее обращение.

Зато мы можем сделать вывод, что для того, чтобы алгоритм замещения был максимально близок к идеальному алгоритму, система должна как можно точнее предсказывать обращения процессов к памяти. Данный алгоритм применяется для оценки качества реализуемых алгоритмов.

4.7 Выталкивание дольше всего не использовавшейся страницы. Алгоритм LRU

Одним из приближений к алгоритму OPT является алгоритм, исходящий из эвристического правила, что недавнее прошлое - хороший ориентир для прогнозирования ближайшего будущего.

Ключевое отличие между FIFO и оптимальным алгоритмом заключается в том, что один смотрит назад, а другой вперед. Если использовать прошлое для аппроксимации будущего, имеет смысл за-

мещать страницу, которая не использовалась в течение самого долгого времени. Такой подход называется least recently used алгоритм (LRU). Работа алгоритма проиллюстрирована на рис. рис. 4.2. Сравнивая рис. 4.1 b и 4.2, можно увидеть, что использование LRU алгоритма позволяет сократить количество страничных нарушений.

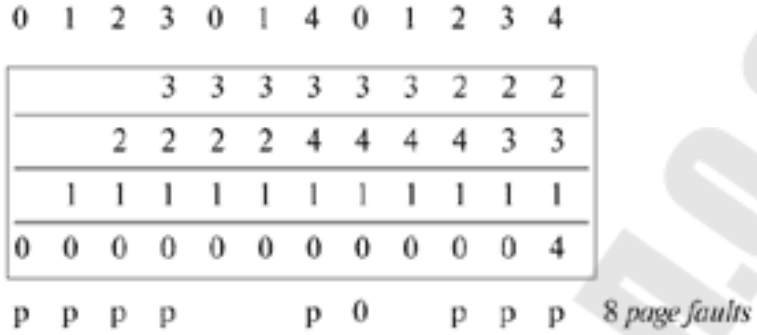

Рис. 4.2. Пример работы алгоритма LRU

LRU - хороший, но труднореализуемый алгоритм. Необходимо иметь связанный список всех страниц в памяти, в начале которого будут хранится недавно использованные страницы. Причем этот список должен обновляться при каждом обращении к памяти. Много времени нужно и на поиск страниц в таком списке.

Существует вариант реализации алгоритма LRU со специальным 64-битным указателем, который автоматически увеличивается на единицу после выполнения каждой инструкции, а в таблице страниц имеется соответствующее поле, в которое заносится значение указателя при каждой ссылке на страницу. При возникновении page fault выгружается страница с наименьшим значением этого поля.

Как оптимальный алгоритм, так и LRU не страдают от аномалии Билэди. Существует класс алгоритмов, для которых при одной и той же строке обращений множество страниц в памяти для n кадров всегда является подмножеством страниц для n+1 кадра. Эти алгоритмы не проявляют аномалии Билэди и называются стековыми (stack) алгоритмами.

4.8 Выталкивание редко используемой страницы. Алгоритм NFU

Поскольку большинство современных процессоров не предоставляют соответствующей аппаратной поддержки для реализации алго-

ритма LRU, хотелось бы иметь алгоритм, достаточно близкий к LRU, но не требующий специальной поддержки.

Программная реализация алгоритма, близкого к LRU, - алгоритм NFU (Not Frequently Used). Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени (а не после каждой инструкции) операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг обращения сбрасывает. Таким образом, кандидатом на освобождение оказывается страница с наименьшим значением счетчика, как страница, к которой реже всего обращались.

Главный недостаток алгоритма NFU состоит в том, что он ничего не забывает. Например, страница, к которой очень часто обращались в течение некоторого времени, а потом обращаться перестали, все равно не будет удалена из памяти, потому что ее счетчик содержит большую величину. Например, в многопроходных компиляторах страницы, которые активно использовались во время первого прохода, могут надолго сохранить большие значения счетчика, мешая загрузке полезных в дальнейшем страниц.

К счастью, возможна небольшая модификация алгоритма, которая позволяет ему "забывать". Достаточно, чтобы при каждом прерывании по времени содержимое счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения.

Другим, уже более устойчивым недостатком алгоритма является длительность процесса сканирования таблиц страниц.

4.9 Другие алгоритмы

Для полноты картины можно упомянуть еще несколько алгоритмов. Например, алгоритм Second-Chance - модификация алгоритма FIFO, которая позволяет избежать потери часто используемых страниц с помощью анализа флага обращений (бита ссылки) для самой старой страницы. Если флаг установлен, то страница, в отличие от алгоритма FIFO, не выталкивается, а ее флаг сбрасывается, и страница переносится в конец очереди. Если первоначально флаги обращений были установлены для всех страниц (на все страницы ссылались), алгоритм Second-Chance превращается в алгоритм FIFO. Данный алгоритм использовался в Multics и BSD Unix.

В компьютере Macintosh использован алгоритм NRU (Not Recently-Used), где страница-"жертва" выбирается на основе анализа битов модификации и ссылки. Интересные стратегии, основанные на буферизации страниц, реализованы в VAX/VMS и Mach.

Имеется также и много других алгоритмов замещения.

4.10 Управление количеством страниц, выделенным процессу. Модель рабочего множества

В стратегиях замещения, рассмотренных в предыдущем разделе, прослеживается предположение о том, что количество кадров, принадлежащих процессу, нельзя увеличить. Это приводит к необходимости выталкивания страницы. Рассмотрим более общий подход, базирующийся на концепции рабочего множества, сформулированной Деннингом. Итак, что делать, если в распоряжении процесса имеется недостаточное число кадров? Нужно ли его приостановить с освобождением всех кадров? Что следует понимать под достаточным количеством кадров?

4.11 Трешинг (Thrashing)

Хотя теоретически возможно уменьшить число кадров процесса до минимума, существует какое-то число активно используемых страниц, без которого процесс часто генерирует page faults. Высокая частота страничных нарушений называется трешинг (thrashing, иногда употребляется русский термин "пробуксовка", см. рис. 10.3). Процесс находится в состоянии трешинга, если при его работе больше времени уходит на подкачку страниц, нежели на выполнение команд. Такого рода критическая ситуация возникает вне зависимости от конкретных алгоритмов замещения.

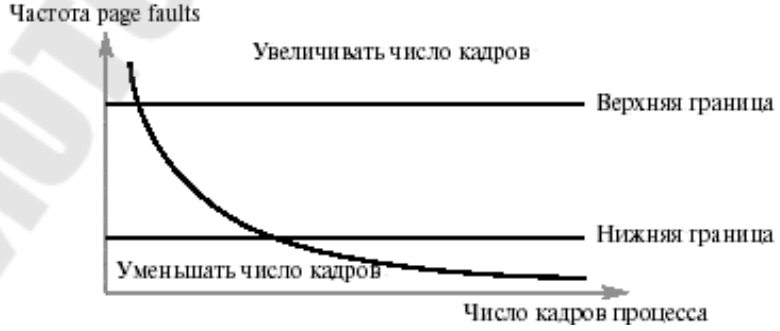

Рис. 4.3. Частота page faults в зависимости от количества кадров, выделенных процессу

Часто результатом трешинга является снижение производительности вычислительной системы. Один из нежелательных сценариев развития событий может выглядеть следующим образом. При глобальном алгоритме замещения процесс, которому не хватает кадров, начинает отбирать кадры у других процессов, которые в свою очередь начинают заниматься тем же. В результате все процессы попадают в очередь запросов к устройству вторичной памяти (находятся в состоянии ожидания), а очередь процессов в состоянии готовности пустеет. Загрузка процессора снижается. Операционная система реагирует на это увеличением степени мультипрограммирования, что приводит к еще большему трешингу и дальнейшему снижению загрузки процессора. Таким образом, пропускная способность системы падает из-за трешинга.

Эффект трешинга, возникающий при использовании глобальных алгоритмов, может быть ограничен за счет применения локальных алгоритмов замещения. При локальных алгоритмах замещения если даже один из процессов попал в трешинг, это не сказывается на других процессах. Однако он много времени проводит в очереди к устройству выгрузки, затрудняя подкачку страниц остальных процессов.

Критическая ситуация типа трешинга возникает вне зависимости от конкретных алгоритмов замещения. Единственным алгоритмом, теоретически гарантирующим отсутствие трешинга, является рассмотренный выше не реализуемый на практике оптимальный алгоритм.

Трешинг – это высокая частота страничных нарушений. Необходимо ее контролировать. Когда она высока, процесс нуждается в кадрах. Можно, устанавливая желаемую частоту page faults, регулировать размер процесса, добавляя или отнимая у него кадры. Может оказаться целесообразным выгрузить процесс целиком. Освободившиеся кадры выделяются другим процессам с высокой частотой page faults.

Для предотвращения трешинга требуется выделять процессу столько кадров, сколько ему нужно. Но как узнать, сколько ему нужно? Необходимо попытаться выяснить, как много кадров процесс реально использует. Для решения этой задачи Деннинг использовал модель рабочего множества, которая основана на применении принципа локальности.

4.12 Модель рабочего множества

Процессы начинают работать, не имея в памяти необходимых страниц. В результате при выполнении первой же машинной инструкции возникает page fault, требующий подкачки порции кода. Следующий page fault происходит при локализации глобальных переменных и еще один - при выделении памяти для стека. После того как процесс собрал большую часть необходимых ему страниц, page faults возникают редко. Таким образом, существует набор страниц (P1, P2, ... Pn), активно использующихся вместе, который позволяет процессу в момент времени t в течение некоторого периода T производительно работать, избегая большого количества page faults. Этот набор страниц называется рабочим множеством W(t,T) (working set) процесса. Число страниц в рабочем множестве определяется параметром T, является неубывающей функцией T и относительно невелико. Иногда T называют размером окна рабочего множества, через которое ведется наблюдение за процессом (рис. 4.4).

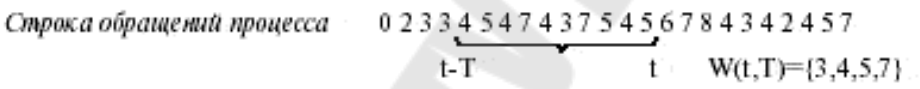

Рис. 4.4. Пример рабочего множества процесса

Легко написать тестовую программу, которая систематически работает с большим диапазоном адресов, но, к счастью, большинство реальных процессов не ведут себя подобным образом, а проявляют свойство локальности. В течение любой фазы вычислений процесс работает с небольшим количеством страниц.

Когда процесс выполняется, он двигается от одного рабочего множества к другому. Программа обычно состоит из нескольких рабочих множеств, которые могут перекрываться. Например, когда вызвана процедура, она определяет новое рабочее множество, состоящее из страниц, содержащих инструкции процедуры, ее локальные и глобальные переменные. После ее завершения процесс покидает это рабочее множество, но может вернуться к нему при новом вызове процедуры. Таким образом, рабочее множество определяется кодом и данными программы. Если процессу выделять меньше кадров, чем

ему требуется для поддержки рабочего множества, он будет находиться в состоянии трешинга.

Принцип локальности ссылок препятствует частым изменениям рабочих наборов процессов. Формально это можно выразить следующим образом. Если в период времени (t-T, t) программа обращалась к страницам W(t,T), то при надлежащем выборе T с большой вероятностью эта программа будет обращаться к тем же страницам в период времени (t, t+T). Другими словами, принцип локальности утверждает, что если не слишком далеко заглядывать в будущее, то можно достаточно точно его прогнозировать исходя из прошлого. Понятно, что с течением времени рабочий набор процесса может изменяться (как по составу страниц, так и по их числу).

Наиболее важное свойство рабочего множества - его размер. ОС должна выделить каждому процессу достаточное число кадров, чтобы поместилось его рабочее множество. Если кадры еще остались, то может быть инициирован другой процесс. Если рабочие множества процессов не помещаются в память и начинается трешинг, то один из процессов можно выгрузить на диск.

Решение о размещении процессов в памяти должно, следовательно, базироваться на размере его рабочего множества. Для впервые инициируемых процессов это решение может быть принято эвристически. Во время работы процесса система должна уметь определять: расширяет процесс свое рабочее множество или перемещается на новое рабочее множество. Если в состав атрибутов страницы включить время последнего использования ti (для страницы с номером i), то принадлежность i-й страницы к рабочему набору, определяемому параметром T в момент времени t будет выражаться неравенством: t-T < ti < t. Алгоритм выталкивания страниц WSClock, использующий информацию о рабочем наборе процесса.

Другой способ реализации данного подхода может быть основан на отслеживании количества страничных нарушений, вызываемых процессом. Если процесс часто генерирует page faults и память не слишком заполнена, то система может увеличить число выделенных ему кадров. Если же процесс не вызывает исключительных ситуаций в течение некоторого времени и уровень генерации ниже какого-то порога, то число кадров процесса может быть урезано. Этот способ регулирует лишь размер множества страниц, принадлежащих процессу, и должен быть дополнен какой-либо стратегией замещения страниц. Несмотря на то что система при этом может пробуксовывать в

моменты перехода от одного рабочего множества к другому, предлагаемое решение в состоянии обеспечить наилучшую производительность для каждого процесса, не требуя никакой дополнительной настройки системы.

4.13 Страничные демоны

Подсистема виртуальной памяти работает производительно при наличии резерва свободных страничных кадров. Алгоритмы, обеспечивающие поддержку системы в состоянии отсутствия трешинга, реализованы в составе фоновых процессов (их часто называют демонами или сервисами), которые периодически "просыпаются" и инспектируют состояние памяти. Если свободных кадров оказывается мало, они могут сменить стратегию замещения. Их задача - поддерживать систему в состоянии наилучшей производительности.

Примером такого рода процесса может быть фоновый процесс - сборщик страниц, реализующий облегченный вариант алгоритма откачки, основанный на использовании рабочего набора и применяемый во многих клонах ОС Unix. Данный демон производит откачку страниц, не входящих в рабочие наборы процессов. Он начинает активно работать, когда количество страниц в списке свободных страниц достигает установленного нижнего порога, и пытается выталкивать страницы в соответствии с собственной стратегией.

Но если возникает требование страницы в условиях, когда список свободных страниц пуст, то начинает работать механизм свопинга, поскольку простое отнятие страницы у любого процесса (включая тот, который затребовал бы страницу) потенциально вело бы к ситуации thrashing, и разрушало бы рабочий набор некоторого процесса. Любой процесс, затребовавший страницу не из своего текущего рабочего набора, становится в очередь на выгрузку в расчете на то, что после завершения выгрузки хотя бы одного из процессов свободной памяти уже может быть достаточно.

В ОС Windows 2000 аналогичную роль играет менеджер балансного набора (Working set manager), который вызывается раз в секунду или тогда, когда размер свободной памяти опускается ниже определенного предела, и отвечает за суммарную политику управления памятью и поддержку рабочих множеств.

4.14 Программная поддержка сегментной модели памяти процесса

Реализация функций операционной системы, связанных с поддержкой памяти, ведение таблиц страниц, трансляция адреса, обработка страничных ошибок, управление ассоциативной памятью и др. - тесно связана со структурами данных, обеспечивающими удобное представление адресного пространства процесса. Формат этих структур сильно зависит от аппаратуры и особенностей конкретной ОС.

Чаще всего виртуальная память процесса ОС разбивается на сегменты пяти типов: кода программы, данных, стека, разделяемый и сегмент файлов, отображаемых в память (рис. 4.5).

Сегмент программного кода содержит только команды. Сегмент программного кода не модифицируется в ходе выполнения процесса, обычно страницы данного сегмента имеют атрибут read-only. Следствием этого является возможность использования одного экземпляра кода для разных процессов.

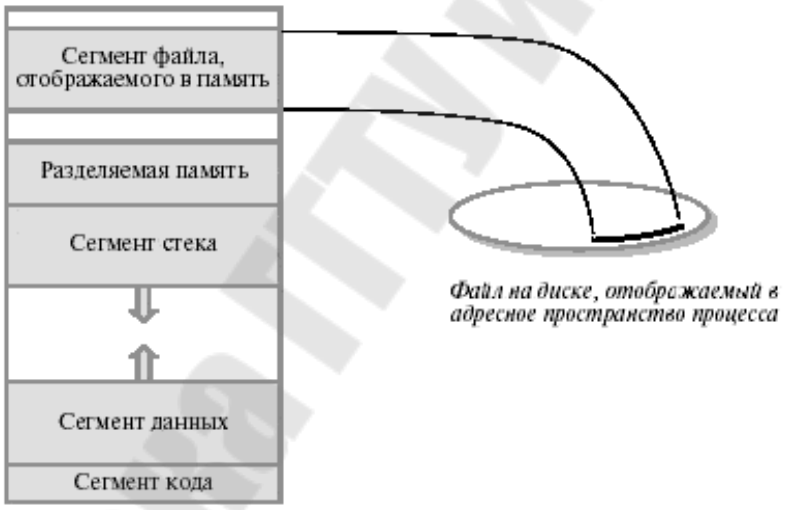

Рис. 4.5. Образ процесса в памяти

Сегмент данных, содержащий переменные программы и сегмент стека, содержащий автоматические переменные, могут динамически менять свой размер (обычно данные в сторону увеличения адресов, а стек - в сторону уменьшения) и содержимое, должны быть доступны по чтению и записи и являются приватными сегментами процесса.

С целью обобществления памяти между несколькими процессами создаются разделяемые сегменты, допускающие доступ по чтению и записи. Вариантом разделяемого сегмента может быть сегмент файла, отображаемого в память. Специфика таких сегментов состоит в том,

что из них откачка осуществляется не в системную область выгрузки, а непосредственно в отображаемый файл. Реализация разделяемых сегментов основана на том, что логические страницы различных процессов связываются с одними и теми же страничными кадрами.

Сегменты представляют собой непрерывные области (в Linux они так и называются – области) в виртуальном адресном пространстве процесса, выровненные по границам страниц. Каждая область состоит из набора страниц с одним и тем же режимом защиты. Между областями в виртуальном пространстве могут быть свободные участки. Естественно, что подобные объекты описаны соответствующими структурами (см., например, структуры mm_struct и vm_area_struct в Linux).

Часть работы по организации сегментов может происходить с участием программиста. Особенно это заметно при низкоуровневом программировании. В частности, отдельные области памяти могут быть поименованы и использоваться для обмена данными между процессами. Два процесса могут общаться через разделяемую область памяти при условии, что им известно ее имя (пароль). Обычно это делается при помощи специальных вызовов (например, map и unmap), входящих в состав интерфейса виртуальной памяти.

Загрузка исполняемого файла (системный вызов exec) осуществляется обычно через отображение (mapping) его частей (кода, данных) в соответствующие сегменты адресного пространства процесса. Например, сегмент кода является сегментом отображаемого в память файла, содержащего исполняемую программу. При попытке выполнить первую же инструкцию система обнаруживает, что нужной части кода в памяти нет, генерирует page fault и подкачивает эту часть кода с диска. Далее процедура повторяется до тех пор, пока вся программа не окажется в оперативной памяти.

Как уже говорилось, размер сегмента данных динамически меняется. Рассмотрим, как организована поддержка сегментов данных в Unix. Пользователь, запрашивая (библиотечные вызовы malloc, new) или освобождая (free, delete) память для динамических данных, фактически изменяет границу выделенной процессу памяти через системный вызов brk (от слова break), который модифицирует значение переменной brk из структуры данных процесса. В результате происходит выделение физической памяти, граница brk смещается в сторону увеличения виртуальных адресов, а соответствующие строки таблиц страниц получают осмысленные значения. При помощи того же вызова brk пользователь может уменьшить размер сегмента данных.

На практике освобожденная пользователем виртуальная память (библиотечные вызовы free, delete) системе не возвращается. На это есть две причины. Во-первых, для уменьшения размеров сегмента данных необходимо организовать его уплотнение или "сборку мусора". А во-вторых, незанятые внутри сегмента данных области естественным образом будут вытолкнуты из оперативной памяти вследствие того, что к ним не будет обращений. Ведение списков занятых и свободных областей памяти в сегменте данных пользователя осуществляется на уровне системных библиотек.

4.15 Отдельные аспекты функционирования менеджера памяти

Корректная работа менеджера памяти помимо принципиальных вопросов, связанных с выбором абстрактной модели виртуальной памяти и ее аппаратной поддержкой, обеспечивается также множеством нюансов и мелких деталей. В качестве примера такого рода компонента рассмотрим более подробно локализацию страниц в памяти, которая применяется в тех случаях, когда поддержка страничной системы приводит к необходимости разрешить определенным страницам, хранящим буферы ввода-вывода, другие важные данные и код, быть блокированными в памяти.

Рассмотрим случай, когда система виртуальной памяти может вступить в конфликт с подсистемой ввода-вывода. Например, процесс может запросить ввод в буфер и ожидать его завершения. Управление передастся другому процессу, который может вызвать page fault и, с отличной от нуля вероятностью, спровоцировать выгрузку той страницы, куда должен быть осуществлен ввод первым процессом. Подобные ситуации нуждаются в дополнительном контроле, особенно если ввод-вывод реализован с использованием механизма прямого доступа к памяти (DMA).

Одно из решений данной проблемы - вводить данные в не вытесняемый буфер в пространстве ядра, а затем копировать их в пользовательское пространство.

Второе решение - локализовать страницы в памяти, используя специальный бит локализации, входящий в состав атрибутов страницы. Локализованная страница замещению не подлежит. Бит локализации сбрасывается после завершения операции ввода-вывода. Другое использование бита локализации может иметь место и при нормаль-

ном замещении страниц. Рассмотрим следующую цепь событий. Низкоприоритетный процесс после длительного ожидания получил в свое распоряжение процессор и подкачал с диска нужную ему страницу. Если он сразу после этого будет вытеснен высокоприоритетным процессом, последний может легко заместить вновь подкачанную страницу низкоприоритетного, так как на нее не было ссылок. Имеет смысл вновь загруженные страницы помечать битом локализации до первой ссылки, иначе низкоприоритетный процесс так и не начнет работать.

Использование бита локализации может быть опасным, если забыть его отключить. Если такая ситуация имеет место, страница становится неиспользуемой. SunOS разрешает использование данного бита в качестве подсказки, которую можно игнорировать, когда пул свободных кадров становится слишком маленьким.

Другим важным применением локализации является ее использование в системах мягкого реального времени. Рассмотрим процесс или нить реального времени. Вообще говоря, виртуальная память - антитеза вычислений реального времени, так как дает непредсказуемые задержки при подкачке страниц. Поэтому системы реального времени почти не используют виртуальную память. ОС Solaris поддерживает как реальное время, так и разделение времени. Для решения проблемы page faults, Solaris разрешает процессам сообщать системе, какие страницы важны для процесса, и локализовать их в памяти. В результате возможно выполнение процесса, реализующего задачу реального времени, содержащего локализованные страницы, где временные задержки страничной системы будут минимизированы.

Помимо системы локализации страниц, есть и другие интересные проблемы, возникающие в процессе управления памятью. Так, например, бывает непросто осуществить повторное выполнение инструкции, вызвавшей page fault. Представляют интерес и алгоритмы отложенного выделения памяти (копирование при записи и др.).

5 Внешняя память

5.1 Файловая система

Файловая система - это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во внешней памяти, и обеспечить пользователю удобный интерфейс при работе с такими данными. Организовать хранение информации на магнитном диске непросто. Это требует, например, хорошего знания устройства контроллера диска, особенностей работы с его регистрами. Непосредственное взаимодействие с диском - прерогатива компонента системы ввода-вывода ОС, называемого драйвером диска. Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой, была придумана ясная абстрактная модель файловой системы. Операции записи или чтения файла концептуально проще, чем низкоуровневые операции работы с устройствами. Основная идея использования внешней памяти состоит в следующем. ОС делит память на блоки фиксированного размера, например, 4096 байт. Файл, обычно представляющий собой неструктурированную последовательность однобайтовых записей, хранится в виде последовательности блоков (не обязательно смежных); каждый блок хранит целое число записей. В некоторых ОС (MS-DOS) адреса блоков, содержащих данные файла, могут быть организованы в связный список и вынесены в отдельную таблицу в памяти. В других ОС (Unix) адреса блоков данных файла хранятся в отдельном блоке внешней памяти (так называемом индексе или индексном узле). Этот прием, называемый индексацией, является наиболее распространенным для приложений, требующих произвольного доступа к записям файлов. Индекс файла состоит из списка элементов, каждый из которых содержит номер блока в файле и сведения о местоположении данного блока. Считывание очередного байта осуществляется с так называемой текущей позиции, которая характеризуется смещением от начала файла. Зная размер блока, легко вычислить номер блока, содержащего текущую позицию. Адрес же нужного блока диска можно затем извлечь из индекса файла. Базовой операцией, выполняемой по отношению к файлу, является чтение блока с диска и перенос его в буфер, находящийся в основной памяти.

Файловая система позволяет при помощи системы справочников (каталогов, директорий) связать уникальное имя файла с блоками

вторичной памяти, содержащими данные файла. Иерархическая структура каталогов, используемая для управления файлами, может служить другим примером индексной структуры. В этом случае каталоги или папки играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги. С этой точки зрения вся файловая система компьютера представляет собой большой индексированный файл. Помимо собственно файлов и структур данных, используемых для управления файлами (каталоги, дескрипторы файлов, различные таблицы распределения внешней памяти), понятие "файловая система" включает программные средства, реализующие различные операции над файлами.

5.2. Основные функции файловой систем

1) Идентификация файлов.
2) Связывание имени файла с выделенным ему пространством внешней памяти.
3) Распределение внешней памяти между файлами. Для работы с конкретным файлом пользователю не требуется иметь информацию о местоположении этого файла на внешнем носителе информации. Например, для того чтобы загрузить документ в редактор с жесткого диска, нам не нужно знать, на какой стороне какого магнитного диска, на каком цилиндре и в каком секторе находится данный документ.
4) Обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера.
5) Обеспечение защиты от несанкционированного доступа.
6) Обеспечение совместного доступа к файлам, так чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа.
7) Обеспечение высокой производительности.

Иногда говорят, что файл - это поименованный набор связанной информации, записанной во вторичную память. Для большинства пользователей файловая система - наиболее видимая часть ОС. Она предоставляет механизм для онлайнового хранения и доступа как к данным, так и к программам для всех пользователей системы. С точки зрения пользователя, файл - единица внешней памяти, то есть данные, записанные на диск, должны быть в составе какого-нибудь файла.

Важный аспект организации файловой системы - учет стоимости операций взаимодействия с вторичной памятью. Процесс считывания блока диска состоит из позиционирования считывающей головки над дорожкой, содержащей требуемый блок, ожидания, пока требуемый блок сделает оборот и окажется под головкой, и собственно считывания блока. Для этого требуется значительное время (десятки миллисекунд). В современных компьютерах обращение к диску осуществляется примерно в 100 000 раз медленнее, чем обращение к оперативной памяти. Таким образом, критерием вычислительной сложности алгоритмов, работающих с внешней памятью, является количество обращений к диску.

5.3. Атрибуты файлов

Имена файлов.

Файлы представляют собой абстрактные объекты. Их задача - хранить информацию, скрывая от пользователя детали работы с устройствами. Когда процесс создает файл, он дает ему имя. После завершения процесса файл продолжает существовать и через свое имя может быть доступен другим процессам. Правила именования файлов зависят от ОС. Многие ОС поддерживают имена из двух частей (имя+расширение), например progr.c (файл, содержащий текст программы на языке Си) или autoexec.bat (файл, содержащий команды интерпретатора командного языка). Тип расширения файла позволяет ОС организовать работу с ним различных прикладных программ в соответствии с заранее оговоренными соглашениями. Обычно ОС накладывают некоторые ограничения, как на используемые в имени символы, так и на длину имени файла. В соответствии со стандартом POSIX, популярные ОС оперируют удобными для пользователя длинными именами (до 255 символов).

Типы файлов

Важный аспект организации файловой системы и ОС - следует ли поддерживать и распознавать типы файлов. Если да, то это может помочь правильному функционированию ОС, например не допустить вывода на принтер бинарного файла.

Основные типы файлов: регулярные (обычные) файлы и директории (справочники, каталоги). Обычные файлы содержат пользовательскую информацию. Директории - системные файлы, поддержи-

вающие структуру файловой системы. В каталоге содержится перечень входящих в него файлов и устанавливается соответствие между файлами и их характеристиками (атрибутами). Мы будем рассматривать директории ниже.

Напомним, что хотя внутри подсистемы управления файлами обычный файл представляется в виде набора блоков внешней памяти, для пользователей обеспечивается представление файла в виде линейной последовательности байтов. Такое представление позволяет использовать абстракцию файла при работе с внешними устройствами, при организации межпроцессных взаимодействий и т. д. Так, например, клавиатура обычно рассматривается как текстовый файл, из которого компьютер получает данные в символьном формате. Поэтому иногда к файлам приписывают другие объекты ОС, например специальные символьные файлы и специальные блочные файлы, именованные каналы и сокеты, имеющие файловый интерфейс.

Обычные (или регулярные) файлы реально представляют собой набор блоков (возможно, пустой) на устройстве внешней памяти, на котором поддерживается файловая система. Такие файлы могут содержать как текстовую информацию (обычно в формате ASCII), так и произвольную двоичную (бинарную) информацию. Текстовые файлы содержат символьные строки, которые можно распечатать, увидеть на экране или редактировать обычным текстовым редактором.

Другой тип файлов - нетекстовые, или бинарные, файлы. Обычно они имеют некоторую внутреннюю структуру. Например, исполняемый файл в ОС Unix имеет пять секций: заголовок, текст, данные, биты реаллокации и символьную таблицу. ОС выполняет файл, только если он имеет нужный формат. Другим примером бинарного файла может быть архивный файл. Типизация файлов не слишком строгая.

Обычно прикладные программы, работающие с файлами, распознают тип файла по его имени в соответствии с общепринятыми соглашениями. Например, файлы с расширениями .c, .pas, .txt - ASCII-файлы, файлы с расширениями .exe - выполнимые, файлы с расширениями .obj, .zip - бинарные и т. д.

Атрибуты файлов

Кроме имени ОС часто связывают с каждым файлом и другую информацию, например дату модификации, размер и т. д. Эти другие характеристики файлов называются атрибутами. Список атрибутов в разных ОС может варьироваться. Обычно он содержит следующие элементы: основную информацию (имя, тип файла), адресную ин-

формацию (устройство, начальный адрес, размер), информацию об управлении доступом (владелец, допустимые операции) и информацию об использовании (даты создания, последнего чтения, модификации и др.). Список атрибутов обычно хранится в структуре директорий или других структурах, обеспечивающих доступ к данным файла.

Организация файлов и доступ к ним

Программист воспринимает файл в виде набора однородных записей. Запись - это наименьший элемент данных, который может быть обработан как единое целое прикладной программой при обмене с внешним устройством. Причем в большинстве ОС размер записи равен одному байту. В то время как приложения оперируют записями, физический обмен с устройством осуществляется большими единицами (обычно блоками). Поэтому записи объединяются в блоки для вывода и разблокируются - для ввода. Вопросы распределения блоков внешней памяти между файлами рассматриваются в следующей лекции. ОС поддерживают несколько вариантов структуризации файлов.

5.4. Последовательный файл

Простейший вариант - так называемый последовательный файл. То есть файл является последовательностью записей. Поскольку записи, как правило, однобайтовые, файл представляет собой неструктурированную последовательность байтов.

Обработка подобных файлов предполагает последовательное чтение записей от начала файла, причем конкретная запись определяется ее положением в файле. Такой способ доступа называется последовательным (модель ленты). Если в качестве носителя файла используется магнитная лента, то так и делается. Текущая позиция считывания может быть возвращена к началу файла (rewind).

5.5. Файл прямого доступа

В реальной практике файлы хранятся на устройствах прямого (random) доступа, например на дисках, поэтому содержимое файла может быть разбросано по разным блокам диска, которые можно считывать в произвольном порядке. Причем номер блока однозначно определяется позицией внутри файла.

Здесь имеется в виду относительный номер, специфицирующий данный блок среди блоков диска, принадлежащих файлу. Естественно, что в этом случае для доступа к середине файла просмотр всего файла с самого начала не обязателен. Для специфицирования места, с которого надо начинать чтение, используются два способа: с начала или с текущей позиции, которую дает операция seek. Файл, байты которого могут быть считаны в произвольном порядке, называется файлом прямого доступа.

Таким образом, файл, состоящий из однобайтовых записей на устройстве прямого доступа, - наиболее распространенный способ организации файла. Базовыми операциями для такого рода файлов являются считывание или запись символа в текущую позицию. В большинстве языков высокого уровня предусмотрены операторы посимвольной пересылки данных в файл или из него.

Подобную логическую структуру имеют файлы во многих файловых системах, например в файловых системах ОС Unix и MS-DOS. ОС не осуществляет никакой интерпретации содержимого файла. Эта схема обеспечивает максимальную гибкость и универсальность. С помощью базовых системных вызовов (или функций библиотеки ввода/вывода) пользователи могут как угодно структурировать файлы. В частности, многие СУБД хранят свои базы данных в обычных файлах.

5.6. Другие формы организации файлов

Известны как другие формы организации файла, так и другие способы доступа к ним, которые использовались в ранних ОС, а также применяются сегодня в больших мэйнфреймах (mainframe), ориентированных на коммерческую обработку данных.

Первый шаг в структурировании - хранение файла в виде последовательности записей фиксированной длины, каждая из которых имеет внутреннюю структуру. Операция чтения производится над записью, а операция записи переписывает или добавляет запись целиком. Ранее использовались записи по 80 байт (это соответствовало числу позиций в перфокарте) или по 132 символа (ширина принтера). В ОС CP/M файлы были последовательностями 128-символьных записей. С введением CRT-терминалов данная идея утратила популярность.

Другой способ представления файлов - последовательность записей переменной длины, каждая из которых содержит ключевое поле в фиксированной позиции внутри записи (см. рис. 5.1). Базисная операция в данном случае - считать запись с каким-либо значением ключа. Записи могут располагаться в файле последовательно (например, отсортированные по значению ключевого поля) или в более сложном порядке. Метод доступа по значению ключевого поля к записям последовательного файла называется индексно-последовательным.

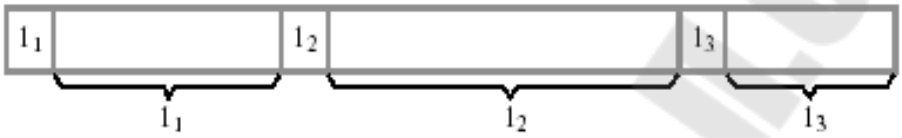

Рис. 5.1. Файл как последовательность записей переменной длины

В некоторых системах ускорение доступа к файлу обеспечивается конструированием индекса файла. Индекс обычно хранится на том же устройстве, что и сам файл, и состоит из списка элементов, каждый из которых содержит идентификатор записи, за которым следует указание о местоположении данной записи. Для поиска записи вначале происходит обращение к индексу, где находится указатель на нужную запись. Такие файлы называются индексированными, а метод доступа к ним - доступ с использованием индекса.

Рис. 5.2. Пример организации индекса для последовательного файла

Предположим, у нас имеется большой несортированный файл, содержащий разнообразные сведения о студентах, состоящие из записей с несколькими полями, и возникает задача организации быстрого поиска по одному из полей, например по фамилии студента. Рис. 5.2 иллюстрирует решение данной проблемы - организацию метода дос-

тупа к файлу с использованием индекса. Следует отметить, что почти всегда главным фактором увеличения скорости доступа является избыточность данных.

Способ выделения дискового пространства при помощи индексных узлов, применяемый в ряде ОС (Unix и некоторых других, см. следующую лекцию), может служить другим примером организации индекса. В этом случае ОС использует древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков файла. Для больших файлов индекс может быть слишком велик. В этом случае создают индекс для индексного файла (блоки промежуточного уровня или блоки косвенной адресации).

5.7. Операции над файлами

Операционная система должна предоставить в распоряжение пользователя набор операций для работы с файлами, реализованных через системные вызовы. Чаще всего при работе с файлом пользователь выполняет не одну, а несколько операций. Во-первых, нужно найти данные файла и его атрибуты по символьному имени, во-вторых, считать необходимые атрибуты файла в отведенную область оперативной памяти и проанализировать права пользователя на выполнение требуемой операции. Затем следует выполнить операцию, после чего освободить занимаемую данными файла область памяти.

Рассмотрим в качестве примера основные файловые операции ОС Unix:

Создание файла, не содержащего данных. Смысл данного вызова - объявить, что файл существует, и присвоить ему ряд атрибутов. При этом выделяется место для файла на диске и вносится запись в каталог.

Удаление файла и освобождение занимаемого им дискового пространства.

Открытие файла. Перед использованием файла процесс должен его открыть. Цель данного системного вызова - разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в оперативную память список адресов блоков файла для быстрого доступа к его данным. Открытие файла является процедурой создания дескриптора или управляющего блока файла. Деск-

риптор (описатель) файла хранит всю информацию о нем. Иногда, в соответствии с парадигмой, принятой в языках программирования, под дескриптором понимается альтернативное имя файла или указатель на описание файла в таблице открытых файлов, используемый при последующей работе с файлом . Например, на языке Си операция открытия файла fd=open(pathname,flags,modes); возвращает дескриптор fd, который может быть задействован при выполнении операций чтения (read(fd,buffer,count);) или записи.

Закрытие файла. Если работа с файлом завершена, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.

Позиционирование. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, то есть задать текущую позицию.

Чтение данных из файла. Обычно это делается с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти.

Запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

Есть и другие операции, например переименование файла, получение атрибутов файла и т. д.

Существует два способа выполнить последовательность действий над файлами. В первом случае для каждой операции выполняются как универсальные, так и уникальные действия (схема stateless). Например, последовательность операций может быть такой: open, read1, close, ... open, read2, close, ... open, read3, close.

Альтернативный способ - это когда универсальные действия выполняются в начале и в конце последовательности операций, а для каждой промежуточной операции выполняются только уникальные действия. В этом случае последовательность вышеприведенных операций будет выглядеть так: open, read1, ... read2, ... read3, close.

Большинство ОС использует второй способ, более экономичный и быстрый. Первый способ более устойчив к сбоям, поскольку результаты каждой операции становятся независимыми от результатов предыдущей операции; поэтому он иногда применяется в распределенных файловых системах (например, Sun NFS).

5.8. Директории.

Количество файлов на компьютере может быть большим. Отдельные системы хранят тысячи файлов, занимающие сотни гигабайтов дискового пространства. Эффективное управление этими данными подразумевает наличие в них четкой логической структуры. Все современные файловые системы поддерживают многоуровневое именование файлов за счет наличия во внешней памяти дополнительных файлов со специальной структурой - каталогов (или директорий). Каждый каталог содержит список каталогов и/или файлов, содержащихся в данном каталоге. Каталоги имеют один и тот же внутренний формат, где каждому файлу соответствует одна запись в файле директории (см., например, рис.5.3).

Число директорий зависит от системы. В ранних ОС имелась только одна корневая директория, затем появились директории для пользователей (по одной директории на пользователя). В современных ОС используется произвольная структура дерева директорий.

Имя файла (каталога)	Тип файла (обычный или каталог)	
Anti	K	атрибуты
Games	K	атрибуты
Autoexec.bat	O	атрибуты
mouse.com	O	атрибуты

Рис. 5.3. Директории

Таким образом, файлы на диске образуют иерархическую древовидную структуру (см. рис. 5.4).

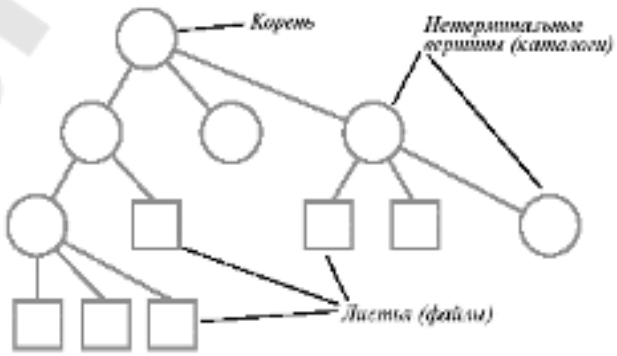

Рис. 5.4. Древовидная структура файловой системы

Существует несколько эквивалентных способов изображения дерева. Структура перевернутого дерева, приведенного на рис. 5.4, наиболее распространена. Верхнюю вершину называют корнем. Если элемент дерева не может иметь потомков, он называется терминальной вершиной или листом (в данном случае является файлом). Нелистовые вершины - справочники или каталоги содержат списки листовых и нелистовых вершин. Путь от корня к файлу однозначно определяет файл. Подобные древовидные структуры являются графами, не имеющими циклов. Можно считать, что ребра графа направлены вниз, а корень - вершина, не имеющая входящих ребер. Как мы увидим в следующей лекции, связывание файлов, которое практикуется в ряде операционных систем, приводит к образованию циклов в графе.

Внутри одного каталога имена листовых файлов уникальны. Имена файлов, находящихся в разных каталогах, могут совпадать. Для того чтобы однозначно определить файл по его имени (избежать коллизии имен), принято именовать файл так называемым абсолютным или полным именем (pathname), состоящим из списка имен вложенных каталогов, по которому можно найти путь от корня к файлу плюс имя файла в каталоге, непосредственно содержащем данный файл. То есть полное имя включает цепочку имен - путь к файлу, например /usr/games/doom. Такие имена уникальны. Компоненты пути разделяют различными символами: "/" (слэш) в Unix или обратными слэшем в MS-DOS (в Multics - ">"). Таким образом, использование древовидных каталогов минимизирует сложность назначения уникальных имен.

Указывать полное имя не всегда удобно, поэтому применяют другой способ задания имени - относительный путь к файлу. Он использует концепцию рабочей или текущей директории, которая обычно входит в состав атрибутов процесса, работающего с данным файлом. Тогда на файлы в такой директории можно ссылаться только по имени, при этом поиск файла будет осуществляться в рабочем каталоге. Это удобнее, но, по существу, то же самое, что и абсолютная форма.

Для получения доступа к файлу и локализации его блоков система должна выполнить навигацию по каталогам. Рассмотрим для примера путь /usr/linux/progr.c. Алгоритм одинаков для всех иерархических систем. Сначала в фиксированном месте на диске находится корневая директория. Затем находится компонент пути usr, т. е. в корневой директории ищется файл /usr. Исследуя этот файл, система

понимает, что данный файл является каталогом, и блоки его данных рассматривает как список файлов и ищет следующий компонент linux в нем. Из строки для linux находится файл, соответствующий компоненту usr/linux/. Затем находится компонент progr.c, который открывается, заносится в таблицу открытых файлов и сохраняется в ней до закрытия файла.

Отклонение от типовой обработки компонентов pathname может возникнуть в том случае, когда этот компонент является не обычным каталогом с соответствующим ему индексным узлом и списком файлов, а служит точкой связывания (принято говорить "точкой монтирования") двух файловых архивов. Этот случай рассмотрен в следующей лекции.

Многие прикладные программы работают с файлами, находящимися в текущей директории, не указывая явным образом ее имени. Это дает пользователю возможность произвольным образом именовать каталоги, содержащие различные программные пакеты. Для реализации этой возможности в большинстве ОС, поддерживающих иерархическую структуру директорий, используется обозначение "." - для текущей директории и ".." - для родительской.

5.9. Разделы диска.

Задание пути к файлу в файловых системах некоторых ОС отличается тем, с чего начинается эта цепочка имен. В современных ОС принято разбивать диски на логические диски (это низкоуровневая операция), иногда называемые разделами (partitions). Бывает, что, наоборот, объединяют несколько физических дисков в один логический диск (например, это можно сделать в ОС Windows NT). Поэтому в дальнейшем изложении мы будем игнорировать проблему физического выделения пространства для файлов и считать, что каждый раздел представляет собой отдельный (виртуальный) диск. Диск содержит иерархическую древовидную структуру, состоящую из набора файлов, каждый из которых является хранилищем данных пользователя, и каталогов или директорий (то есть файлов, которые содержат перечень других файлов, входящих в состав каталога), необходимых для хранения информации о файлах системы.

В некоторых системах управления файлами требуется, чтобы каждый архив файлов целиком располагался на одном диске (разделе диска). В этом случае полное имя файла начинается с имени дисково-

го устройства, на котором установлен соответствующий диск (буквы диска). Например, c:\util\nu\ndd.exe. Такой способ именования используется в файловых системах DEC и Microsoft.

В других системах (Multics) вся совокупность файлов и каталогов представляет собой единое дерево. Сама система, выполняя поиск файлов по имени, начиная с корня, требовала установки необходимых дисков.

В ОС Unix предполагается наличие нескольких архивов файлов, каждый на своем разделе, один из которых считается корневым. После запуска системы можно "смонтировать" корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему.

Технически это осуществляется с помощью создания в корневой файловой системе специальных пустых каталогов (см. также следующую лекцию). Специальный системный вызов mount ОС Unix позволяет подключить к одному из этих пустых каталогов корневой каталог указанного архива файлов. После монтирования общей файловой системы именование файлов производится так же, как если бы она с самого начала была централизованной. Задачей ОС является беспрепятственный проход точки монтирования при получении доступа к файлу по цепочке имен. Если учесть, что обычно монтирование файловой системы производится при загрузке системы, пользователи ОС Unix обычно и не задумываются о происхождении общей файловой системы.

5.10 Операции над директориями

Как и в случае с файлами, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы. Несмотря на то что директории - это файлы, логика работы с ними отличается от логики работы с обычными файлами и определяется природой этих объектов, предназначенных для поддержки структуры файлового архива. Совокупность системных вызовов для управления директориями зависит от особенностей конкретной ОС. Напомним, что операции над каталогами являются прерогативой ОС, то есть пользователь не может, например, выполнить запись в каталог начиная с текущей позиции. Рассмотрим в качестве примера некоторые системные вызовы, необходимые для работы с каталогами:

- Создание директории. Вновь созданная директория включает записи с именами '.' и '..', однако считается пустой.
- Удаление директории. Удалена может быть только пустая директория.
- Открытие директории для последующего чтения. Например, чтобы перечислить файлы, входящие в директорию, процесс должен открыть директорию и считать имена всех файлов, которые она включает.
- Закрытие директории после ее чтения для освобождения места во внутренних системных таблицах.
- Поиск. Данный системный вызов возвращает содержимое текущей записи в открытой директории. Вообще говоря, для этих целей может использоваться системный вызов Read, но в этом случае от программиста потребуется знание внутренней структуры директории.
- Получение списка файлов в каталоге.
- Переименование. Имена директорий можно менять, как и имена файлов.
- Создание файла. При создании нового файла необходимо добавить в каталог соответствующий элемент.
- Удаление файла. Удаление из каталога соответствующего элемента. Если удаляемый файл присутствует только в одной директории, то он вообще удаляется из файловой системы, в противном случае система ограничивается только удалением специфицируемой записи.

Очевидно, что создание и удаление файлов предполагает также выполнение соответствующих файловых операций. Имеется еще ряд других системных вызовов, например связанных с защитой информации.

5.11. Защита файлов

Информация в компьютерной системе должна быть защищена как от физического разрушения (reliability), так и от несанкционированного доступа (protection).

Наличие в системе многих пользователей предполагает организацию контролируемого доступа к файлам. Выполнение любой операции над файлом должно быть разрешено только в случае наличия у пользователя соответствующих привилегий. Обычно контролируются

следующие операции: чтение, запись и выполнение. Другие операции, например копирование файлов или их переименование, также могут контролироваться. Однако они чаще реализуются через перечисленные. Так, операцию копирования файлов можно представить как операцию чтения и последующую операцию записи.

Наиболее общий подход к защите файлов от несанкционированного использования - сделать доступ зависящим от идентификатора пользователя, то есть связать с каждым файлом или директорией список прав доступа (access control list), где перечислены имена пользователей и типы разрешенных для них способов доступа к файлу. Любой запрос на выполнение операции сверяется с таким списком. Основная проблема реализации данного способа - список может быть длинным. Чтобы разрешить всем пользователям читать файл, необходимо всех их внести в список. У такой техники есть два нежелательных следствия:

1) конструирование подобного списка может оказаться сложной задачей, особенно если мы не знаем заранее пользователей системы;
2) запись в директории должна иметь переменный размер (включать список потенциальных пользователей).

Для решения этих проблем создают классификации пользователей, например, в ОС Unix все пользователи разделены на три группы.

1) Владелец (Owner).
2) Группа (Group). Набор пользователей, разделяющих файл и нуждающихся в типовом способе доступа к нему.
3) Остальные (Univers).

Это позволяет реализовать конденсированную версию списка прав доступа. В рамках такой ограниченной классификации задаются только три поля (по одному для каждой группы) для каждой контролируемой операции. В итоге в Unix операции чтения, записи и исполнения контролируются при помощи 9 бит (rwxrwxrwx).

5.12. Общая структура файловой системы

Нижний уровень - оборудование. Это в первую очередь магнитные диски с подвижными головками - основные устройства внешней памяти, представляющие собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнит-

ных головок. Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр магнитного диска. Цилиндры делятся на дорожки (треки), а каждая дорожка размечается на одно и то же количество блоков (секторов) таким образом, что в каждый блок можно записать по максимуму одно и то же число байтов. Следовательно, для обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока. Таким образом, диски могут быть разбиты на блоки фиксированного размера и можно непосредственно получить доступ к любому блоку (организовать прямой доступ к файлам).

Непосредственно с устройствами (дисками) взаимодействует часть ОС, называемая ***системой ввода-вывода***. Система ввода-вывода предоставляет в распоряжение более высокоуровневого компонента ОС - файловой системы - используемое дисковое пространство в виде непрерывной последовательности блоков фиксированного размера. Система ввода-вывода имеет дело с физическими блоками диска, которые характеризуются адресом, например диск 2, цилиндр 75, сектор 11. Файловая система имеет дело с логическими блоками, каждый из которых имеет номер (от 0 или 1 до N). Размер логических блоков файла совпадает или является кратным размеру физического блока диска и может быть задан равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

В структуре системы управления файлами можно выделить базисную подсистему, которая отвечает за выделение дискового пространства конкретным файлам, и более высокоуровневую логическую подсистему, которая использует структуру дерева директорий для предоставления модулю базисной подсистемы необходимой ей информации, исходя из символического имени файла. Она также ответственна за авторизацию доступа к файлам.

Стандартный запрос на открытие (open) или создание (create) файла поступает от прикладной программы к логической подсистеме. Логическая подсистема, используя структуру директорий, проверяет права доступа и вызывает базовую подсистему для получения доступа к блокам файла. После этого файл считается открытым, он содержится в таблице открытых файлов, и прикладная программа получает в свое распоряжение дескриптор (или handle в системах Microsoft) это-

го файла. Дескриптор файла является ссылкой на файл в таблице открытых файлов и используется в запросах прикладной программы на чтение-запись из этого файла. Запись в таблице открытых файлов указывает через систему выделения блоков диска на блоки данного файла. Если к моменту открытия файл уже используется другим процессом, то есть содержится в таблице открытых файлов, то после проверки прав доступа к файлу может быть организован совместный доступ. При этом новому процессу также возвращается дескриптор - ссылка на файл в таблице открытых файлов. Далее в тексте подробно проанализирована работа наиболее важных системных вызовов.

5.13. Управление внешней памятью

Прежде чем описывать структуру данных файловой системы на диске, необходимо рассмотреть алгоритмы выделения дискового пространства и способы учета свободной и занятой дисковой памяти. Эти задачи связаны между собой. Ключевым, безусловно, является вопрос, какой тип структур используется для учета отдельных блоков файла, то есть способ связывания файлов с блоками диска. В ОС используется несколько методов выделения файлу дискового пространства. Для каждого из методов запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому можно найти все блоки данного файла.

Рис. 5.5. Блок-схема файловой системы

Ключевым, безусловно, является вопрос, какой тип структур используется для учета отдельных блоков файла, то есть способ связывания файлов с блоками диска. В ОС используется несколько методов выделения файлу дискового пространства. Для каждого из методов запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому можно найти все блоки данного файла.

Простейший способ - хранить каждый файл как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки b+1, b+2, ... b+n-1. Эта схема имеет два преимущества. Во-первых, ее легко реализовать, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок. Во-вторых, она обеспечивает хорошую производительность, так как целый файл может быть считан за одну дисковую операцию. Непрерывное выделение используется в ОС IBM/CMS, в ОС RSX-11 (для выполняемых файлов) и в ряде других.

Этот способ распространен мало, и вот почему. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Не всегда имеется подходящий по размеру свободный фрагмент для нового файла. Проблема непрерывного расположения может рассматриваться как частный случай более общей проблемы выделения блока нужного размера из списка свободных блоков. Типовыми решениями этой задачи являются стратегии первого подходящего, наиболее подходящего и наименее подходящего (сравните с проблемой выделения памяти в методе с динамическим распределением). Как и в случае выделения нужного объема оперативной памяти в схеме с динамическими разделами, метод страдает от внешней фрагментации, в большей или меньшей степени, в зависимости от размера диска и среднего размера файла.

Кроме того, непрерывное распределение внешней памяти неприменимо до тех пор, пока неизвестен максимальный размер файла. Иногда размер выходного файла оценить легко (при копировании). Чаще, однако, это трудно сделать, особенно в тех случаях, когда размер файла меняется. Если места не хватило, то пользовательская программа может быть приостановлена с учетом выделения дополнительного места для файла при последующем рестарте. Некоторые ОС используют модифицированный вариант непрерывного выделения - основные блоки файла + резервные блоки. Однако с выделением блоков из резерва возникают те же проблемы, так как приходится решать задачу выделения непрерывной последовательности блоков диска теперь уже из совокупности резервных блоков.

Единственным приемлемым решением перечисленных проблем является периодическое уплотнение содержимого внешней памяти, или "сборка мусора", цель которой состоит в объединении свободных

участков в один большой блок. Но это дорогостоящая операция, которую невозможно осуществлять слишком часто.

Таким образом, когда содержимое диска постоянно изменяется, данный метод нерационален. Однако для стационарных файловых систем, например для файловых систем компакт-дисков, он вполне пригоден.

5.14. Связный список

Внешняя фрагментация - основная проблема рассмотренного выше метода - может быть устранена за счет представления файла в виде связного списка блоков диска. Запись в директории содержит указатель на первый и последний блоки файла (иногда в качестве варианта используется специальный знак конца файла - EOF). Каждый блок содержит указатель на следующий блок (см. рис. 5.6).

Рис. 5.6. Хранение файла в виде связного списка дисковых блоков

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса. Заметим, что нет необходимости декларировать размер файла в момент создания. Файл может расти неограниченно.

Связное выделение имеет, однако, несколько существенных недостатков:

- при прямом доступе к файлу для поиска i-го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до i-1, то есть выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени. Здесь мы теряем все преимущества прямого доступа к файлу.
- данный способ не очень надежен. Наличие дефектного блока в списке приводит к потере информации в оставшейся части файла и потенциально к потере дискового пространства, отведенного под этот файл.

- для указателя на следующий блок внутри блока нужно выделить место, что не всегда удобно. Емкость блока, традиционно являющаяся степенью двойки (многие программы читают и пишут блоками по степеням двойки), таким образом, перестает быть степенью двойки, так как указатель отбирает несколько байтов.

Поэтому метод связного списка обычно в чистом виде не используется.

5.15. Таблица отображения файлов

Одним из вариантов предыдущего способа является хранение указателей не в дисковых блоках, а в индексной таблице в памяти, которая называется таблицей отображения файлов (FAT - file allocation table) (см. рис. 5.7). Этой схемы придерживаются многие ОС (MS-DOS, OS/2, MS Windows и др.)

Номера блоков диска		
1		
2	10	
3	11	Начало файла F_2
4		
5	EOF	
6	2	Начало файла F_1
7	EOF	
8		
9		
10	7	
11	5	

Рис. 5.7. Метод связного списка с использованием таблицы в оперативной памяти

По-прежнему существенно, что запись в директории содержит только ссылку на первый блок. Далее при помощи таблицы FAT можно локализовать блоки файла независимо от его размера. В тех строках таблицы, которые соответствуют последним блокам файлов, обычно записывается некоторое граничное значение, например EOF.

Главное достоинство данного подхода состоит в том, что по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Минусом данной схемы может быть необходимость хранения в памяти этой довольно большой таблицы.

5.16. Индексные узлы

Наиболее распространенный метод выделения файлу блоков диска - связать с каждым файлом небольшую таблицу, называемую индексным узлом (i-node), которая перечисляет атрибуты и дисковые адреса блоков файла (см. рис 5.8). Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения.

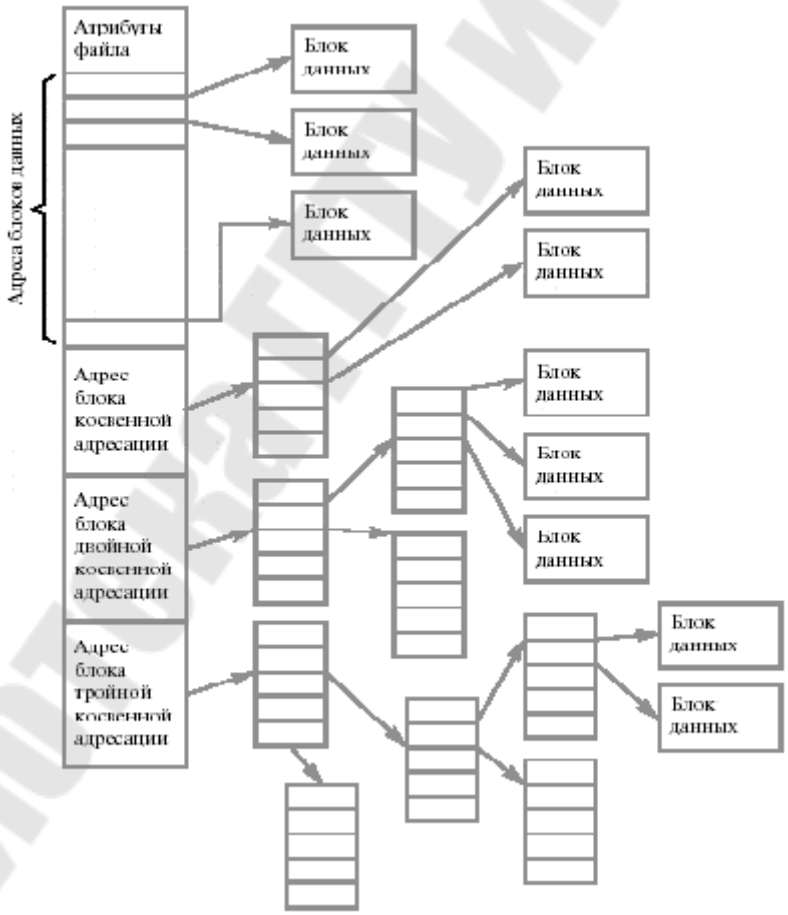

Рис. 5.8. Структура индексного узла

Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации. Индексированное размещение широко распространено и поддерживает как последовательный, так и прямой доступ к файлу.

Обычно применяется комбинация одноуровневого и многоуровневых индексов. Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, таким образом, для маленьких файлов индексный узел хранит всю необходимую информацию об адресах блоков диска. Для больших файлов один из адресов индексного узла указывает на блок косвенной адресации. Данный блок содержит адреса дополнительных блоков диска. Если этого недостаточно, используется блок двойной косвенной адресации, который содержит адреса блоков косвенной адресации. Если и этого не хватает, используется блок тройной косвенной адресации.

Данную схему используют файловые системы Unix (а также файловые системы HPFS, NTFS и др.). Такой подход позволяет при фиксированном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от нескольких байтов до нескольких гигабайтов. Существенно, что для маленьких файлов используется только прямая адресация, обеспечивающая максимальную производительность.

5.17. Управление свободным и занятым дисковым пространством

Дисковое пространство, не выделенное ни одному файлу, также должно быть управляемым. В современных ОС используется несколько способов учета используемого места на диске. Рассмотрим наиболее распространенные.

Часто список свободных блоков диска реализован в виде битового вектора (bit map или bit vector). Каждый блок представлен одним битом, принимающим значение 0 или 1, в зависимости от того, занят он или свободен. Например, 00111100111100011000001 ... .Главное преимущество этого подхода состоит в том, что он относительно прост и эффективен при нахождении первого свободного блока или n последовательных блоков на диске. Многие компьютеры имеют инструкции манипулирования битами, которые могут использоваться для этой цели. Например, компьютеры семейств Intel и Motorola имеют инструкции, при помощи которых можно легко локализовать первый

единичный бит в слове. Описываемый метод учета свободных блоков используется в Apple Macintosh.

Несмотря на то что размер описанного битового вектора наименьший из всех возможных структур, даже такой вектор может оказаться большого размера. Поэтому данный метод эффективен, только если битовый вектор помещается в памяти целиком, что возможно лишь для относительно небольших дисков. Например, диск размером 4 Гбайт с блоками по 4 Кбайт нуждается в таблице размером 128 Кбайт для управления свободными блоками. Иногда, если битовый вектор становится слишком большим, для ускорения поиска в нем его разбивают на регионы и организуют резюмирующие структуры данных, содержащие сведения о количестве свободных блоков для каждого региона.

Другой подход - связать в список все свободные блоки, размещая указатель на первый свободный блок в специально отведенном месте диска, попутно кэшируя в памяти эту информацию. Подобная схема не всегда эффективна. Для трассирования списка нужно выполнить много обращений к диску. Однако, к счастью, нам необходим, как правило, только первый свободный блок.

Иногда прибегают к модификации подхода связного списка, организуя хранение адресов n свободных блоков в первом свободном блоке. Первые n-1 этих блоков действительно используются. Последний блок содержит адреса других n блоков и т. д.

Существуют и другие методы, например, свободное пространство можно рассматривать как файл и вести для него соответствующий индексный узел.

5.18. Размер блока

Размер логического блока играет важную роль. В некоторых системах (Unix) он может быть задан при форматировании диска. Небольшой размер блока будет приводить к тому, что каждый файл будет содержать много блоков. Чтение блока осуществляется с задержками на поиск и вращение, таким образом, файл из многих блоков будет читаться медленно. Большие блоки обеспечивают более высокую скорость обмена с диском, но из-за внутренней фрагментации (каждый файл занимает целое число блоков, и в среднем половина последнего блока пропадает) снижается процент полезного дискового

пространства. Для систем со страничной организацией памяти характерна сходная проблема с размером страницы.

Проведенные исследования показали, что большинство файлов имеют небольшой размер. Например, в Unix приблизительно 85% файлов имеют размер менее 8 Кбайт и 48% - менее 1 Кбайта.

Можно также учесть, что в системах с виртуальной памятью желательно, чтобы единицей пересылки диск-память была страница (наиболее распространенный размер страниц памяти - 4 Кбайта). Отсюда обычный компромиссный выбор блока размером 512 байт, 1 Кбайт, 2 Кбайт, 4 Кбайт.

5.19. Структура файловой системы на диске

Рассмотрение методов работы с дисковым пространством дает общее представление о совокупности служебных данных, необходимых для описания файловой системы. Структура служебных данных типовой файловой системы, например Unix, на одном из разделов диска, таким образом, может состоять из четырех основных частей (см. рис. 5.9).

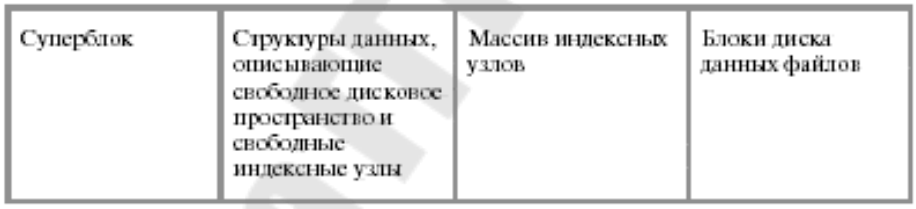

Рис. 5.9. Примерная структура файловой системы на диске

В начале раздела находится суперблок, содержащий общее описание файловой системы, например:

- тип файловой системы;
- размер файловой системы в блоках;
- размер массива индексных узлов;
- размер логического блока.

Описанные структуры данных создаются на диске в результате его форматирования (например, утилитами format, makefs и др.). Их наличие позволяет обращаться к данным на диске как к файловой системе, а не как к обычной последовательности блоков.

В файловых системах современных ОС для повышения устойчивости поддерживается несколько копий суперблока. В некоторых вер-

сиях Unix суперблок включал также и структуры данных, управляющие распределением дискового пространства, в результате чего суперблок непрерывно подвергался модификации, что снижало надежность файловой системы в целом. Выделение структур данных, описывающих дисковое пространство, в отдельную часть является более правильным решением.

Массив индексных узлов (ilist) содержит список индексов, соответствующих файлам данной файловой системы. Размер массива индексных узлов определяется администратором при установке системы. Максимальное число файлов, которые могут быть созданы в файловой системе, определяется числом доступных индексных узлов.

В блоках данных хранятся реальные данные файлов. Размер логического блока данных может задаваться при форматировании файловой системы. Заполнение диска содержательной информацией предполагает использование блоков хранения данных для файлов директорий и обычных файлов и имеет следствием модификацию массива индексных узлов и данных, описывающих пространство диска. Отдельно взятый блок данных может принадлежать одному и только одному файлу в файловой системе.

5.20. Реализация директорий

Как уже говорилось, директория или каталог - это файл, имеющий вид таблицы и хранящий список входящих в него файлов или каталогов. Основная задача файлов-директорий - поддержка иерархической древовидной структуры файловой системы. Запись в директории имеет определенный для данной ОС формат, зачастую неизвестный пользователю, поэтому блоки данных файла-директории заполняются не через операции записи, а при помощи специальных системных вызовов (например, создание файла).

Для доступа к файлу ОС использует путь (pathname), сообщенный пользователем. Запись в директории связывает имя файла или имя поддиректории с блоками данных на диске (см. рис. 5.10). В зависимости от способа выделения файлу блоков диска (см. раздел "Методы выделения дискового пространства") эта ссылка может быть номером первого блока или номером индексного узла. В любом случае обеспечивается связь символьного имени файла с данными на диске.

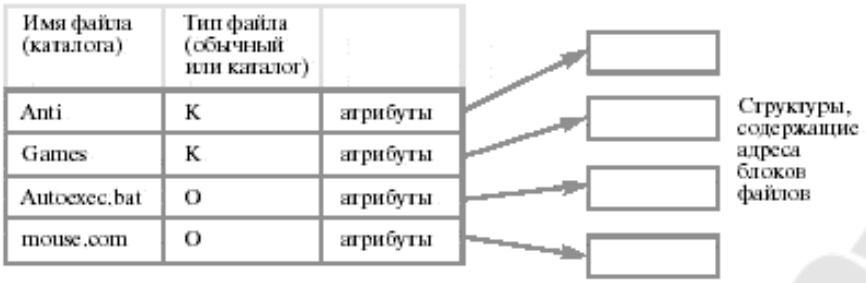

Рис. 5.10. Реализация директорий

Когда система открывает файл, она ищет его имя в директории. Затем из записи в директории или из структуры, на которую запись в директории указывает, извлекаются атрибуты и адреса блоков файла на диске. Эта информация помещается в системную таблицу в главной памяти. Все последующие ссылки на данный файл используют эту информацию. Атрибуты файла можно хранить непосредственно в записи в директории, как показано на рис. 12.6. Однако для организации совместного доступа к файлам удобнее хранить атрибуты в индексном узле, как это делается в Unix.

В ОС MS-DOS типовая запись в директории имеет вид, показанный на рис. 5.11.

Рис. 5.11. Вариант записи в директории MS-DOS

В ОС MS-DOS, как и в большинстве современных ОС, директории могут содержать поддиректории (специфицируемые битом атрибута), что позволяет конструировать произвольное дерево директорий файловой системы. Номер первого блока используется в качестве индекса в таблице FAT. Далее по цепочке в этой таблице могут быть найдены остальные блоки.

В ОС Unix структура директории проста. Каждая запись содержит имя файла и номер его индексного узла (см. рис. 5.12). Вся остальная информация о файле (тип, размер, время модификации, владелец и т. д. и номера дисковых блоков) находится в индексном узле.

Рис. 5.12. Вариант записи в директории Unix

В более поздних версиях Unix форма записи претерпела ряд изменений, например имя файла описывается структурой. Однако суть осталась прежней.

5.21. Поиск в директории

Список файлов в директории обычно не является упорядоченным по именам файлов. Поэтому правильный выбор алгоритма поиска имени файла в директории имеет большое влияние на эффективность и надежность файловых систем.

Существует несколько стратегий просмотра списка символьных имен. Простейшей из них является линейный поиск. Директория просматривается с самого начала, пока не встретится нужное имя файла. Хотя это наименее эффективный способ поиска, оказывается, что в большинстве случаев он работает с приемлемой производительностью. Например, авторы Unix утверждали, что линейного поиска вполне достаточно. По-видимому, это связано с тем, что на фоне относительно медленного доступа к диску некоторые задержки, возникающие в процессе сканирования списка, несущественны. Метод прост, но требует временных затрат. Для создания нового файла вначале нужно проверить директорию на наличие такого же имени. Затем имя нового файла вставляется в конец директории (если, разумеется, файл с таким же именем в директории не существует, в противном случае нужно информировать пользователя). Для удаления файла нужно также выполнить поиск его имени в списке и пометить запись как неиспользуемую.

Реальный недостаток данного метода - последовательный поиск файла. Информация о структуре директории используется часто, и неэффективный способ поиска будет заметен пользователями. Можно свести поиск к бинарному, если отсортировать список файлов. Однако это усложнит создание и удаление файлов, так как требуется перемещение большого объема информации.

Хеширование – другой способ, который может использоваться для размещения и последующего поиска имени файла в директории. В данном методе имена файлов также хранятся в каталоге в виде линейного списка, но дополнительно используется хеш-таблица. Хеш-таблица, точнее построенная на ее основе хеш-функция, позволяет по имени файла получить указатель на имя файла в списке. Таким обра-

зом, можно существенно уменьшить время поиска. В результате хеширования могут возникать коллизии, то есть ситуации, когда функция хеширования, примененная к разным именам файлов, дает один и тот же результат. Обычно имена таких файлов объединяют в связные списки, предполагая в дальнейшем осуществление в них последовательного поиска нужного имени файла. Выбор подходящего алгоритма хеширования позволяет свести к минимуму число коллизий. Однако всегда есть вероятность неблагоприятного исхода, когда непропорционально большому числу имен файлов функция хеширования ставит в соответствие один и тот же результат. В таком случае преимущество использования этой схемы по сравнению с последовательным поиском практически утрачивается.

Помимо описанных методов поиска имени файла, в директории существуют и другие. В качестве примера можно привести организацию поиска в каталогах файловой системы NTFS при помощи так называемого B-дерева, которое стало стандартным способом организации индексов в системах баз данных.

5.22. Монтирование файловых систем

Так же как файл должен быть открыт перед использованием, и файловая система, хранящаяся на разделе диска, должна быть смонтирована, чтобы стать доступной процессам системы.

Функция mount (монтировать) связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем, а функция umount (демонтировать) выключает файловую систему из иерархии. Функция mount, таким образом, дает пользователям возможность обращаться к данным в дисковом разделе как к файловой системе, а не как к последовательности дисковых блоков. Процедура монтирования состоит в следующем. Пользователь (в Unix это суперпользователь) сообщает ОС имя устройства и место в файловой структуре (имя пустого каталога), куда нужно присоединить файловую систему (точка монтирования) (см. рис. 5.13 и рис. 5.14). Например, в ОС Unix библиотечный вызов mount имеет вид:

```
mount(special pathname,directory pathname,
options);
```

где special pathname - имя специального файла устройства (в общем случае имя раздела), соответствующего дисковому разделу с монтируемой файловой системой, directory pathname - каталог в существующей иерархии, где будет монтироваться файловая система (другими словами, точка или место монтирования), а options указывает, следует ли монтировать файловую систему "только для чтения" (при этом не будут выполняться такие функции, как write и create, которые производят запись в файловую систему). Затем ОС должна убедиться, что устройство содержит действительную файловую систему ожидаемого формата с суперблоком, списком индексов и корневым индексом.

Некоторые ОС осуществляют монтирование автоматически, как только встретят диск в первый раз (жесткие диски на этапе загрузки, гибкие - когда они вставлены в дисковод), ОС ищет файловую систему на устройстве. Если файловая система на устройстве имеется, она монтируется на корневом уровне, при этом к цепочке имен абсолютного имени файла (pathname) добавляется буква раздела.

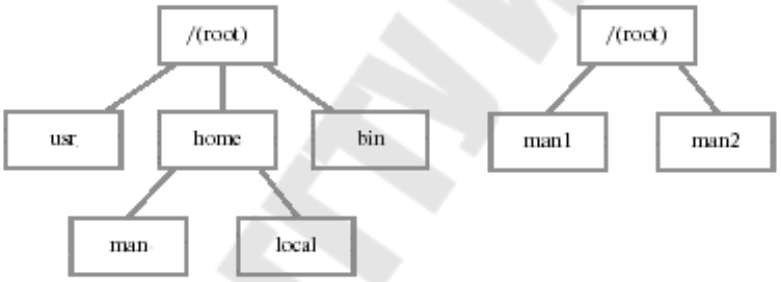

Рис. 5.13. Две файловые системы до монтирования

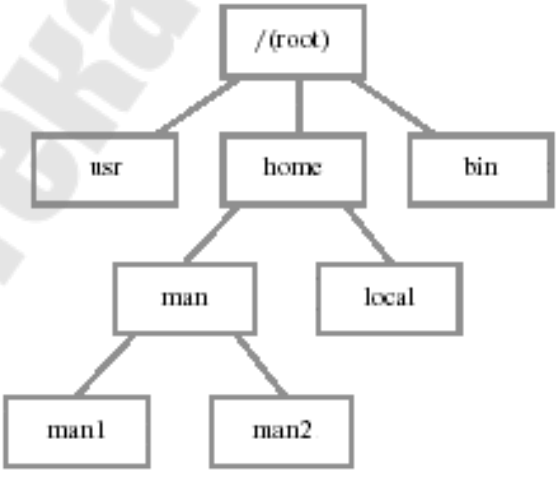

Рис. 5.14. Общая файловая система после монтирования

Ядро поддерживает таблицу монтирования с записями о каждой смонтированной файловой системе. В каждой записи содержится информация о вновь смонтированном устройстве, о его суперблоке и корневом каталоге, а также сведения о точке монтирования. Для устранения потенциально опасных побочных эффектов число линков (см. следующий раздел) к каталогу - точке монтирования - должно быть равно 1. Занесение информации в таблицу монтирования производится немедленно, поскольку может возникнуть конфликт между двумя процессами. Например, если монтирующий процесс приостановлен для открытия устройства или считывания суперблока файловой системы, а тем временем другой процесс может попытаться смонтировать файловую систему.

Наличие в логической структуре файлового архива точек монтирования требует аккуратной реализации алгоритмов, осуществляющих навигацию по каталогам. Точку монтирования можно пересечь двумя способами: из файловой системы, где производится монтирование, в файловую систему, которая монтируется (в направлении от глобального корня к листу), и в обратном направлении. Алгоритмы поиска файлов должны предусматривать ситуации, в которых очередной компонент пути к файлу является точкой монтирования, когда вместо анализа индексного узла очередной директории приходится осуществлять обработку суперблока монтированной системы.

5.23. Связывание файлов

Иерархическая организация, положенная в основу древовидной структуры файловой системы современных ОС, не предусматривает выражения отношений, в которых потомки связываются более чем с одним предком. Такая негибкость частично устраняется возможностью реализации связывания файлов или организации линков (link).

Ядро позволяет пользователю связывать каталоги, упрощая написание программ, требующих пересечения дерева файловой системы (см. рис. 5.15). Часто имеет смысл хранить под разными именами одну и ту же команду (выполняемый файл). Например, выполняемый файл традиционного текстового редактора ОС Unix vi обычно может вызываться под именами ex, edit, vi, view и vedit файловой системы. Соединение между директорией и разделяемым файлом называется "связью" или "ссылкой" (link). Дерево файловой системы превращает-

ся в циклический граф. Это удобно, но создает ряд дополнительных проблем.

Простейший способ реализовать связывание файла - просто дублировать информацию о нем в обеих директориях. При этом, однако, может возникнуть проблема совместимости в случае, если владельцы этих директорий попытаются независимо друг от друга изменить содержимое файла. Например, в ОС CP/M запись в директории о файле непосредственно содержит адреса дисковых блоков. Поэтому копии тех же дисковых адресов должны быть сделаны и в другой директории, куда файл линкуется. Если один из пользователей что-то добавляет к файлу, новые блоки будут перечислены только у него в директории и не будут "видны" другому пользователю.

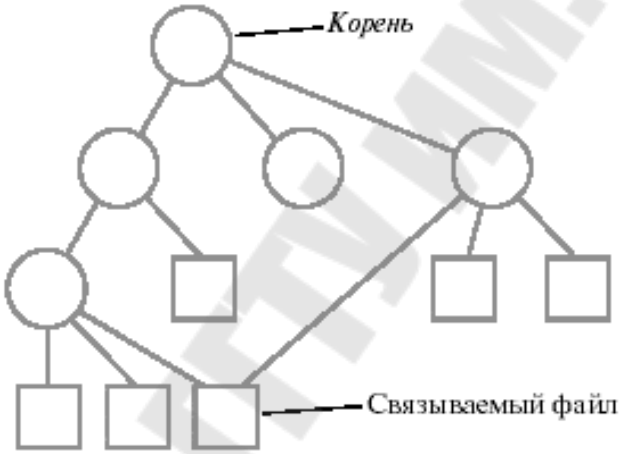

Рис. 5.15. Структура файловой системы с возможностью связывания файла с новым именем

Проблема такого рода может быть решена двумя способами. Первый из них - так называемая жесткая связь (hard link). Если блоки данных файла перечислены не в директории, а в небольшой структуре данных (например, в индексном узле), связанной собственно с файлом, то второй пользователь может связаться непосредственно с этой, уже существующей структурой.

Альтернативное решение - создание нового файла, который содержит путь к связываемому файлу. Такой подход называется символической линковкой (soft или symbolic link). При этом в соответствующем каталоге создается элемент, в котором имени связи сопоставляется некоторое имя файла (этот файл даже не обязан существовать к моменту создания символической связи). Для символической

связи может создаваться отдельный индексный узел и даже заводиться отдельный блок данных для хранения потенциально длинного имени файла.

Каждый из этих методов имеет свои минусы. В случае жесткой связи возникает необходимость поддержки счетчика ссылок на файл для корректной реализации операции удаления файла. Например, в Unix такой счетчик является одним из атрибутов, хранящихся в индексном узле. Удаление файла одним из пользователей уменьшает количество ссылок на файл на 1. Реальное удаление файла происходит, когда число ссылок на файл становится равным 0.

В случае символической линковки такая проблема не возникает, так как только реальный владелец имеет ссылку на индексный узел файла. Если собственник удаляет файл, то он разрушается, и попытки других пользователей работать с ним закончатся провалом. Удаление символического линка на файл никак не влияет. Проблема организации символической связи - потенциальное снижение скорости доступа к файлу. Файл символического линка хранит путь к файлу, содержащий список вложенных директорий, для прохождения по которому необходимо осуществить несколько обращений к диску.

Символический линк имеет то преимущество, что он может использоваться для организации удобного доступа к файлам удаленных компьютеров, если, например, добавить к пути сетевой адрес удаленной машины.

Циклический граф - структура более гибкая, нежели простое дерево, но работа с ней требует большой аккуратности. Поскольку теперь к файлу существует несколько путей, программа поиска файла может найти его на диске несколько раз. Простейшее практическое решение данной проблемы - ограничить число директорий при поиске. Полное устранение циклов при поиске - довольно трудоемкая процедура, выполняемая специальными утилитами и связанная с многократной трассировкой директорий файловой системы.

5.24. Кооперация процессов при работе с файлами

Когда различные пользователи работают вместе над проектом, они часто нуждаются в разделении файлов.

Разделяемый файл - разделяемый ресурс. Как и в случае любого совместно используемого ресурса, процессы должны синхронизировать доступ к совместно используемым файлам, каталогам, чтобы из-

бежать тупиковых ситуаций, дискриминации отдельных процессов и снижения производительности системы.

Например, если несколько пользователей одновременно редактируют какой-либо файл и не принято специальных мер, то результат будет непредсказуем и зависит от того, в каком порядке осуществлялись записи в файл. Между двумя операциями read одного процесса другой процесс может модифицировать данные, что для многих приложений неприемлемо. Простейшее решение данной проблемы - предоставить возможность одному из процессов захватить файл, то есть блокировать доступ к разделяемому файлу других процессов на все время, пока файл остается открытым для данного процесса. Однако это было бы недостаточно гибко и не соответствовало бы характеру поставленной задачи.

Рассмотрим вначале грубый подход, то есть временный захват пользовательским процессом файла или записи (части файла между указанными позициями).

Системный вызов, позволяющий установить и проверить блокировки на файл, является неотъемлемым атрибутом современных многопользовательских ОС. В принципе, было бы логично связать синхронизацию доступа к файлу как к единому целому с системным вызовом open (т. е., например, открытие файла в режиме записи или обновления могло бы означать его монопольную блокировку соответствующим процессом, а открытие в режиме чтения - совместную блокировку). Так поступают во многих операционных системах (начиная с ОС Multics). В ОС Unix это не так, что имеет исторические причины.

В первой версии системы Unix, разработанной Томпсоном и Ричи, механизм захвата файла отсутствовал. Применялся очень простой подход к обеспечению параллельного (от нескольких процессов) доступа к файлам: система позволяла любому числу процессов одновременно открывать один и тот же файл в любом режиме (чтения, записи или обновления) и не предпринимала никаких синхронизационных действий. Вся ответственность за корректность совместной обработки файла ложилась на использующие его процессы, и система даже не предоставляла каких-либо особых средств для синхронизации доступа процессов к файлу. Однако впоследствии для того, чтобы повысить привлекательность системы для коммерческих пользователей, работающих с базами данных, в версию V системы были включены меха-

низмы захвата файла и записи, базирующиеся на системном вызове fcntl.

Допускается два варианта синхронизации: с ожиданием, когда требование блокировки может привести к откладыванию процесса до того момента, когда это требование может быть удовлетворено, и без ожидания, когда процесс немедленно оповещается об удовлетворении требования блокировки или о невозможности ее удовлетворения в данный момент. Установленные блокировки относятся только к тому процессу, который их установил, и не наследуются процессами-потомками этого процесса. Более того, даже если некоторый процесс пользуется синхронизационными возможностями системного вызова fcntl, другие процессы по-прежнему могут работать с тем файлом без всякой синхронизации. Другими словами, это дело группы процессов, совместно использующих файл, - договориться о способе синхронизации параллельного доступа.

Более тонкий подход заключается в прозрачной для пользователя блокировке отдельных структур ядра, отвечающих за работу с файлами части пользовательских данных. Например, в ОС Unix во время системного вызова, осуществляющего ту или иную операцию с файлом, как правило, происходит блокирование индексного узла, содержащего адреса блоков данных файла. Может показаться, что организация блокировок или запрета более чем одному процессу работать с файлом во время выполнения системного вызова является излишней, так как в подавляющем большинстве случаев выполнение системных вызовов и так не прерывается, то есть ядро работает в условиях невытесняющей многозадачности. Однако в данном случае это не совсем так. Операции чтения и записи занимают продолжительное время и лишь инициируются центральным процессором, а осуществляются по независимым каналам, поэтому установка блокировок на время системного вызова является необходимой гарантией атомарности операций чтения и записи. На практике оказывается достаточным заблокировать один из буферов кэша диска, в заголовке которого ведется список процессов, ожидающих освобождения данного буфера. Таким образом, в соответствии с семантикой Unix изменения, сделанные одним пользователем, немедленно становятся "видны" другому пользователю, который держит данный файл открытым одновременно с первым.

5.25. Примеры разрешения коллизий и тупиковых ситуаций

Логика работы системы в сложных ситуациях может проиллюстрировать особенности организации мультидоступа. Рассмотрим в качестве примера образование потенциального тупика при создании связи (link), когда разрешен совместный доступ к файлу. Два процесса, выполняющие одновременно следующие функции:

процесс A: `link("a/b/c/d","e/f/g");`
процесс B: `link("e/f","a/b/c/d/ee");`

могут зайти в тупик. Предположим, что процесс A обнаружил индекс файла "a/b/c/d" в тот самый момент, когда процесс B обнаружил индекс файла "e/f". Фраза "в тот же самый момент" означает, что системой достигнуто состояние, при котором каждый процесс получил искомый индекс. Когда же теперь процесс A попытается получить индекс файла "e/f", он приостановит свое выполнение до тех пор, пока индекс файла "f" не освободится. В то же время процесс B пытается получить индекс каталога "a/b/c/d" и приостанавливается в ожидании освобождения индекса файла "d". Процесс A будет удерживать заблокированным индекс, нужный процессу B, а процесс B, в свою очередь, будет удерживать заблокированным индекс, необходимый процессу A.

Для предотвращения этого классического примера взаимной блокировки в файловой системе принято, чтобы ядро освобождало индекс исходного файла после увеличения значения счетчика связей. Тогда, поскольку первый из ресурсов (индекс) свободен при обращении к следующему ресурсу, взаимной блокировки не происходит.

Поводов для нежелательной конкуренции между процессами много, особенно при удалении имен каталогов. Предположим, что один процесс пытается найти данные файла по его полному символическому имени, последовательно проходя компонент за компонентом, а другой процесс удаляет каталог, имя которого входит в путь поиска. Допустим, процесс A делает разбор имени "a/b/c/d" и приостанавливается во время получения индексного узла для файла "c". Он может приостановиться при попытке заблокировать индексный узел или при попытке обратиться к дисковому блоку, где этот индексный узел хранится. Если процессу B нужно удалить связь для каталога с именем "c", он может приостановиться по той же самой причине, что и процесс A. Пусть ядро впоследствии решит возобновить процесс B рань-

ше процесса А. Прежде чем процесс А продолжит свое выполнение, процесс В завершится, удалив связь каталога "с" и его содержимое по этой связи. Позднее процесс А попытается обратиться к несуществующему индексному узлу, который уже был удален. Алгоритм поиска файла, проверяющий в первую очередь неравенство значения счетчика связей> нулю, должен сообщить об ошибке.

5.26. Надежность файловой системы

Жизнь полна неприятных неожиданностей, а разрушение файловой системы зачастую более опасно, чем разрушение компьютера. Поэтому файловые системы должны разрабатываться с учетом подобной возможности. Помимо очевидных решений, например своевременное дублирование информации (backup), файловые системы современных ОС содержат специальные средства для поддержки собственной совместимости.

Важный аспект надежной работы файловой системы - контроль ее целостности. В результате файловых операций блоки диска могут считываться в память, модифицироваться и затем записываться на диск. Причем многие файловые операции затрагивают сразу несколько объектов файловой системы. Например, копирование файла предполагает выделение ему блоков диска, формирование индексного узла, изменение содержимого каталога и т. д. В течение короткого периода времени между этими шагами информация в файловой системе оказывается несогласованной. И если вследствие непредсказуемой остановки системы на диске будут сохранены изменения только для части этих объектов (нарушена атомарность файловой операции), файловая система на диске может быть оставлена в несовместимом состоянии. В результате могут возникнуть нарушения логики работы с данными, например появиться "потерянные" блоки диска, которые не принадлежат ни одному файлу и в то же время помечены как занятые, или, наоборот, блоки, помеченные как свободные, но в то же время занятые (на них есть ссылка в индексном узле) или другие нарушения.

В современных ОС предусмотрены меры, которые позволяют свести к минимуму ущерб от порчи файловой системы и затем полностью или частично восстановить ее целостность.

Очевидно, что для правильного функционирования файловой системы значимость отдельных данных неравноценна. Искажение со-

держимого пользовательских файлов не приводит к серьезным (с точки зрения целостности файловой системы) последствиям, тогда как несоответствия в файлах, содержащих управляющую информацию (директории, индексные узлы, суперблок и т. п.), могут быть катастрофическими. Поэтому должен быть тщательно продуман порядок выполнения операций со структурами данных файловой системы.

Рассмотрим пример создания жесткой связи для файла. Для этого файловой системе необходимо выполнить следующие операции:

- создать новую запись в каталоге, указывающую на индексный узел файла;
- увеличить счетчик связей в индексном узле.

Если аварийный останов произошел между 1-й и 2-й операциями, то в каталогах файловой системы будут существовать два имени файла, адресующих индексный узел со значением счетчика связей, равному 1. Если теперь будет удалено одно из имен, это приведет к удалению файла как такового. Если же порядок операций изменен и, как прежде, останов произошел между первой и второй операциями, файл будет иметь несуществующую жесткую связь, но существующая запись в каталоге будет правильной. Хотя это тоже является ошибкой, но ее последствия менее серьезны, чем в предыдущем случае.

Другим средством поддержки целостности является заимствованный из систем управления базами данных прием, называемый журнализация (иногда употребляется термин "журналирование"). Последовательность действий с объектами во время файловой операции протоколируется, и если произошел останов системы, то, имея в наличии протокол, можно осуществить откат системы назад в исходное целостное состояние, в котором она пребывала до начала операции. Подобная избыточность может стоить дорого, но она оправданна, так как в случае отказа позволяет реконструировать потерянные данные. Для отката необходимо, чтобы для каждой протоколируемой в журнале операции существовала обратная. Например, для каталогов и реляционных СУБД это именно так. По этой причине, в отличие от СУБД, в файловых системах протоколируются не все изменения, а лишь изменения метаданных (индексных узлов, записей в каталогах и др.). Изменения в данных пользователя в протокол не заносятся. Кроме того, если протоколировать изменения пользовательских данных, то этим будет нанесен серьезный ущерб производительности системы, поскольку кэширование потеряет смысл.

Журнализация реализована в NTFS, Ext3FS, ReiserFS и других системах. Чтобы подчеркнуть сложность задачи, нужно отметить, что существуют не вполне очевидные проблемы, связанные с процедурой отката. Например, отмена одних изменений может затрагивать данные, уже использованные другими файловыми операциями. Это означает, что такие операции также должны быть отменены. Данная проблема получила название каскадного отката транзакций.

Если же нарушение все же произошло, то для устранения проблемы несовместимости можно прибегнуть к утилитам (fsck, chkdsk, scandisk и др.), которые проверяют целостность файловой системы. Они могут запускаться после загрузки или после сбоя и осуществляют многократное сканирование разнообразных структур данных файловой системы в поисках противоречий.

Возможны также эвристические проверки. Например, нахождение индексного узла, номер которого превышает их число на диске или поиск в пользовательских директориях файлов, принадлежащих суперпользователю.

К сожалению, приходится констатировать, что не существует никаких средств, гарантирующих абсолютную сохранность информации в файлах, и в тех ситуациях, когда целостность информации нужно гарантировать с высокой степенью надежности, прибегают к дорогостоящим процедурам дублирования.

5.27. Управление "плохими" блоками

Наличие дефектных блоков на диске - обычное дело. Внутри блока наряду с данными хранится контрольная сумма данных. Под "плохими" блоками обычно понимают блоки диска, для которых вычисленная контрольная сумма считываемых данных не совпадает с хранимой контрольной суммой. Дефектные блоки обычно появляются в процессе эксплуатации. Иногда они уже имеются при поставке вместе со списком, так как очень затруднительно для поставщиков сделать диск полностью свободным от дефектов. Рассмотрим два решения проблемы дефектных блоков - одно на уровне аппаратуры, другое на уровне ядра ОС.

Первый способ - хранить список плохих блоков в контроллере диска. Когда контроллер инициализируется, он читает плохие блоки и замещает дефектный блок резервным, помечая отображение в списке плохих блоков. Все реальные запросы будут идти к резервному блоку.

Следует иметь в виду, что при этом механизм подъемника (наиболее распространенный механизм обработки запросов к блокам диска) будет работать неэффективно. Дело в том, что существует стратегия очередности обработки запросов к диску (подробнее см. лекцию "ввод-вывод"). Стратегия диктует направление движения считывающей головки диска к нужному цилиндру. Обычно резервные блоки размещаются на внешних цилиндрах. Если плохой блок расположен на внутреннем цилиндре и контроллер осуществляет подстановку прозрачным образом, то кажущееся движение головки будет осуществляться к внутреннему цилиндру, а фактическое - к внешнему. Это является нарушением стратегии и, следовательно, минусом данной схемы.

Решение на уровне ОС может быть следующим. Прежде всего, необходимо тщательно сконструировать файл, содержащий дефектные блоки. Тогда они изымаются из списка свободных блоков. Затем нужно каким-то образом скрыть этот файл от прикладных программ.

5.28. Кэширование

Поскольку обращение к диску - операция относительно медленная, минимизация количества таких обращений - ключевая задача всех алгоритмов, работающих с внешней памятью. Наиболее типичная техника повышения скорости работы с диском - кэширование.

Кэш диска представляет собой буфер в оперативной памяти, содержащий ряд блоков диска (см. рис. 5.16). Если имеется запрос на чтение/запись блока диска, то сначала производится проверка на предмет наличия этого блока в кэше. Если блок в кэше имеется, то запрос удовлетворяется из кэша, в противном случае запрошенный блок считывается в кэш с диска. Сокращение количества дисковых операций оказывается возможным вследствие присущего ОС свойства локальности (о свойстве локальности много говорилось в лекциях, посвященных описанию работы системы управления памятью).

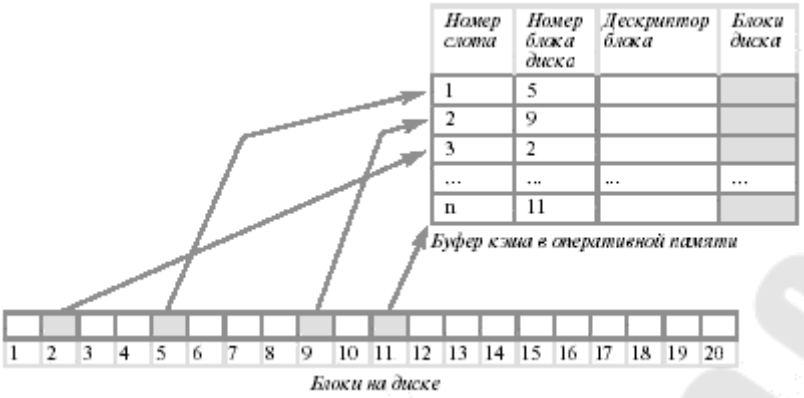

Рис. 5.16. Структура блочного кэша

Аккуратная реализация кэширования требует решения нескольких проблем:

- емкость буфера кэша ограничена. Когда блок должен быть загружен в заполненный буфер кэша, возникает проблема замещения блоков, то есть отдельные блоки должны быть удалены из него. Здесь работают те же стратегии и те же FIFO, Second Chance и LRU-алгоритмы замещения, что и при выталкивании страниц памяти.
- замещение блоков должно осуществляться с учетом их важности для файловой системы. Блоки должны быть разделены на категории, например: блоки индексных узлов, блоки косвенной адресации, блоки директорий, заполненные блоки данных и т. д., и в зависимости от принадлежности блока к той или иной категории можно применять к ним разную стратегию замещения.
- поскольку кэширование использует механизм отложенной записи, при котором модификация буфера не вызывает немедленной записи на диск, серьезной проблемой является "старение" информации в дисковых блоках, образы которых находятся в буферном кэше. Несвоевременная синхронизация буфера кэша и диска может привести к очень нежелательным последствиям в случае отказов оборудования или программного обеспечения. Поэтому стратегия и порядок отображения информации из кэша на диск должна быть тщательно продумана. Так, блоки, существенные для совместимости файловой системы (блоки индексных узлов, блоки косвенной адресации, блоки директорий), должны быть переписаны на диск немедленно, независимо от того, в ка-

кой части LRU-цепочки они находятся. Необходимо тщательно выбрать порядок такого переписывания. В Unix имеется для этого вызов SYNC, который заставляет все модифицированные блоки записываться на диск немедленно. Для синхронизации содержимого кэша и диска периодически запускается фоновый процесс-демон. Кроме того, можно организовать синхронный режим работы с отдельными файлами, задаваемый при открытии файла, когда все изменения в файле немедленно сохраняются на диске.

- проблема конкуренции процессов на доступ к блокам кэша решается ведением списков блоков, пребывающих в различных состояниях, и отметкой о состоянии блока в его дескрипторе. Например, блок может быть заблокирован, участвовать в операции ввода-вывода, а также иметь список процессов, ожидающих освобождения данного блока.

5.29. Оптимальное размещение информации на диске

Кэширование - не единственный способ увеличения производительности системы. Другая важная техника - сокращение количества движений считывающей головки диска за счет разумной стратегии размещения информации. Например, массив индексных узлов в Unix стараются разместить на средних дорожках. Также имеет смысл размещать индексные узлы поблизости от блоков данных, на которые они ссылаются и т. д.

Кроме того, рекомендуется периодически осуществлять дефрагментацию диска (сборку мусора), поскольку в популярных методиках выделения дисковых блоков (за исключением, может быть, FAT) принцип локальности не работает, и последовательная обработка файла требует обращения к различным участкам диска.

Оглавление

Курочка Константин Сергеевич
Литвинов Дмитрий Александрович

ПРИНЦИПЫ ОРГАНИЗАЦИИ ВНЕШНЕЙ И ВНУТРЕННЕЙ ПАМЯТИ В ОПЕРАЦИОННЫХ СИСТЕМАХ

**Пособие
по курсам «Операционные системы»
и «Основы мультипроцессорной
и мультипрограммной обработки данных»
для студентов специальности 1-40 01 02
«Информационные системы и технологии
(по направлениям)»
дневной формы обучения**

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 26.11.2010.
Рег. № 38Е.
E-mail: ic@gstu.by
http://www.gstu.by