БИЛЕТ № 5

1) Схема расположения супервизора в ОП. Состав. Последовательность формирования состава.

Как обрабатывается прерывание по обращению к супервизору.

> *Прерывания по обращению к супервизору*. Вызываются при выполнении процессором **команды обращения к супервизору** (вызов функции операционной системы). Обычно такая команда инициируется выполняемым процессом при необходимости получения дополнительных ресурсов либо при взаимодействии с устройствами ввода/вывода.

Структура ядра супервизора.

Совокупность программ, обеспечивающих функционирование ВС и находящихся в системной области оперативной памяти, составляет ядро супервизора.

Структура ядра:

-- резидентные программы

-- системные таблицы

2) Управление задачами и ее функции.

3) Проблемы выбора загружаемой части программы в ОП и методы их решения.

Теперь рассмотрим алгоритм замещения страниц, основанный на рабочем наборе. Его базовая идея заключается в том, чтобы найти страницу, не включенную в рабочий набор, и выгрузить ее. На рис. 4.20 изображена часть таблицы страниц для некоторой машины. Поскольку в качестве кандидатов на удаление рассматриваются только те страницы, которые в настоящее время находятся в памяти, отсутствующие в памяти страницы этим алгоритмом игнорируются. Каждая запись содержит (по крайней мере) два элемента информации: приближенное время, в которое страница использовалась в последний раз, и бит *R* (обращения). Пустые белые прямоугольники символизируют другие поля, ненужные для данного алгоритма, такие как номер страничного блока, биты защиты и бит *M* (изменения).

Алгоритм работает следующим образом. Предполагается, что аппаратное обеспечение устанавливает биты *R* и *M*, как мы описывали выше. Предполагается также, что периодическое прерывание по таймеру вызывает запуск программы, очищающей бит *R* при каждом тике часов. При каждом страничном прерывании исследуется таблица страниц и ищется страница, подходящая для удаления из памяти.

В процессе обработки каждой записи проверяется бит *R*. Если он равен 1, текущее виртуальное время записывается в поле *Время последнего использования* (*Time of last use*) в таблице страниц, указывая, что страница использовалась в тот момент, когда произошло прерывание. Так как к странице было обращение в течение данного такта, ясно, что она находится в рабочем наборе и не является кандидатом на удаление (предполагается, что τ охватывает несколько тиков часов).

Если бит *R* равен 0, это означает, что к странице не было обращений в течение последнего тика часов и она может быть кандидатом на удаление. Чтобы понять, нужно ли ее выгружать, вычисляется ее возраст, то есть текущее виртуальное время минус ее *Время последнего использования*, и сравнивается с τ. Если возраст больше величины τ, это означает, что страница более не находится в рабочем наборе.

Хотя алгоритм «вторая попытка» является корректным, он слишком неэффективен, потому что постоянно передвигает страницы по списку. Поэтому лучше хранить все страничные блоки в кольцевом списке в форме часов, как показано на рис. 4.16. Стрелка указывает на старейшую страницу.

Когда происходит страничное прерывание, проверяется та страница, на которую направлена стрелка. Если ее бит *R* равен 0, страница выгружается, на ее место в часовой круг встает новая страница, а стрелка сдвигается вперед на одну позицию. Если бит *R* равен 1, то он сбрасывается, стрелка перемещается к следующей странице. Этот процесс повторяется до тех пор, пока не находится та страница, у которой бит *R* = 0. Неудивительно, что этот алгоритм называется **«часы»**. Он отличается от алгоритма «вторая попытка» только своей реализацией.

Рис. 4.16. Алгоритм замещения страниц «часы»

Алгоритм «рабочий набор»

В простейшей схеме страничной подкачки в момент запуска процессов нужные им страницы отсутствуют в памяти. Как только центральный процессор пытается выбрать первую команду, он получает страничное прерывание, побуждающее операционную систему перенести в память страницу, содержащую первую инструкцию. Обычно следом быстро происходят страничные прерывания для глобальных переменных и стека. Через некоторое время в памяти скапливается большинство необходимых процессу страниц, и он приступает к работе с относительно небольшим количеством ошибок из-за отсутствия страниц. Этот метод называется **замещением страниц по запросу** (demand paging), потому что страницы загружаются в память по требованию, а не заранее.

Конечно, достаточно легко написать тестовую программу, систематически читающую все страницы в огромном адресном пространстве, вызывая так много страничных прерываний, что будет не хватать памяти для их обработки. К счастью, большинство процессов не работают таким образом. Они характеризуются **локальностью обращений**, означающей, что во время выполнения любой фразы процесс обращается только к сравнительно небольшой части своих страниц. Каждый проход многоходового компилятора, например, обращается только к части от общего количества страниц, и каждый раз к другой части.

Другим требующим небольших издержек алгоритмом является **FIFO** (First-In, First-Out — «первым прибыл — первым обслужен»). Чтобы проиллюстрировать его работу, рассмотрим универсам, на полках которого можно выставить ровно k различных продуктов. Он предлагает новую удобную пищу: растворимый, глубоко замороженный, экологически чистый йогурт, который можно мгновенно приготовить в микроволновой печи. Покупатели тут же обратили внимание на этот продукт, поэтому наш ограниченный в размерах супермаркет, для того чтобы продавать йогурт, должен избавиться от одного из старых товаров.

Один из вариантов состоит в том, чтобы найти продукт, который супермаркет продает дольше всего (то есть что-нибудь, что начали реализовывать 120 лет назад), и освободить от него магазин на том основании, что им никто больше не интересуется. В действительности супермаркет хранит перечень всех продаваемых в данный момент товаров, упорядоченный по времени их появления. Каждый новый продукт помещается в конец перечня, а из начала списка удаляется одно старое наименование.

Ту же самую идею можно применить в качестве алгоритма замещения страниц. Операционная система поддерживает список всех страниц, находящихся в данный момент в памяти, в котором первая страница является старейшей, а страницы в хвосте списка попали в него совсем недавно. Когда происходит страничное прерывание, выгружается из памяти страница в голове списка, а новая страница добавляется в его конец. Если алгоритм FIFO использовать в магазине, то он может удалить воск для усов, но также может удалить и муку, соль или масло. Применительно к компьютерам возникает та же проблема. По этой причине алгоритм FIFO редко используется в своей исходной форме.

Она стирается, а на ее место загружается новая страница. Однако сканирование таблицы продолжается, обновляя остальные записи.

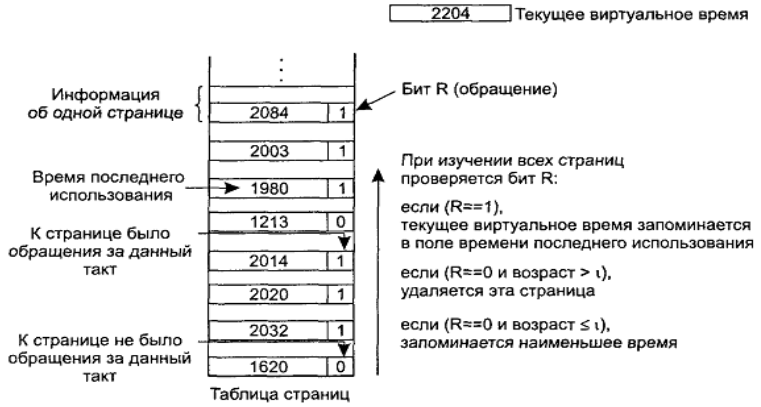

Рис. 4.20. Алгоритм «рабочий набор»

Если же бит R равен 0, но возраст страницы меньше или равен времени τ, это значит, что страница до сих пор находится в рабочем наборе. Она временно обходится, но страница с наибольшим возрастом запоминается (наименьшим значением *Времени последнего использования*). Если проверена вся таблица, а кандидат на удаление не найден, это означает, что все страницы входят в рабочий набор. В этом случае, если были найдены одна или больше страниц с битом $R = 0$, удаляется та из них, которая имеет наибольший возраст. В худшем случае ко всем страницам произошло обращение за время текущего такта часов (и, следовательно, все они имеют бит $R = 1$), тогда для удаления случайным образом выбирается одна из них, причем желательно чистая, если такая страница существует.

Предположим, что в момент времени 20 происходит страничное прерывание. Самой старшей страницей является страница *A*, она была загружена в память во время 0, когда начал работу процесс. Если бит *R* страницы *A* равен 0, она выгружается из памяти или путем записи на диск (если страница «грязная»), или просто удаляется (если она «чистая»). Во втором случае, если бит *R* равен 1, страница *A* передвигается в конец списка, а ее «загрузочное время» принимает текущее значение (20). При этом бит *R* очищается[1]. Поиск подходящей страницы продолжается; следующей проверяется страница *B*.

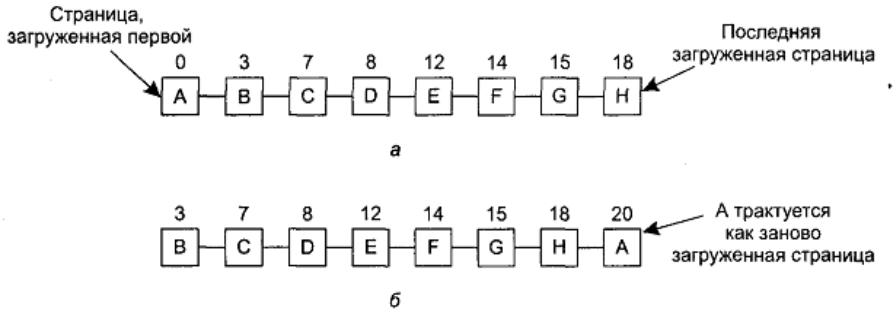

Рис. 4.15. Действие алгоритма «вторая попытка»: страницы, отсортированные в порядке очереди (FIFO) (*а*); список страниц, если страничное прерывание произошло во время 20, а страница *A* имеет бит *R*, равный 0 (*б*)

Алгоритм «вторая попытка» ищет в списке самую старую страницу, к которой *не было обращений в предыдущем временном интервале. Если же происходили* ссылки на все страницы, то «вторая попытка» превращается в обычный алгоритм FIFO. Представьте, что у всех страниц на рис. 4.15, *а* бит *R* равен 1. Одну за другой передвигает операционная система страницы в конец списка, очищая бит *R* каждый раз, когда она перемещает страницу в хвост. Наконец, она вернется к странице *A*, но теперь уже ее биту *R* присвоено значение 0. В этот момент страница *A* выгружается из памяти. Таким образом, алгоритм всегда успешно завершает свою работу.

все страницы ь памяти. Бит *R* каждой страницы (он равен 0 или 1) прибавляется к счетчику. В сущности, счетчики пытаются отследить, как часто происходило обращение к каждой странице. При страничном прерывании для замещения выбирается страница с наименьшим значением счетчика.

Основная проблема, возникающая при работе с алгоритмом NFU, заключается в том, что он никогда ничего не забывает. Например, в многоходовом компиляторе страницы, которые часто использовались во время первого прохода, могут все еще иметь высокое значение счетчика при более поздних проходах. Фактически, если случается так, что первый проход занимает самое долгое время выполнения из всех, страницы, содержащие программный код для следующих проходов, могут всегда иметь более низкое значение счетчика, чем страницы первого прохода. Следовательно, операционная система удалит полезные страницы вместо тех, которые больше не нужны.

все страницы ь памяти. Бит *R* каждой страницы (он равен 0 или 1) прибавляется к счетчику. В сущности, счетчики пытаются отследить, как часто происходило обращение к каждой странице. При страничном прерывании для замещения выбирается страница с наименьшим значением счетчика.

Основная проблема, возникающая при работе с алгоритмом NFU, заключается в том, что он никогда ничего не забывает. Например, в многоходовом компиляторе страницы, которые часто использовались во время первого прохода, могут все еще иметь высокое значение счетчика при более поздних проходах. Фактически, если случается так, что первый проход занимает самое долгое время выполнения из всех, страницы, содержащие программный код для следующих проходов, могут всегда иметь более низкое значение счетчика, чем страницы первого прохода. Следовательно, операционная система удалит полезные страницы вместо тех, которые больше не нужны.

К счастью, небольшая доработка алгоритма NFU делает его способным моделировать алгоритм LRU достаточно хорошо. Изменение состоит из двух частей. Во-первых, каждый счетчик сдвигается вправо на один разряд перед прибавлением бита *R*. Во-вторых, бит *R* добавляется в крайний слева, а не в крайний справа бит счетчика.

На рис. 4.18 продемонстрировано, как работает видоизмененный алгоритм, известный под названием **«старение»** (aging). Предположим, что после первого тика часов биты *R* для страниц от 0 до 5 имеют значения 1, 0, 1, 0, 1, 1 соответственно (у страницы 0 бит *R* равен 1, у страницы 1 $R = 0$, у страницы 2 $R = 1$ и т. д.). Другими словами, между тиком 0 и тиком 1 произошло обращение к страницам 0, 2, 4 и 5, их биты *R* приняли значение 1, остальные сохранили значение 0. После того как шесть соответствующих счетчиков сдвинулись на разряд и бит *R* занял крайнюю слева позицию, счетчики получили значения, показанные на рис. 4.18, *а*. Остальные четыре колонки рисунка изображают шесть счетчиков после следующих четырех тиков часов.

Когда происходит страничное прерывание, удаляется та страница, чей счетчик имеет наименьшую величину. Ясно, что счетчик страницы, к которой не было обращений, скажем, за четыре тика, будет начинаться с четырех нулей и, таким образом, иметь более низкое значение, чем счетчик страницы, на которую не ссылались в течение только трех тиков часов.

Эта схема отличается от алгоритма LRU в двух случаях. Рассмотрим страницы 3 и 5 на рис. 4.18, *д*. Ни к одной из них не было обращений за последние два тика, к обеим было обращение за предшествующий этому тик. Следуя алгоритму LRU, при удалении страницы из памяти мы должны выбрать одну из этих двух. Проблема в том, что мы не знаем, к какой из них позже произошло обращение в интервале времени между тиками 1 и 2. Записывая только один бит за промежуток времени, мы теряем возможность отличить более ранние от более поздних обращений в этом интервале времени. Все, что мы можем сделать — это выгрузить страницу 3, потому что к странице 5 также обращались двумя тиками раньше, а к странице 3 — нет.

Второе отличие между алгоритмами LRU и «старения» заключается в том, что в последнем счетчик имеет конечное число разрядов, например 8. Предположим, что каждая из двух страниц имеет значение счетчика, равное 0. В данной ситуации мы только случайным образом можем выбрать одну из них. На самом деле

Алгоритм NRU — не использовавшаяся в последнее время страница

Чтобы дать возможность операционной системе собирать полезные статистические данные о том, какие страницы используются, а какие — нет, большинство компьютеров с виртуальной памятью поддерживают два статусных бита, связанных с каждой страницей. Бит *R* (Referenced — обращения) устанавливается всякий раз, когда происходит обращение к странице (чтение или запись). Бит *M* (Modified — изменение) устанавливается, когда страница записывается (то есть изменяется). Биты содержатся в каждом элементе таблицы страниц, как показано на рис. 4.13. Важно реализовать обновление этих битов при каждом обращении к памяти, поэтому необходимо, чтобы они задавались аппаратно. Если однажды бит был установлен в 1, то он остается равным 1 до тех пор, пока операционная система программно не вернет его в состояние 0.

Если аппаратное обеспечение не поддерживает эти биты, их можно смоделировать следующим образом. Когда процесс запускается, все его записи в таблице страниц помечаются как отсутствующие в памяти. Как только происходит обращение к странице, происходит страничное прерывание. Затем операционная система устанавливает бит *R* (в своих внутренних таблицах); изменяет запись в таблице страниц, чтобы она указывала на корректную страницу с режимом READ ONLY (только для чтения), и перезапускает команду. Если страница позднее записывается, происходит другое страничное прерывание, позволяющее операционной системе установить бит *M* и изменить состояние страницы на READ/WRITE (чтение/запись).

Биты *R* и *M* могут использоваться для построения простого алгоритма замещения страниц, описанного ниже. Когда процесс запускается, оба страничных бита для всех его страниц операционной системой установлены на 0. Периодически (например, при каждом прерывании по таймеру) бит *R* очищается, чтобы отличить страницы, к которым давно не происходило обращения от тех, на которые были ссылки.

Когда возникает страничное прерывание, операционная система проверяет все страницы и делит их на четыре категории на основании текущих значений битов *R* и *M*:

Класс 0: не было обращений и изменений.
Класс 1: не было обращений, страница изменена.
Класс 2: было обращение, страница не изменена.
Класс 3: произошло и обращение, и изменение.

Хотя класс 1 на первый взгляд кажется невозможным, такое случается, когда у страницы из класса 3 бит *R* сбрасывается во время прерывания по таймеру. Прерывания по таймеру не стирают бит *M*, потому что эта информация необходима для того, чтобы знать, нужно ли переписывать страницу на диске или нет. Поэтому если бит *R* устанавливается на ноль, а *M* остается нетронутым, страница попадает в класс 1.

Алгоритм **NRU** (Not Recently Used — не использовавшийся в последнее время) удаляет страницу с помощью случайного поиска в непустом классе с наименьшим номером. В этом алгоритме подразумевается, что лучше выгрузить измененную страницу, к которой не было обращений по крайней мере в течение одного тика системных часов (обычно 20 мс), чем стереть часто используемую страницу. Привлекательность алгоритма NRU заключается в том, что он легок для понимания, умеренно сложен в реализации и дает производительность, которая, конечно, не оптимальна, но может вполне оказаться достаточной.

Алгоритм WSClock

Исходный алгоритм «рабочий набор» громоздок, так как при каждом страничном прерывании следует проверять таблицу страниц до тех пор, пока не определится местоположение подходящего кандидата. Усовершенствованный алгоритм, основанный на часовом алгоритме, но также использующий информацию рабочего набора, называется **WSClock** [54]. Благодаря простоте реализации и хорошей производительности этот алгоритм широко используется на практике.

Для него необходима структура данных в виде кольцевого списка страничных блоков, как в алгоритме «часы», что изображено на рис. 4.21, *а*. В исходном положении этот список пустой. Когда загружается первая страница, она добавляется в список. По мере прихода страниц они поступают в список, формируя кольцо.

Каждая запись, кроме бита *R* (показан) и бита *M* (не показан), содержит поле «*время последнего использования*» из базового алгоритма «рабочий набор».

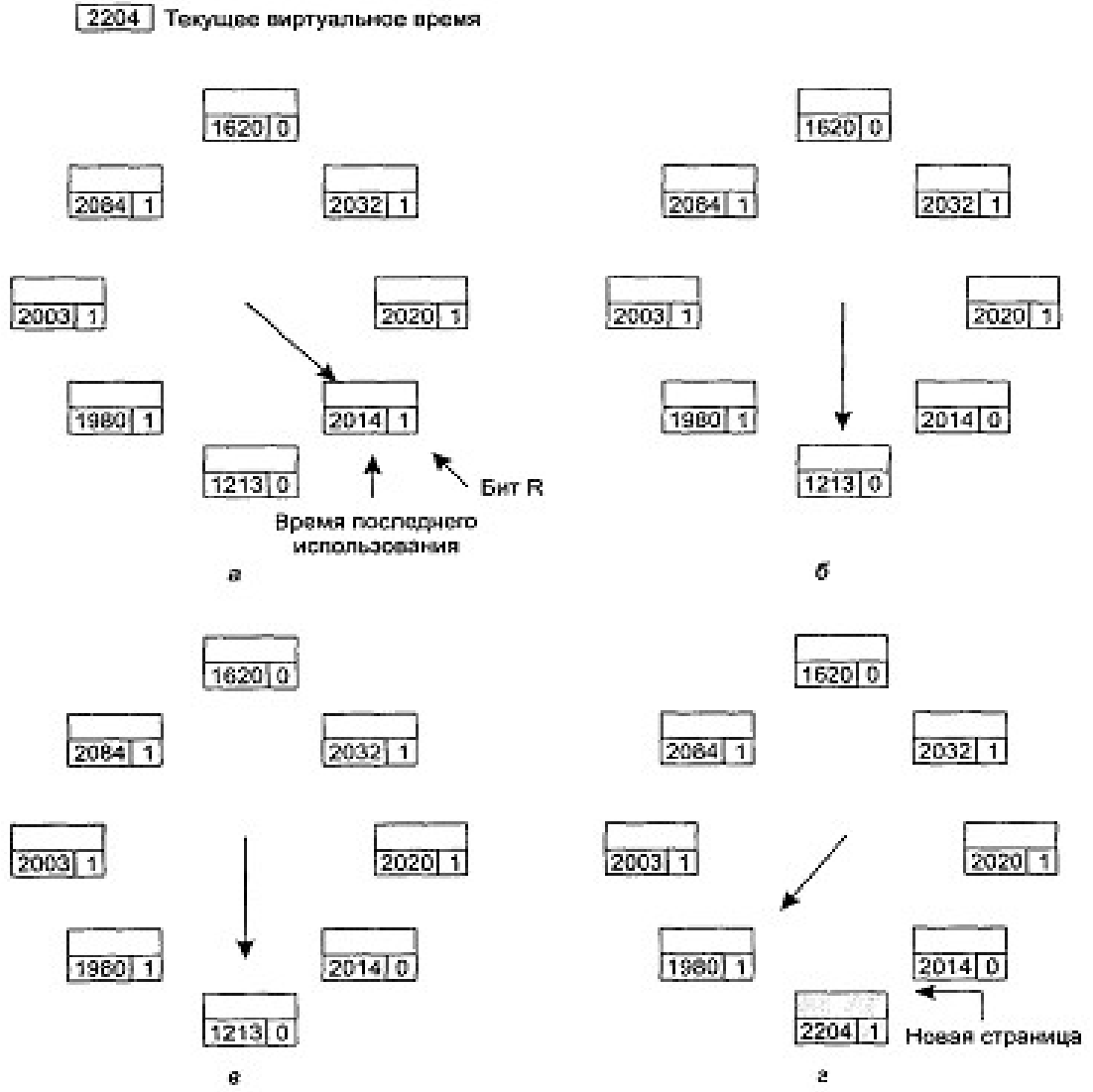

Рис. 4.21. Работа алгоритма WSClock: пример того, что происходит при бите *R* = 1 (*а*) и (*б*); пример для бита *R* = 0 (*в*) и (*г*)

Как и в случае алгоритма «часы», при каждом страничном прерывании первой проверяется та страница, на которую указывает стрелка. Если бит *R* равен 1, это значит, что страница использовалась в течение последнего такта часов, поэтому она не является идеальным кандидатом на удаление. Тогда бит *R* устанавливается на 0, стрелка передвигается на следующую страницу и для нее повторяется алгоритм. Состояние после такой последовательности действий продемонстрировано на рис. 4.21, *б*.

Теперь рассмотрим, что происходит, если страница, на которую указывает стрелка, имеет бит $R = 0$, как показано на рис. 4.21, *а*. Если возраст страницы больше величины τ и страница — чистая, то она не входит в рабочий набор и на диске есть ее действительная копия. Тогда в данный страничный блок просто загружается новая страница, как изображено на рис. 4.21, *г*. Если, напротив, страница «грязная», ее нельзя немедленно стереть, так как на диске нет ее последней копии. Чтобы избежать переключения процессов, запись на диск включается в график планирования, но стрелка сдвигается на позицию, и алгоритм продолжает работу со следующей страницей. Несмотря на то что «грязная» страница может быть старше, чистая находится ближе в ряду страниц, которые можно использовать немедленно.

Теоретически за один обход вокруг циферблата часов для всех страниц может оказаться запланированным ввод-вывод с диска. Чтобы уменьшить поток обмена с диском, можно установить предел, позволяющий быть записанными максимум n страницам. После достижения этой границы новые операции записи перестают включаться в график.

Что происходит, если стрелка обходит целый круг и возвращается к начальной точке? Существует два варианта:

1. Запланирована, по крайней мере, одна операция записи на диск.
2. Ни одной операции записи не запланировано.

В первом случае стрелка продолжает движение, отыскивая чистую страницу. Так как запланирована одна или больше операций записи на диск, со временем какая-нибудь из них будет выполнена, и соответствующая страница будет помечена как чистая. Выгружается первая попавшаяся чистая страница. Это не обязательно та страница, запись которой запланирована первой, потому что драйвер диска может изменить порядок работы с диском, чтобы оптимизировать его производительность.

Во втором случае все страницы находятся в рабочем наборе, иначе планировалась бы, по крайней мере, одна операция записи. За недостатком дополнительной информации проще всего предъявить права на любую чистую страницу и использовать ее. Расположение чистой страницы могло бы отслеживаться во время «чистки». Если в памяти нет чистых страниц, тогда выбирается текущая страница и переписывается на диск.

4) Методы обеспечения надежности файловых систем.

Создание резервных копий занимает много времени и требует много места, поэтому особенное значение при этом получают эффективность и удобство. Во-первых, следует ли архивировать всю файловую систему или только ее часть? Многие операционные системы хранят двоичные файлы в отдельных каталогах. Эти файлы не обязательно архивировать, так как они могут быть переустановлены с CD-ROM производителя. Кроме того, большинство систем имеет каталог для временных файлов. В их архивировании также нет необходимости. В системе UNIX все специальные файлы (устройств ввода-вывода) хранятся в каталоге */dev*. Создавать резервную копию этого каталога не только ненужно, но и опасно, так как программа архивации может навсегда повиснуть, если попытается читать эти файлы. Поэтому обычно создаются резервные копии отдельных каталогов, а не всей файловой системы.

Во-вторых, архивировать не изменившиеся с момента последней архивации файлы неэффективно, поэтому на практике применяется идея **инкрементных резервных копий**, или, как их еще называют, инкрементных дампов. Простейшая форма инкрементного архивирования состоит в том, что полная резервная копия создается, скажем, раз в неделю или раз в месяц, а ежедневно сохраняются только те файлы, которые изменились с момента последней полной архивации. Еще лучше архивировать только те файлы, которые изменились с момента последней архивации. Такая схема сокращает время создания резервных копий, но усложняет процесс восстановления, так как для этого нужно сначала восстановить файлы последней полной архивации, а затем проделать ту же процедуру со всеми инкрементными резервными копиями. Чтобы упростить восстановление, часто применяются более сложные схемы инкрементной архивации.

В-третьих, поскольку объем архивируемых данных обычно очень велик, желательно сжимать эти данные до записи на магнитную ленту. Однако многие алгоритмы сжатия данных устроены так, что малейший дефект ленты может привести к нечитаемости всего файла или даже всей ленты. Поэтому следует хорошенько подумать, прежде чем принимать решение о сжатии архивируемых данных.

В-четвертых, создание резервной копии сложно выполнять в активной файловой системе. Если во время архивации создаются, удаляются и изменяются файлы и каталоги, то информация в создаваемом архиве может оказаться противоречивой. В то же время, поскольку создание резервной копии может занять несколько часов, для этого может понадобиться отключение системы на большую часть ночи, что не всегда приемлемо. По этой причине были разработаны алгоритмы, способные быстро фиксировать состояние файловой системы для ее архивации, копируя критические структуры данных. Последующие изменения файлов и каталогов требовали вместо их замены дополнительного копирования отдельных блоков [160]. Таким образом, файловая система как бы замораживается в момент фиксации и может архивироваться позднее.

Наконец, в-пятых, создание резервных копий создает множество нетехнических проблем. Лучшая система безопасности, охраняющая информацию, может оказаться бесполезной, если системный администратор хранит все свои магнитные ленты с резервными копиями в неохраняемом помещении, которое еще и оставляет открытым. Все, что нужно сделать шпиону, это заскочить на секунду в комнату,

положить кассету в карман и не спеша покинуть здание. Прощай, безопасность. Кроме того, ежедневная архивация принесет мало пользы, если при пожаре погибнут не только компьютеры, но и ленты с резервными копиями. По этой причине резервные копии следует хранить в удаленном месте, в результате чего поддерживать безопасность на достаточно высоком уровне становится еще сложнее. Детальное обсуждение этого и других административных вопросов приводится в [244]. Ниже мы обсудим только технические аспекты создания резервных копий файловой системы.

Для создания резервной копии диска на ленте существует две стратегии: физическая архивация и логическая архивация. **Физическая архивация** состоит в поблочном копировании на магнитную ленту всего диска с блока 0 по последний блок. Эта программа настолько проста, что, возможно, она даже может быть полностью отлажена, чего нельзя сказать об остальных полезных программах.

И все же о физической архивации следует сказать несколько слов. Во-первых, в копировании неиспользуемых блоков диска мало пользы. Если программа архивации сможет получить доступ к структуре данных, хранящей информацию о свободных и занятых блоках диска, она может избежать копирования неиспользуемых блоков. Однако тогда придется перед каждым блоком записывать его номер.

Вторая проблема возникает при столкновении программы архивации с дефектными блоками диска. Если все дефектные блоки контроллер автоматически заменяет запасными блоками, как было описано в разделе «Обработка ошибок» главы 5, тогда физическая архивация работает прекрасно. Однако если дефектные блоки видны операционной системе, которая учитывает их в одном или нескольких специальных файлах или битовых массивах, то абсолютно необходимо, чтобы программа архивации могла получить доступ к этой информации во избежание ошибок чтения диска.

Главное преимущество физической архивации состоит в ее простоте и высокой скорости (обычно она может работать со скоростью диска). Основными ее недостатками являются неспособность пропускать определенные каталоги, производить инкрементную архивацию и восстанавливать отдельные файлы. По этим причинам в большинстве систем применяется логическая архивация.

Логическая архивация сканирует один или несколько указанных каталогов со всеми их подкаталогами и копирует на ленту все содержащиеся в них файлы и каталоги, изменившиеся с указанной даты (например, с момента последней архивации). Таким образом, при логической архивации на магнитную ленту записываются последовательности детально идентифицированных каталогов и файлов, что позволяет восстановить отдельный файл или каталог.

Поскольку логическая архивация применяется на практике чаще физической, познакомимся более детально с алгоритмом архивации на примере, показанном на рис. 6.21. Этот алгоритм используется в большинстве систем UNIX. На рисунке показано дерево файлов с каталогами (квадраты) и файлами (кружки). Затененные элементы были изменены с момента последней архивации и поэтому следует создать их резервную копию. Светлые элементы не нуждаются в архивации. Каждый каталог и файл помечены номером своего i-узла.

Непротиворечивость файловой системы

Еще одним аспектом, относящимся к проблеме надежности, является непротиворечивость файловой системы. Файловые системы обычно читают блоки данных, модифицируют их и записывают обратно. Если в системе произойдет сбой прежде, чем все модифицированные блоки будут записаны на диск, файловая система может оказаться в противоречивом состоянии. Эта проблема становится особенно важной в случае, если одним из модифицированных и не сохраненных блоков оказывается блок i-узла, каталога или списка свободных блоков.

Для решения проблемы противоречивости файловой системы на большинстве компьютеров имеется специальная обслуживающая программа, проверяющая непротиворечивость файловой системы. Например, в системе UNIX такой программой является *fsck*, а в системе Windows это программа *scandisk*. Эта программа может быть запущена сразу после загрузки системы, особенно если до этого произошел сбой. Ниже будет описано, как работает утилита *fsck*. Утилита *scandisk* несколько отличается от *fsck*, поскольку работает в другой файловой системе, однако для нее также остается верным принцип использования избыточной информации для восстановления файловой системы. Все программы проверки файловой системы проверяют различные файловые системы (дисковые разделы) независимо друг от друга.

Существует два типа проверки непротиворечивости: блоков и файлов. При проверке непротиворечивости блоков программа создает две таблицы, каждая из которых содержит счетчик для каждого блока, изначально установленный на 0. Счетчики в первой таблице учитывают, сколько раз каждый блок присутствует в файле. Счетчики во второй таблице записывают, сколько раз каждый блок учитывается в списке свободных блоков (или в битовом массиве свободных блоков).

Затем программа считывает все i-узлы. Начиная с i-узла, можно построить список всех номеров блоков, используемых в соответствующем файле. При считывании каждого номера блока соответствующий ему счетчик увеличивается на единицу. Затем программа анализирует список или битовый массив свободных блоков, чтобы обнаружить все неиспользуемые блоки. Каждый раз, встречая номер блока в списке свободных блоков, программа увеличивает на единицу соответствующий счетчик во второй таблице.

Если файловая система непротиворечива, то каждый блок будет встречаться только один раз, либо в первой, либо во второй таблице, как показано на рис. 6.23, *а*. Однако в результате сбоя эти таблицы могут принять вид, показанный на рис. 6.23, *б*. В этом случае блок два отсутствует в каждой таблице. О таком блоке программа сообщит как о **недостающем блоке**. Хотя пропавшие блоки не причиняют вреда, они занимают место на диске, снижая его емкость. Решить проблему пропавших блоков очень просто: программа проверки файловой системы просто добавляет эти блоки к списку свободных блоков.

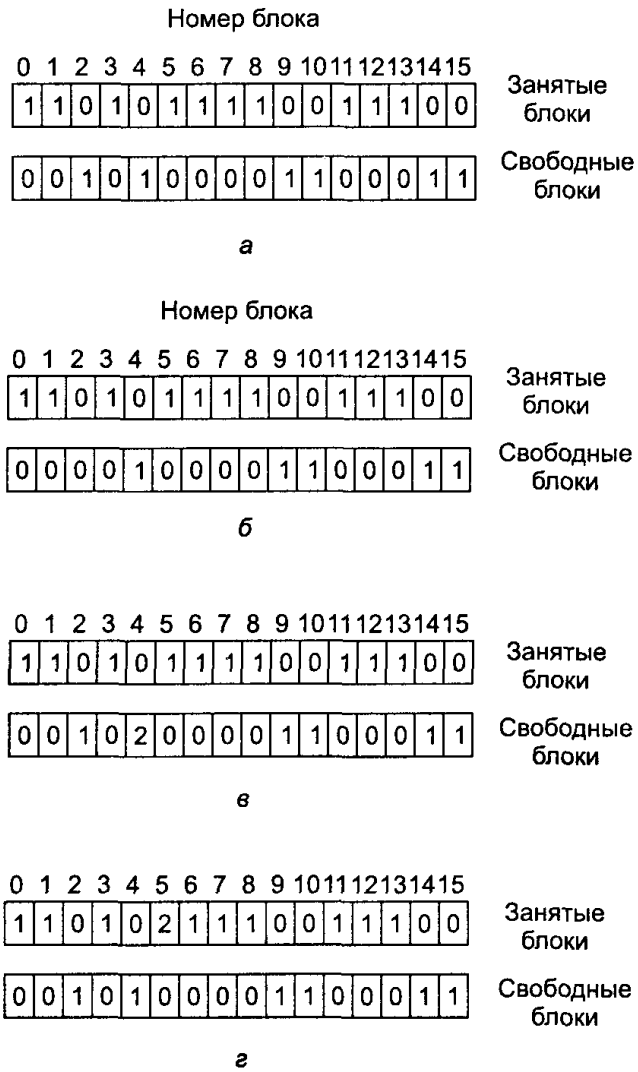

Рис. 6.23. Состояния файловой системы: непротиворечивое (*а*); пропавший блок (*б*); дубликат блока в списке свободных блоков (*в*); дубликат блока данных (*г*)

Ответ:

Доп.инфа:

5) Этапы входа в прерывающую программу.

Вход в прерывающую программу – ХЗ, другого не нашол, походу это именно то, что нада

Основная функция – формирование начального адреса прерывающей программы. Любому запросу соответствует своя прерывающая программа. Существует три различных способа, используемых при формировании адреса:

Код приоритета

запроса

Перераспределены приоритеты

1) **Размещение прерывающих программ по фиксированным адресам.** В некоторой постоянно распределенной области основной памяти по фиксированным адресам размещаются прерывающие программы, это размещение не меняется. Вход реализуется аппаратно, то есть адрес формируется аппаратно – это самый быстрый способ. Но у данного способа существуют серьезные ограничения:

a) привязка к адресам

b) количество причин прерывания должно быть достаточно малым

Этот способ применяется при малых системах прерывания и для тех причин, которые требуют немедленной реакции.

2) вход на основании слов состояния программы (PSW). Типичная структура слова состояния программы:

Маска прерывания Ключ защиты памяти Код состояния CPU Адрес команды (пр-мы)

Схема входа в прерывающую программу. В некоторую постоянно распределенной области основной памяти формируется два массива: массив старых PSW и массив новых PSW. Любая пара слов состояния соответствует определенному запросу на прерывание. После выполнения активного запроса по соответствующему адресу в массив старых PSW загружается PSW текущей (прерываемой) программы. Характеристика программы в виде PSW записывается по определенному адресу. Из второго массива загружается новое (соответствующее прерывающей программе) PSW. Адрес записан в памяти – из PSW. Массив новых PSW всегда формируется при загрузке ОС. Массив старых PSW формируется в процессе работы. В отличие от предыдущего способа, использование PSW позволяет обслуживать и вложенные прерывания, если их приоритет выше текущей программы. Все это позволяет прерывать прерывающую программу. Недостаток: Вход в прерывающую программу требует загрузки достаточно больших слов (большого

процессорного времени), следовательно, данный способ не очень быстрый. Этот вариант используется в универсальных компьютерах (для решения расчетных задач, т.е. не критичных ко времени)

3) Векторное прерывание – самый распространенный способ. Данный способ является программно-аппартным, т.е. для любого запроса (для любого выделенного запроса) аппаратно формируется адрес вектора прерывания. Чаще всего эти адреса фиксированы. Адреса векторных прерываний хранятся в системной области памяти. На основе адреса вектора из таблицы векторов прерывания извлекается начальный адрес прерывающей программы. Это приводит к тому, что код запроса может быть малобитным, но таблица векторов прерывания должна хранится в начальной области памяти. В качестве вектора прерывания используются:

a) адрес начала прерывающей программы (применяется в PC)

b) команда безусловного перехода к программе

Способ a) оказывается универсальным и допускает любую глубину прерывающих программ. Если таблица векторов является загружаемой, то это дает возможность пользователю изменять прерывающие программы даже в процессе решения задачи, которые обрабатывают один и тот же запрос.

Запоминание состояния прерванной программы.

Вся запоминаемая информация делится на основную и дополнительную. Основная информация должна запоминаться всегда – адрес текущей программы, в которой произошло прерывание, состояние процессора, уровень приоритетности программы. Основная информация компонуется в слово-состояние. Основная информация запоминается аппаратно. Дополнительную информацию запоминает сам пользователь. При запоминании основной информации используются два способа:

1) Использование PSW (запоминание старого PSW – основная информация).

Маска прерывания Ключ защиты памяти Код состояния CPU Адрес команды (пр-мы)

2) Запоминание основной информации в системном стеке, который поддерживается ОС. Использование стековых структур при входе в прерывающую программу позволяет не ограничивать глубину вложения прерываний. Ограничения только в связи с размерами стека.

Дополнительная информация с точки зрения объема различна. В каждом конкретном случае определяется самостоятельно – ресурсы процессора, которые используются при работе самой прерывающей программы.

Восстановление состояния прерванной программы.

Инвертирование тех действий, которые выполнены при запоминании.

Возврат.

Передача управления в ту точку, где произошло прерывание. Реализуется обычно аппаратно.

Зависит от организации входа в прерывающую программу.