* + 1. **ИС2К Операционные системы**

**3.4. Алгоритмы замещения страниц**

При возникновении ошибки отсутствия страницы операционная система должна выбрать выселяемую (удаляемую из памяти) страницу, чтобы освободить место для загружаемой страницы. Если предназначенная для удаления страница за время своего нахождения в памяти претерпела изменения, она должна быть переписана на диске, чтобы привести дисковую копию в актуальное состояние. Но если страница не изменялась (например, она содержала текст программы), дисковая копия не утратила своей актуальности и перезапись не требуется. Тогда считываемая страница просто пишется поверх выселяемой.

Если бы при каждой ошибке отсутствия страницы можно было выбирать для выселения произвольную страницу, то производительность системы была бы намного выше, если бы выбор падал на редко востребуемую страницу. При удалении интенсивно используемой страницы высока вероятность того, что она в скором времени будет загружена опять, что приведет к лишним издержкам. На выработку алгоритмов замещения страниц было потрачено множество усилий как в теоретической, так и в экспериментальной областях. Далее мы рассмотрим некоторые из наиболее важных алгоритмов.

Следует заметить, что проблема «замещения страниц» имеет место и в других областях проектирования компьютеров. К примеру, у большинства компьютеров имеется более одного кэша памяти, содержащих последние использованные 32- или 64-байтные блоки памяти. При заполнении кэша нужно выбрать удаляемые блоки. Это проблема в точности повторяет проблему замещения страниц, за исключением более короткого времени (все должно быть сделано за несколько наносекунд, а не миллисекунд, как при замещении страниц). Причиной необходимости более короткого времени является то, что ненайденные блоки кэша берутся из оперативной памяти без затрат времени на поиск и без задержек на раскрутку диска.

В качестве второго примера можно взять веб-сервер. В кэше памяти сервера может содержаться некоторое количество часто востребуемых веб-страниц. Но при заполнении кэша памяти и обращении к новой странице должно быть принято решение о том, какую веб-страницу нужно выселить. Здесь используются те же принципы, что и при работе со страницами виртуальной памяти, за исключением того, что веб-страницы, находящиеся в кэше, никогда не подвергаются модификации, поэтому на диске всегда имеется их свежая копия. А в системе, использующей виртуальную память, страницы, находящиеся в оперативной памяти, могут быть как измененными, так и неизмененными.

**Тема Память**

Во всех рассматриваемых далее алгоритмах замещения страниц ставится вполне определенный вопрос: когда возникает необходимость удаления страницы из памяти, должна ли эта страница быть одной из тех, что принадлежат процессу, в работе которого произошла ошибка отсутствия страницы, или это может быть страница, принадлежащая другому процессу? В первом случае мы четко ограничиваем каждый процесс фиксированным количеством используемых страниц, а во втором таких ограничений не накладываем. Возможны оба варианта, а к этому вопросу мы еще вернемся.

1. **Оптимальный алгоритм замещения страниц**

Наилучший алгоритм замещения страниц несложно описать, но совершенно невозможно реализовать. В нем все происходит следующим образом. На момент возникновения ошибки отсутствия страницы в памяти находится определенный набор страниц. К некоторым из этих страниц будет осуществляться обращение буквально из следующих команд (эти команды содержатся на странице). К другим страницам обращения может не быть и через 10, 100 или, возможно, даже 1000 команд. Каждая страница может быть помечена количеством команд, которые должны быть выполнены до первого обращения к странице.

Оптимальный алгоритм замещения страниц гласит, что должна быть удалена страница, имеющая пометку с наибольшим значением. Если какая-то страница не будет использоваться на протяжении 8 млн команд, а другая какая-нибудь страница не будет использоваться на протяжении 6 млн команд, то удаление первой из них приведет к ошибке отсутствия страницы, в результате которой она будет снова выбрана с диска в самом отдаленном будущем. Компьютеры, как и люди, пытаются по возможности максимально отсрочить неприятные события.

Единственной проблемой такого алгоритма является невозможность его реализации. К тому времени, когда произойдет ошибка отсутствия страницы, у операционной системы не будет способа узнать, когда каждая из страниц будет востребована в следующий раз. (Подобная ситуация наблюдалась и ранее, когда мы рассматривали алгоритм планирования, выбирающий сначала самое короткое задание, — как система может определить, какое из заданий самое короткое?) Тем не менее при прогоне программы на симуляторе и отслеживании всех обращений к страницам появляется возможность реализовать оптимальный алгоритм замещения страниц при *втором* прогоне, воспользовавшись информацией об обращении к страницам, собранной во время *первого* прогона.

Таким образом появляется возможность сравнить производительность осуществимых алгоритмов с наилучшим из возможных. Если операционная система достигает производительности, скажем, на 1 % хуже, чем у оптимального алгоритма, то усилия, затраченные на поиски более совершенного алгоритма, дадут не более 1 % улучшения.

Чтобы избежать любой возможной путаницы, следует уяснить, что подобная регистрация обращений к страницам относится только к одной программе, прошедшей оценку, и только при одном вполне определенном наборе входных данных. Таким образом, полученный в результате этого алгоритм замещения страниц относится только к этой конкретной программе и к конкретным входным данным. Хотя этот метод и применяется для оценки алгоритмов замещения страниц, в реальных системах он бесполезен. Далее мы будем рассматривать те алгоритмы, которые *действительно* полезны для реальных систем.

1. **Алгоритм исключения недавно использовавшейся страницы**

Чтобы позволить операционной системе осуществить сбор полезной статистики востребованности страниц, большинство компьютеров, использующих виртуальную память, имеют два бита состояния, *R* и M, связанных с каждой страницей. Бит *R* устанавливается при каждом обращении к странице (при чтении или записи). Бит *M* устанавливается, когда в страницу ведется запись (то есть когда она модифицируется). Эти биты, как показано на рис. 3.11, присутствуют в каждой записи таблицы страниц. Важно усвоить, что эти биты должны обновляться при каждом обращении к памяти, поэтому необходимо, чтобы их значения устанавливались аппаратной частью. После установки бита в 1 он сохраняет это значение до тех пор, пока не будет сброшен операционной системой.

Если у аппаратуры нет таких битов, они должны быть созданы искусственно с помощью механизмов операционной системы ошибки отсутствия страницы и прерывания таймера. При запуске процесса все записи в его таблице страниц помечаются отсутствующими в памяти. Как только произойдет обращение к странице, возникнет ошибка отсутствия страницы. Тогда операционная система устанавливает бит *R* (в своих внутренних таблицах), изменяет запись в таблице страниц, чтобы она указывала на правильную страницу, с режимом доступа только для чтения (READ ONLY), и перезапускает команду. Если впоследствии страница модифицируется, возникает другая ошибка страницы, позволяющая операционной системе установить бит *M* и изменить режим доступа к странице на чтение-запись (READ/WRITE).

Биты *R* и *M* могут использоваться для создания следующего простого алгоритма замещения страниц. При запуске процесса оба страничных бита для всех его страниц устанавливаются операционной системой в 0. Время от времени (например, при каждом прерывании по таймеру) бит *R* сбрасывается, чтобы отличить те страницы, к которым в последнее время не было обращений, от тех, к которым такие обращения были.

При возникновении ошибки отсутствия страницы операционная система просматривает все страницы и на основе текущих значений принадлежащих им битов *R* и *M* делит их на четыре категории:

1. Класс 0: в последнее время не было ни обращений, ни модификаций.
2. Класс 1: обращений в последнее время не было, но страница модифицирована.
3. Класс 2: в последнее время были обращения, но модификаций не было.
4. Класс 3: в последнее время были и обращения, и модификации.

Хотя на первый взгляд страниц класса 1 быть не может, но они появляются в том случае, если у страниц класса 3 бит *R* сбрасывается по прерыванию от таймера. Эти прерывания не сбрасывают бит *M*, поскольку содержащаяся в нем информация необходима для того, чтобы узнать, нужно переписывать страницу, хранящуюся на диске, или нет. Сброс бита *R* без сброса бита *M* и приводит к возникновению страниц класса 1.

Алгоритм исключения недавно использовавшейся страницы (Not Recently Used (**NRU**)) удаляет произвольную страницу, относящуюся к самому низкому непустому классу. В этот алгоритм заложена идея, суть которой в том, что лучше удалить модифицированную страницу, к которой не было обращений по крайней мере за последний такт системных часов (обычно это время составляет около 20 мс), чем удалить интен

сивно используемую страницу. Главная привлекательность алгоритма NRU в том, что его нетрудно понять, сравнительно просто реализовать и добиться от него производительности, которая, конечно, не оптимальна, но может быть вполне приемлема.

1. **Алгоритм «первой пришла, первой и ушла»**

Другим низкозатратным алгоритмом замещения страниц является алгоритм FIFO (First In, First Out — «первым пришел, первым ушел»). Чтобы проиллюстрировать его работу, рассмотрим супермаркет, у которого вполне достаточно полок для представления как раз *k* различных товаров. И вот однажды какая-то компания представляет новый удобный продукт — быстрорастворимый, полученный в результате сублимационной сушки натуральный йогурт, который может быть восстановлен в микроволновой печи. Он сразу же приобретает популярность, поэтому наш забитый под завязку супермаркет должен избавиться от одного старого продукта, чтобы запастись новым.

Можно, конечно, найти самый залежалый товар (то есть что-нибудь, чем торгуют уже лет сто двадцать) и избавиться от него на том основании, что им уже больше никто не интересуется. В реальности супермаркет ведет связанный список всех продуктов, имеющихся на текущий момент в продаже, в порядке их поступления. Новый продукт попадает в конец списка, а продукт из самого начала списка удаляется.

Для алгоритма замещения страниц можно воспользоваться той же идеей. Операционная система ведет список всех страниц, находящихся на данный момент в памяти, причем совсем недавно поступившие находятся в хвосте, поступившие раньше всех — в голове списка. При возникновении ошибки отсутствия страницы удаляется страница, находящаяся в голове списка, а к его хвосту добавляется новая страница. Применительно к магазину принцип FIFO может привести к удалению воска для эпиляции усов, но он также может привести и к удалению муки, соли или масла. Применительно к компьютерам может возникнуть та же проблема: самая старая страница все еще может пригодиться. Поэтому принцип FIFO в чистом виде используется довольно редко.

1. **Алгоритм «второй шанс»**

Простой модификацией алгоритма FIFO, исключающей проблему удаления часто востребуемой страницы, может стать проверка бита *R* самой старой страницы. Если его значение равно нулю, значит, страница не только старая, но и невостребованная, поэтому она тут же удаляется. Если бит *R* имеет значение 1, он сбрасывается, а страница помещается в конец списка страниц и время ее загрузки обновляется, как будто она только что поступила в память. Затем поиск продолжается.

Действие этого алгоритма, названного «**второй шанс**», показано на рис. 3.14. Страницы с *A* по *H* содержатся в связанном списке отсортированными по времени их поступления в память.

Предположим, что ошибка отсутствия страницы возникла на отметке времени 20. Самой старой является страница *A*, время поступления которой соответствует началу процесса и равно 0. Если бит *R* для страницы *A* сброшен, страница либо удаляется из памяти с записью на диск (если она измененная), либо просто удаляется (если она неизмененная). Но если бит *R* установлен, то *A* помещается в конец списка и ее «время загрузки» переключается на текущее (20). Также при этом сбрасывается бит R. А поиск подходящей страницы продолжается со страницы *B.*

**Страница,**

**загруженная первой Последняя**



**а**

**3 7 8 12 14 15 18 20 А трактуется**

**Г^~|—Г^~|—|~Р~|—Г^~|—Г^П—Г^~|—П~П—Г^П загруженнаястраница**

*6*

Рис. 3.14. Действие алгоритма «второй шанс»: *а* — страницы, отсортированные в порядке FIFO;  
*б* — список страниц при возникновении ошибки отсутствия страницы, показателе времени 20  
и установленном в странице *А* бите R; числа над страницами — это время, когда они были за-  
гружены

Алгоритм «второй шанс» занимается поиском ранее загруженной в память страницы, к которой за только что прошедший интервал времени таймера не было обращений. Если обращения были ко всем страницам, то алгоритм «второй шанс» превращается в простой алгоритм FIFO. Представим, в частности, что у всех страниц на рис. 3.14, *а* бит *R* установлен. Операционная система поочередно перемещает страницы в конец списка, очищая бит *R* при каждом добавлении страницы к концу списка. В конце концов она возвращается к странице A, у которой бит *R* теперь уже сброшен. И тогда страница *A* выселяется. Таким образом, алгоритм всегда завершает свою работу.

1. **Алгоритм «часы»**

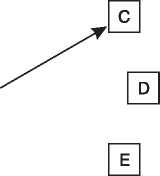
При всей своей логичности алгоритм «второй шанс» слишком неэффективен, поскольку он постоянно перемещает страницы в своем списке. Лучше содержать все страничные блоки в циклическом списке в виде часов (рис. 3.15). Стрелка указывает на самую старую страницу.

**□**

0



□



**Когда происходит страничное  
прерывание, поверяется страница,  
на которую указывает стрелка.  
Предпринимаемые действия  
зависят от бита И:**

**И=0: страница выгружается**

**И=1: бит И сбрасывается, стрелка  
движется вперед**



**0**

Рис. 3.15. Алгоритм «часы;

При возникновении ошибки отсутствия страницы проверяется та страница, на которую указывает стрелка. Если ее бит *R* имеет значение 0, страница выселяется, на ее место в «циферблате» вставляется новая страница и стрелка передвигается вперед на одну позицию. Если значение бита *R* равно 1, то он сбрасывается и стрелка перемещается на следующую страницу. Этот процесс повторяется до тех пор, пока не будет найдена страница с *R* = 0. Неудивительно, что этот алгоритм называется «**часы**».

1. **Алгоритм замещения наименее востребованной страницы**

В основе неплохого приближения к оптимальному алгоритму лежит наблюдение, что страницы, интенсивно используемые несколькими последними командами, будут, скорее всего, снова востребованы следующими несколькими командами. И наоборот, долгое время не востребованные страницы наверняка еще долго так и останутся невостребованными. Эта мысль наталкивает на вполне реализуемый алгоритм: при возникновении ошибки отсутствия страницы нужно избавиться от той страницы, которая длительное время не была востребована. Эта стратегия называется **замещением наименее востребованной страницы** (Least Recently Used (**LRU**)).

Теоретически реализовать алгоритм LRU вполне возможно, но его практическая реализация дается нелегко. Для его полной реализации необходимо вести связанный список всех страниц, находящихся в памяти. В начале этого списка должна быть только что востребованная страница, а в конце — наименее востребованная. Сложность в том, что этот список должен обновляться при каждом обращении к памяти. Для поиска страницы в списке, ее удаления из него и последующего перемещения этой страницы вперед потребуется довольно много времени, даже если это будет возложено на аппаратное обеспечение (если предположить, что такое оборудование можно создать).

Существуют и другие способы реализации LRU с использованием специального оборудования. Сначала рассмотрим самый простой из них. Для его реализации аппаратное обеспечение необходимо оснастить 64-разрядным счетчиком C, значение которого автоматически увеличивается после каждой команды. Кроме этого каждая запись в таблице страниц должна иметь довольно большое поле, чтобы содержать значение этого счетчика. После каждого обращения к памяти текущее значение счетчика *C* сохраняется в записи таблицы страниц, относящейся к той странице, к которой было это обращение. При возникновении ошибки отсутствия страницы операционная система проверяет все значения счетчика в таблице страниц, чтобы найти наименьшее из них. Та страница, к чьей записи относится это значение, и будет наименее востребованной.

1. **Моделирование LRU в программном обеспечении**

При всей принципиальной возможности реализации алгоритма LRU вряд ли найдется машина, обладающая нужным оборудованием. Скорее всего, нам понадобится решение, которое может быть реализовано в программном обеспечении. Одно из возможных решений называется алгоритмом **нечастого востребования** (Not Frequently Used (**NFU**)). Для его реализации потребуется программный счетчик с начальным нулевым значением, связанный с каждой страницей. При каждом прерывании от таймера операционная система сканирует все находящиеся в памяти страницы. Для каждой страницы к счетчику добавляется значение бита *R*, равное 0 или 1. Счетчики позволяют приблизительно отследить частоту обращений к каждой странице. При

возникновении ошибки отсутствия страницы для замещения выбирается та страница, чей счетчик имеет наименьшее значение.

Основная проблема при использовании алгоритма NFU заключается в том, что он похож на слона: никогда ничего не забывает. К примеру, при многопроходной компиляции те страницы, которые интенсивно использовались при первом проходе, могут сохранять высокие значения счетчиков и при последующих проходах. Фактически если на первый проход затрачивается больше времени, чем на все остальные проходы, то страницы, содержащие код для последующих проходов, могут всегда иметь более низкие показатели счетчиков, чем страницы, использовавшиеся при первом проходе. Вследствие этого операционная система будет замещать нужные страницы вместо тех, надобность в которых уже отпала.

К счастью, небольшая модификация алгоритма NFU позволяет довольно близко подойти к имитации алгоритма LRU. Модификация состоит из двух частей. Во-первых, перед добавлением к счетчикам значения бита *R* их значение сдвигается на один разряд вправо. Во-вторых, значение бита *R* добавляется к самому левому, а не к самому правому биту.

На рис. 3.16 показано, как работает модифицированный алгоритм, известный как **алгоритм старения**. Предположим, что после первого прерывания от таймера бит *R* для страниц от 0-й до 5-й имеет значения соответственно 1, 0, 1, 0, 1 и 1 (для страницы 0 оно равно 1, для страницы 1 — 0, для страницы 2 — 1 и т. д.). Иными словами, между прерываниями от таймера, соответствующими тактам 0 и 1, было обращение к страницам 0, 2, 4 и 5, в результате которого их биты *R* были установлены в 1, а у остальных страниц их значение осталось равным 0. После того как были смещены значения шести соответствующих счетчиков и слева вставлено значение бита *R*, они приобрели значения, показанные на рис. 3.16, *а.* В четырех оставшихся столбцах показаны состояния шести счетчиков после следующих четырех прерываний от таймера.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Биты R для страниц 0-5, такт 0 | Биты R для страниц 0-5, такт 1 | Биты R для страниц 0-5, такт 2 | Биты R для страниц 0-5, такт 3 | Биты R для страниц 0-5, такт 4 |
| 10 10 11 | 110 0 10 | 110 10 1 | 1 0 0 0 1 0 | 0 110 0 0 |

Страница

0

1

2

3

4

5

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10000000 |  |  | 11000000 |  |  | 11100000 |  |  | 11110000 |  |  | 01111000 |
|  | |  | | |  | | |  | | |  | |
| 00000000 |  |  | 10000000 |  |  | 11000000 |  |  | 01100000 |  |  | 10110000 |
|  | |  | | |  | | |  | | |  | |
| 10000000 |  |  | 01000000 |  |  | 00100000 |  |  | 00100000 |  |  | 10001000 |
|  | |  | | |  | | |  | | |  | |
| 00000000 |  |  | 00000000 |  |  | 10000000 |  |  | 01000000 |  |  | 00100000 |
|  | |  | | |  | | |  | | |  | |
| 10000000 |  |  | 11000000 |  |  | 01100000 |  |  | 10110000 |  |  | 01011000 |
|  | |  | | |  | | |  | | |  | |
| 10000000 |  |  | 01000000 |  |  | 10100000 |  |  | 01010000 |  |  | 00101000 |
| а | | *б* | | | *в* | | | *г* | | | а | |

Рис. 3.16. Алгоритм старения является программной моделью алгоритма ИЗЫ. Здесь показаны  
шесть страниц в моменты пяти таймерных прерываний, обозначенных буквами от *а* до *д*

При возникновении ошибки отсутствия страницы удаляется та страница, чей счетчик имеет наименьшее значение. Очевидно, что в счетчике страницы, к которой не было обращений за, скажем, четыре прерывания от таймера, будет четыре ведущих нуля, и поэтому значение ее счетчика будет меньшим, чем счетчика страницы, к которой не было обращений в течение трех прерываний от таймера.

Этот алгоритм отличается от алгоритма LRU двумя особенностями. Рассмотрим страницы 3 и 5 на рис. 3.16, *д.* Ни к одной из них за два прерывания от таймера не было ни одного обращения, но к обеим было обращение за прерывание от таймера, предшествующее этим двум. В соответствии с алгоритмом LRU, если страница должна быть удалена, то мы должны выбрать одну из этих двух страниц. Проблема в том, что мы не знаем, к какой из них обращались в последнюю очередь между тактом 1 и тактом 2. При записи только одного бита за интервал между двумя прерываниями от таймера мы утратили возможность отличить более раннее обращение от более позднего. Все, что мы можем сделать, — удалить страницу 3, поскольку к странице 5 также было обращение двумя тактами ранее, а к странице 3 такого обращения не было.

Второе различие между алгоритмом LRU и алгоритмом старения заключается в том, что в алгоритме старения счетчик имеет ограниченное количество бит (в данном примере — 8 бит), которое сужает просматриваемый им горизонт прошлого. Предположим, что у каждой из двух страниц значение счетчика равно нулю. Все, что мы можем сделать, — это выбрать одну из них произвольным образом. В действительности вполне может оказаться, что к одной из этих страниц последнее обращение было 9 тактов назад, а ко второй — 1000 тактов назад. И это обстоятельство установить невозможно. Но на практике 8 бит вполне достаточно, если между прерываниями от таймера проходит примерно 20 мс. Если к странице не было обращений в течение 160 мс, то она, наверное, уже не так важна.

1. **Алгоритм «рабочий набор»**

При использовании замещения страниц в простейшей форме процессы начинают свою работу, не имея в памяти вообще никаких страниц. Как только центральный процессор попытается извлечь первую команду, он получает ошибку отсутствия страницы, заставляющую операционную систему ввести в память страницу, содержащую первую команду. Обычно вскоре за этим следуют ошибки отсутствия страниц с глобальными переменными и стеком. Через некоторое время процесс располагает большинством необходимых ему страниц и приступает к работе, сталкиваясь с ошибками отсутствия страниц относительно редко. Эта стратегия называется **замещением страниц по требованию** (demand paging), поскольку страницы загружаются только по мере надобности, а не заранее.

Разумеется, нетрудно написать тестовую программу, систематически читающую все страницы в огромном адресном пространстве, вызывая при этом такое количество ошибок отсутствия страниц, что для их обработки не хватит памяти. К счастью, большинство процессов так не работают. Они применяют **локальность обращений**, означающую, что в течение любой фазы выполнения процесс обращается только к относительно небольшой части своих страниц. К примеру, при каждом проходе многопроходного компилятора обращение идет только к части имеющихся страниц, причем всякий раз к иной.

Набор страниц, который процесс использует в данный момент, известен как **рабочий набор** (Denning, 1968а; Denning, 1980). Если в памяти находится весь рабочий набор, процесс будет работать, не вызывая многочисленных ошибок отсутствия страниц, пока

не перейдет к другой фазе выполнения (например, к следующему проходу компилятора). Если объем доступной памяти слишком мал для размещения всего рабочего набора, процесс вызовет множество ошибок отсутствия страниц и будет работать медленно, поскольку выполнение команды занимает всего несколько наносекунд, а считывание страницы с диска — обычно 10 мс. Если он будет выполнять одну или две команды за 10 мс, то завершение его работы займет целую вечность. О программе, вызывающей ошибку отсутствия страницы через каждые несколько команд, говорят, что она **пробуксовывает** (Denning, 19686).

В многозадачных системах процессы довольно часто сбрасываются на диск (то есть все их страницы удаляются из памяти), чтобы дать возможность другим процессам воспользоваться своей очередью доступа к центральному процессору. Возникает вопрос: что делать, когда процесс возобновляет свою работу? С технической точки зрения ничего делать не нужно. Процесс просто будет вызывать ошибки отсутствия страниц до тех пор, пока не будет загружен его рабочий набор. Проблема в том, что наличие многочисленных ошибок отсутствия страниц при каждой загрузке процесса замедляет работу, а также вызывает пустую трату значительной части рабочего времени центрального процессора, поскольку на обработку операционной системой одной ошибки отсутствия страницы затрачивается несколько миллисекунд процессорного времени.

Поэтому многие системы замещения страниц пытаются отслеживать рабочий набор каждого процесса и обеспечивать его присутствие в памяти, перед тем как позволить процессу возобновить работу. Такой подход называется **моделью рабочего набора**. Он был разработан для существенного сокращения количества ошибок отсутствия страниц. Загрузка страниц *до того,* как процессу будет позволено возобновить работу, называется также **опережающей подкачкой страниц** (prepaging). Следует заметить, что со временем рабочий набор изменяется.

Давно подмечено, что большинство программ неравномерно обращается к своему адресному пространству: их обращения склонны группироваться на небольшом количестве страниц. Обращение к памяти может быть извлечением команды или данных или сохранением данных. В любой момент времени *t* существует некий набор, состоящий из всех страниц, используемый в *k* самых последних обращений к памяти. Этот набор, *w(k,* t), и является рабочим набором. Так как все недавние обращения к памяти для *k* > 1 обязательно должны были обращаться ко всем страницам, использовавшимся для *(k* = 1)-го обращения к памяти, то есть к последней и, возможно, еще к некоторым страницам, *w(k, t)* является монотонно неубывающей функцией от *k.* По мере роста значения *k* значение функции *w(k, t)* достигает своего предела, поскольку программа не может обращаться к количеству страниц, превышающему по объему имеющееся адресное пространство, а каждая отдельно взятая страница будет использоваться лишь немногими программами. На рис. 3.17 показано, что размер рабочего набора является функцией от *k.*

Тот факт, что большинство программ произвольно обращается к небольшому количеству страниц, но со временем этот набор медленно изменяется, объясняет начальный быстрый взлет кривой графика, а затем, при больших значениях *k*, существенное замедление этого взлета. К примеру, программа, выполняющая цикл, при этом занимающая две страницы и использующая данные, расположенные на четырех страницах, может обращаться ко всем шести страницам каждые 1000 команд, но самые последние обращения к некоторым другим страницам могут состояться за 1 млн команд до этого, в процессе фазы инициализации. Благодаря такому асимптотическому поведению содержимое рабочего набора нечувствительно к выбранному значению *k*.

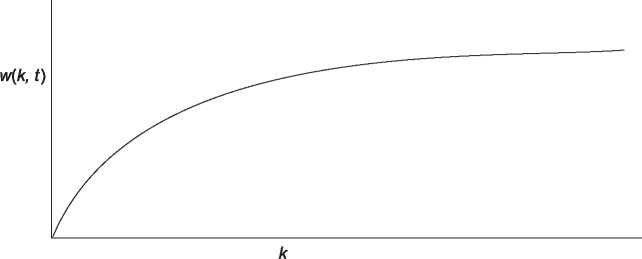


Рис. 3.17. Рабочий набор — это набор страниц, используемых при *к* самых последних  
обращений. Значение функции *w(k,* Г) является размером рабочего набора в момент времени Г

Иначе говоря, существует широкий диапазон значений *к,* для которого рабочий набор остается неизменным. Поскольку со временем рабочий набор изменяется медленно, появляется возможность выстроить разумные предположения о том, какие страницы понадобятся при возобновлении работы программы, основываясь на том, каков был ее рабочий набор при последней приостановке ее работы. Опережающая подкачка страниц как раз и заключается в загрузке этих страниц перед возобновлением процесса.

Для реализации модели рабочего набора необходимо, чтобы операционная система отслеживала, какие именно страницы входят в рабочий набор. При наличии такой информации тут же напрашивается возможный алгоритм замещения страниц: при возникновении ошибки отсутствия страницы нужно выселить ту страницу, которая не относится к рабочему набору. Для реализации подобного алгоритма нам необходим четкий способ определения того, какие именно страницы относятся к рабочему набору. По определению рабочий набор — это набор страниц, используемых в *к* самых последних обращений (некоторые авторы используют термин *«к* самых последних страничных обращений», но это дело вкуса). Для реализации любого алгоритма рабочего набора некоторые значения *к* должны быть выбраны заранее. Затем после каждого обращения к памяти однозначно определяется и набор страниц, используемый при самых последних *к* обращениях к памяти.

Разумеется, это определение рабочего набора не означает наличия эффективного способа его вычисления в процессе выполнения программы. Можно представить себе регистр со сдвигом, имеющий длину *к*, в котором при каждом обращении к памяти его содержимое сдвигается влево на одну позицию и справа вставляется номер страницы, к которой было самое последнее обращение. Набор из всех *к* номеров страниц в регистре со сдвигом и будет представлять собой рабочий набор. Теоретически при возникновении ошибки отсутствия страницы содержимое регистра со сдвигом может быть считано и отсортировано. Затем могут быть удалены продублированные страницы. В результате должен получиться рабочий набор. Но обслуживание регистра со сдвигом и обработка его содержимого при возникновении ошибки отсутствия страницы окажется недопустимо затратным делом, поэтому эта технология никогда не используется.

Вместо нее используются различные приближения. Одно из часто используемых приближений сводится к отказу от идеи вычисления *к* обращений к памяти и использованию вместо этого времени выполнения. Например, вместо определения рабочего набора в качестве страниц, использовавшихся в течение предыдущих 10 млн обращений к па

мяти, мы можем определить его как набор страниц, используемых в течение последних 100 мс времени выполнения. На практике с таким определением работать гораздо лучше и проще. Следует заметить, что для каждого процесса вычисляется только его собственное время выполнения. Таким образом, если процесс запускается во время *T* и получает 40 мс времени центрального процессора за время *T* + 100 мс, то для определения рабочего набора берется время 40 мс. Интервал времени центрального процессора, реально занимаемый процессом с момента его запуска, часто называют **текущим виртуальным временем**. При этом приближении рабочим набором процесса является набор страниц, к которым он обращался в течение последних *t* секунд виртуального времени.

Теперь взглянем на алгоритм замещения страниц, основанный на рабочем наборе. Основной замысел состоит в том, чтобы найти страницу, не принадлежащую рабочему набору, и удалить ее из памяти. На рис. 3.18 показана часть таблицы страниц, используемой в некой машине. Поскольку в качестве кандидатов на выселение рассматриваются только страницы, находящиеся в памяти, страницы, в ней отсутствующие, этим алгоритмом игнорируются. Каждая запись состоит (как минимум) из двух ключевых элементов информации: времени (приблизительного) последнего использования страницы и бита *R* (Referenced — бита обращения). Пустыми белыми прямоугольниками обозначены другие поля, не нужные для этого алгоритма, например номер страничного блока, биты защиты и бит изменения — *M* (Modified).

1. **2204 I Текущее виртуальное время**

**Информация  
об одной странице**

**Время последнего использования '**

**К странице было  
обращения за данный  
такт**

**К странице не было  
обращения за данный  
такт**

**Таблица страниц**

**2084**

**2003**

**-> 1980**

**1213**

**2014**

2020

**2032**

**1620**

**Бит R (обращение)**

**При изучении всех страниц проверяется бит И:**

**если (1Ч==1),**

**текущее виртуальное время запоминается в поле времени последнего использования**

**если (И==0 и возраст > г), удаляется эта страница**

**если (И==0 и возраст < г), запоминается наименьшее время**

Рис. 3.18. Алгоритм рабочего набора

Рассмотрим работу алгоритма. Предполагается, что аппаратура, как было рассмотрено ранее, устанавливает биты *Я* и М. Также предполагается, что периодические прерывания от таймера запускают программу, очищающую бит обращения *Я.* При каждой ошибке отсутствия страницы происходит сканирование таблицы страниц с целью найти страницу, пригодную для удаления.

При каждой обработке записи проверяется состояние бита Я. Если его значение равно 1, текущее виртуальное время записывается в поле времени последнего исполь

зования таблицы страниц, показывая, что страница была использована при возникновении ошибки отсутствия страницы. Если обращение к странице происходит в течение текущего такта времени, становится понятно, что она принадлежит рабочему набору и не является кандидатом на удаление (предполагается, что £ охватывает несколько системных тактов).

Если значение *Я* равно 0, значит, за текущий такт времени обращений к странице не было и она может быть кандидатом на удаление. Чтобы понять, должна ли она быть удалена или нет, вычисляется ее возраст (текущее виртуальное время за вычетом времени последнего использования), который сравнивается со значением £. Если возраст превышает значение £, то страница уже не относится к рабочему набору и заменяется новой страницей. Сканирование продолжается, и происходит обновление всех остальных записей.

Но если значение *Я* равно 0, но возраст меньше или равен £, то страница все еще относится к рабочему набору. Страница временно избегает удаления, но страница с наибольшим возрастом (наименьшим значением времени последнего использования) берется на заметку. Если будет просканирована вся таблица страниц и не будет найдена страница — кандидат на удаление, значит, к рабочему набору относятся все страницы. В таком случае, если найдена одна и более страниц с *Я* = 0, удаляется одна из них, имеющая наибольший возраст. В худшем случае в течение текущего такта было обращение ко всем страницам (и поэтому у всех страниц *Я* = 1), поэтому для удаления одна из них выбирается случайным образом, при этом предпочтение отдается неизмененной странице, если таковая имеется.

**Алгоритм WSClock**

Базовый алгоритм рабочего набора слишком трудоемок, поскольку при возникновении ошибки отсутствия страницы для определения местонахождения подходящего кандидата на удаление необходимо просканировать всю таблицу страниц. Усовершенствованный алгоритм, основанный на алгоритме «часы», но также использующий информацию о рабочем наборе, называется **WSClock** Благодаря простоте реализации и хорошей производительности он довольно широко используется на практике.

Необходимая структура данных сводится к циклическому списку страничных блоков, как в алгоритме «часы» и как показано на рис. 3.19, *а.* Изначально этот список пуст. При загрузке первой страницы она добавляется к списку. По мере загрузки следующих страниц они попадают в список, формируя замкнутое кольцо. В каждой записи содержится поле *времени последнего использования* из базового алгоритма рабочего набора, а также бит *R* (показанный на рисунке) и бит *M* (не показанный на рисунке).

Как и в алгоритме «часы», при каждой ошибке отсутствия страницы сначала проверяется страница, на которую указывает стрелка. Если бит *R* установлен в 1, значит, страница была использована в течение текущего такта, поэтому она не является идеальным кандидатом на удаление. Затем бит *R* устанавливается в 0, стрелка перемещается на следующую страницу, и алгоритм повторяется уже для нее. Состояние, получившееся после этой последовательности событий, показано на рис. 3.19, *б.*

Теперь посмотрим, что получится, если у страницы, на которую указывает стрелка, бит *R* = 0 (рис. 3.19, в). Если ее возраст превышает значение *t* и страница не изме-

**2204 | Текущее виртуальное время**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **1620| 0** |  | **1620|0** |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **2084|1** |  | **2032|1** |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **2084|1** |  | **2032|1** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | |  |
| **2003|1** |  | **2020|1** |  | **20031 1** |  |  | **2020|1** |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **1980|1** |  | **2014| 1** |

**1980| 1**

**2014|0**

**1213| 0**

**! N**

**Бит Р**

**1213| О**

**Время последнего  
использования**

***а 6***

**1620| О**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **20841 1** |  | **20321 1** |

**1620| О**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **2084|1** |  | **2032|1** |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| **2003|1** |  |  | **2020|1** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **1** | **Г** |  |
| **1980|1** |  | | **2014|0** |

**1213| О**

**2003| 1**



2020|1

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **1980| 1** |  | **2014|0** |

**2204|1**

**Новая страница**

***в г***

Рис. 3.19. Работа алгоритма WSClock: *а* и *б* — пример того, что происходит, когда *Я* = 1;  
*в* и *г* — пример того, что происходит, когда *Я* = 0

нена, она не относится к рабочему набору и ее точная копия присутствует на диске. Тогда страничный блок просто истребуется и в него помещается новая страница (рис. 3.19, г). Но если страница изменена, ее блок не может быть тотчас же истребован, поскольку на диске нет ее точной копии. Чтобы избежать переключения процесса, запись на диск планируется, а стрелка перемещается дальше и алгоритм продолжает свою работу на следующей странице. В конце концов должна попасться старая, неизмененная страница, которой можно будет тут же и воспользоваться.

В принципе, за один оборот часовой стрелки может быть запланирована операция  
дискового ввода-вывода для всех страниц. Для уменьшения потока обмена данными

с диском может быть установлен лимит, позволяющий сбрасывать на диск максимум *п* страниц. По достижении этого лимита новые записи на диск планироваться уже не должны.

А что делать, если стрелка пройдет полный круг и вернется в начальную позицию? Тогда следует рассмотреть два варианта.

1. Была запланирована хотя бы одна запись на диск.
2. Не было запланировано ни одной записи на диск.

В первом случае стрелка просто продолжит движение, выискивая неизмененную страницу. Поскольку была запланирована одна или более записей на диск, со временем одна из записей завершится, и задействованная в ней страница будет помечена неизмененной. Первая же неизмененная страница и будет удалена. Эта страница не обязательно должна быть первой запланированной, поскольку драйвер диска может изменить порядок записи, чтобы оптимизировать производительность его работы.

Во втором случае все страницы относятся к рабочему набору, иначе должна была быть запланирована хотя бы одна запись. При недостатке дополнительной информации простейшее, что можно сделать, — истребовать любую неизмененную страницу и воспользоваться ею. Расположение неизмененной страницы может быть отслежено в процессе оборота стрелки. Если неизмененных страниц не имеется, то в качестве жертвы выбирается текущая страница, которая и сбрасывается на диск.

1. **Краткая сравнительная характеристика алгоритмов замещения страниц**

Только что мы рассмотрели несколько различных алгоритмов замещения страниц. В этом разделе дадим им краткую сравнительную характеристику. Список рассмотренных алгоритмов представлен в табл. 3.2.

**Таблица 3.2. Список рассмотренных алгоритмов замещения страниц**

|  |  |
| --- | --- |
| Алгоритм | Особенности |
| Оптимальный | Не может быть реализован, но полезен в качестве оценочного критерия |
| NRU (Not Recently Used) — алгоритм исключения недавно использовавшейся страницы | Является довольно грубым приближением к алгоритму LRU |
| FIFO (First In, First Out) — алгоритм «первой пришла, первой и ушла» | Может выгрузить важные страницы |
| Алгоритм «второй шанс» | Является существенным усовершенствованием алгоритма FIFO |
| Алгоритм «часы» | Вполне реализуемый алгоритм |
| LRU (Least Recently Used) — алгоритм замещения наименее востребованной страницы | Очень хороший, но труднореализуемый во всех тонкостях алгоритм |
| NFU (Not Frequently Used) — алгоритм нечастого востребования | Является довольно грубым приближением к алгоритму LRU |
| Алгоритм старения | Вполне эффективный алгоритм, являющийся неплохим приближением к алгоритму LRU |
| Алгоритм рабочего набора | Весьма затратный для реализации алгоритм |
| WSClock | Вполне эффективный алгоритм |
|  |  |

Оптимальный алгоритм удаляет страницу с самым отдаленным предстоящим обращением. К сожалению, у нас нет способа определения, какая это будет страница, поэтому на практике этот алгоритм использоваться не может. Но он полезен в качестве оценочного критерия при рассмотрении других алгоритмов.

Алгоритм исключения недавно использованной страницы (NRU) делит страницы на четыре класса в зависимости от состояния битов R и M. Затем выбирает произвольную страницу из класса с самым низким номером. Этот алгоритм нетрудно реализовать, но он слишком примитивен. Есть более подходящие алгоритмы.

Алгоритм FIFO предполагает отслеживание порядка, в котором страницы были загружены в память, путем сохранения сведений об этих страницах в связанном списке. Это упрощает удаление самой старой страницы, но она-то как раз и может все еще использоваться, поэтому FIFO — неподходящий выбор.

Алгоритм «второй шанс» является модификацией алгоритма FIFO и перед удалением страницы проверяет, не используется ли она в данный момент. Если страница все еще используется, она остается в памяти. Эта модификация существенно повышает производительность. Алгоритм «часы» является простой разновидностью алгоритма «второй шанс». Он имеет такой же показатель производительности, но требует несколько меньшего времени на свое выполнение.

Алгоритм LRU превосходен во всех отношениях, но не может быть реализован без специального оборудования. Если такое оборудование недоступно, то он не может быть использован. Алгоритм NFU является грубой попыткой приблизиться к алгоритму LRU. Его нельзя признать удачным. А вот алгоритм старения — куда более удачное приближение к алгоритму LRU, которое к тому же может быть эффективно реализовано и считается хорошим выбором.

В двух последних алгоритмах используется рабочий набор. Алгоритм рабочего набора обеспечивает приемлемую производительность, но его реализация обходится слишком дорого. Алгоритм WSClock является вариантом, который не только обеспечивает неплохую производительность, но и может быть эффективно реализован.

В итоге наиболее приемлемыми алгоритмами являются алгоритм старения и алгоритм WSClock. Они основаны на LRU и рабочем наборе соответственно. Оба обеспечивают неплохую производительность страничной организации памяти и могут быть эффективно реализованы. Существует также ряд других хороших алгоритмов