Распределение ресурсов. Управление памятью. Виртуальная память

Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

Дата последней модификации: 10.03.2010 23:45:00

История изменений

09.03.2010 – Версия 0.1. Первичный документ. Ковтун В.Ю.

Содержание

Лекция 5. Распределение ресурсов. Управление памятью. Виртуальная память. Часть 2

Вопросы

1. Виртуальная память (продолжение).
2. Алгоритмы замещения страниц.
3. Выводы.

Виртуальная память

Таблицы страниц

В простейшем случае отображение виртуальных адресов на физические, происходит так, как только что было описано. Виртуальный адрес делится на номер виртуальной страницы (старшие биты) и сдвиг (младшие биты). Например, при 16-разрядных адресах и размере страницы 4 Кбайт старшие 4 бита могут указывать одну из 16 виртуальных страниц, а нижние 12 бит могут определять байт смещения (от 0 до 4095) внутри выбранной страницы. Однако разбиение страницы на 3,5 или какое-нибудь другое число битов также возможно. Разные части подразумевают различные размеры страниц.

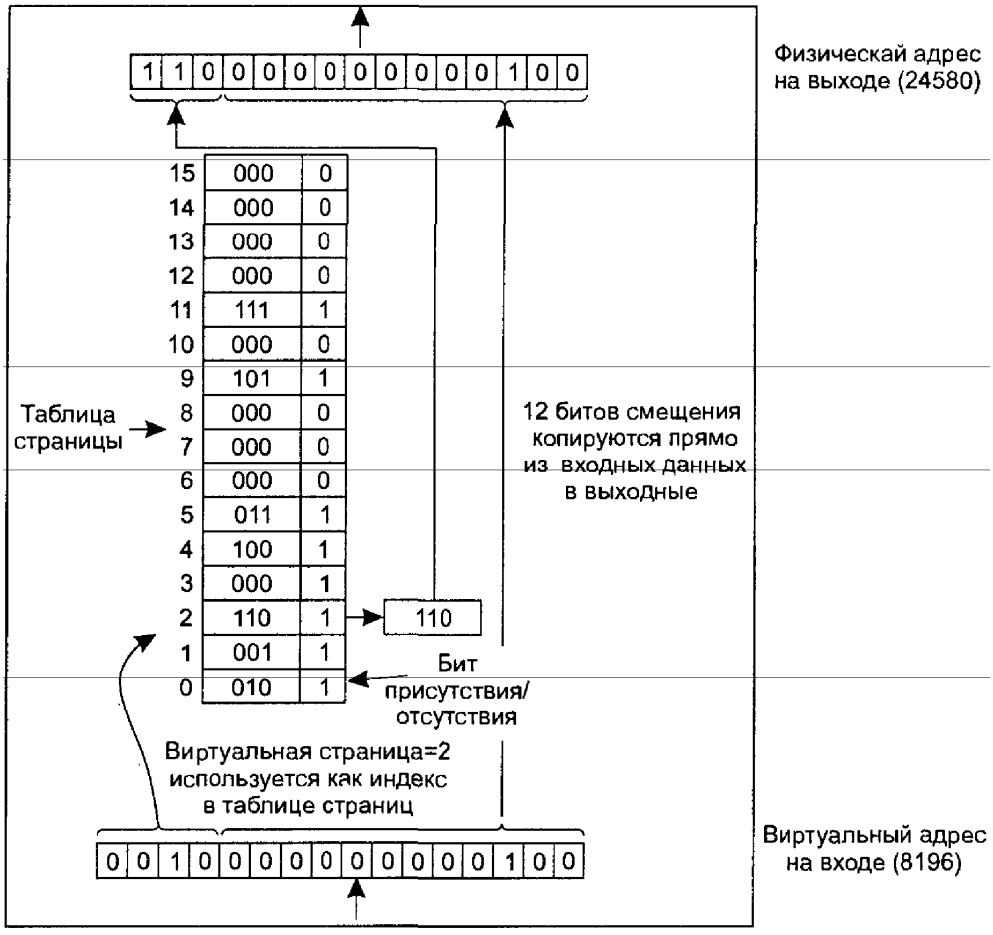

Рис. 1. Внутренняя операция MMU в системе с шестнадцатью страницами размером 4 Кбайт

Номер виртуальной страницы используется как индекс в таблице страниц для поиска записи этой страницы. По записи в таблице страниц находится номер физического блока страницы (если это имеет место). Данный номер присоединяется к старшим разрядам числа смещения, замещая номер виртуальной страницы и тем самым формируя физический адрес, который может быть послан в память.

Назначение таблицы страниц заключается в отображении виртуальных страниц на страничные блоки. Говоря математически, таблица страниц — это функция, имеющая в качестве аргумента номер виртуальной страницы и получающая в результате номер

физического блока. Используя результат действия этой функции, поле виртуальной страницы в виртуальном адресе может быть заменено полем страничного блока, таким образом, формируется физический адрес.

Несмотря на столь простое описание, нам придется столкнуться с двумя важными проблемами:

1. Таблица страниц может быть слишком большой.

2. Отображение должно быть быстрым.

Первый пункт следует из того факта, что современные процессоры используют по крайней мере 32-разрядные виртуальные адреса. При размере страницы, скажем, 4 Кбайт, 32-разрядное адресное пространство будет состоять из одного миллиона страниц, а 64-разрядное адресное пространство будет включать в себя намного больше страниц, чем то количество, с которым вы захотите иметь дело. При одном миллионе страниц в виртуальном адресном пространстве таблица страниц должна состоять из одного миллиона записей. И помните, что каждый процесс нуждается в своей собственной таблице страниц (потому что у него есть свое собственное виртуальное адресное пространство).

Второй пункт — это вывод из того факта, что преобразование виртуальных адресов в физические должно быть выполнено для каждого обращения к ячейке памяти. Типичная команда процессора включает в себя слово-команду и часто также операнд памяти. В результате необходимо сделать 1, 2 или иногда больше обращений к таблице страниц за команду. Если выполнение команды занимает, скажем, 4 нс, то поиск в таблице страниц должен быть сделан меньше, чем за 1 нс, чтобы преобразование виртуальных адресов не стало главным узким местом системы.

Потребность в огромном, но при этом быстром страничном отображении накладывает существенные ограничения на способы построения компьютеров. Хотя проблема наиболее серьезно встает для старших моделей семейства, она также появляется и для младших моделей, когда стоимость и соотношение цена/производительность имеют критическое значение. В этом и следующих разделах мы рассмотрим устройство таблицы страниц в деталях и покажем несколько аппаратных решений, которые использовались в реальных компьютерах.

Простейшее конструкторское решение (по крайней мере, концептуально) заключается в поддержании таблицы страниц, состоящей из массива быстрых аппаратных регистров с одной записью для каждой виртуальной страницы, индексированного по номерам виртуальных страниц, как показано на рис. 1. Когда процесс запускается, ОС загружает в регистры таблицу страниц процесса, данные берутся из копии, хранящейся в ОЗУ. Во время выполнения процесса таблице страниц больше не нужно обращаться к памяти. Преимущество этого метода заключается в его простоте и отсутствии необходимости обращений к памяти во время преобразования адресов. **Недостатком** является его **потенциально высокая стоимость** (если таблица страниц велика). Необходимость загрузки полной таблицы в регистры при каждом контекстном переключении наносит ущерб производительности.

Другая крайность заключается в том, что **таблица страниц целиком располагается в ОЗУ**. Тогда все необходимое оборудование состоит из одного-единственного регистра, указывающего на начало таблицы страниц. Такая схема позволяет изменять карту памяти при контекстном переключении путем перезагрузки только одного регистра. Конечно, она имеет свой недостаток: во время выполнения каждой инструкции программы требуется одно или несколько обращений к памяти для чтения записей таблицы страниц. По этой причине данный метод редко используется в своем чистом виде, но ниже будет рассмотрено несколько его разновидностей, имеющих более высокую производительность.

Многоуровневые таблицы страниц

Чтобы обойти проблему необходимости постоянного хранения в памяти огромных таблиц страниц, многие компьютеры используют **многоуровневую таблицу страниц**. Простой пример представлен на рис. 2. На рис. 2(а) изображен 32-разрядный виртуальный адрес, который разделен на 10-разрядное поле PT1, 10-разрядное поле PT2 и 12-разрядное поле Offset (смещение). Так как под смещение отведено 12 бит, страницы имеют размер 4 Кбайт, и их всего 2^{20}.

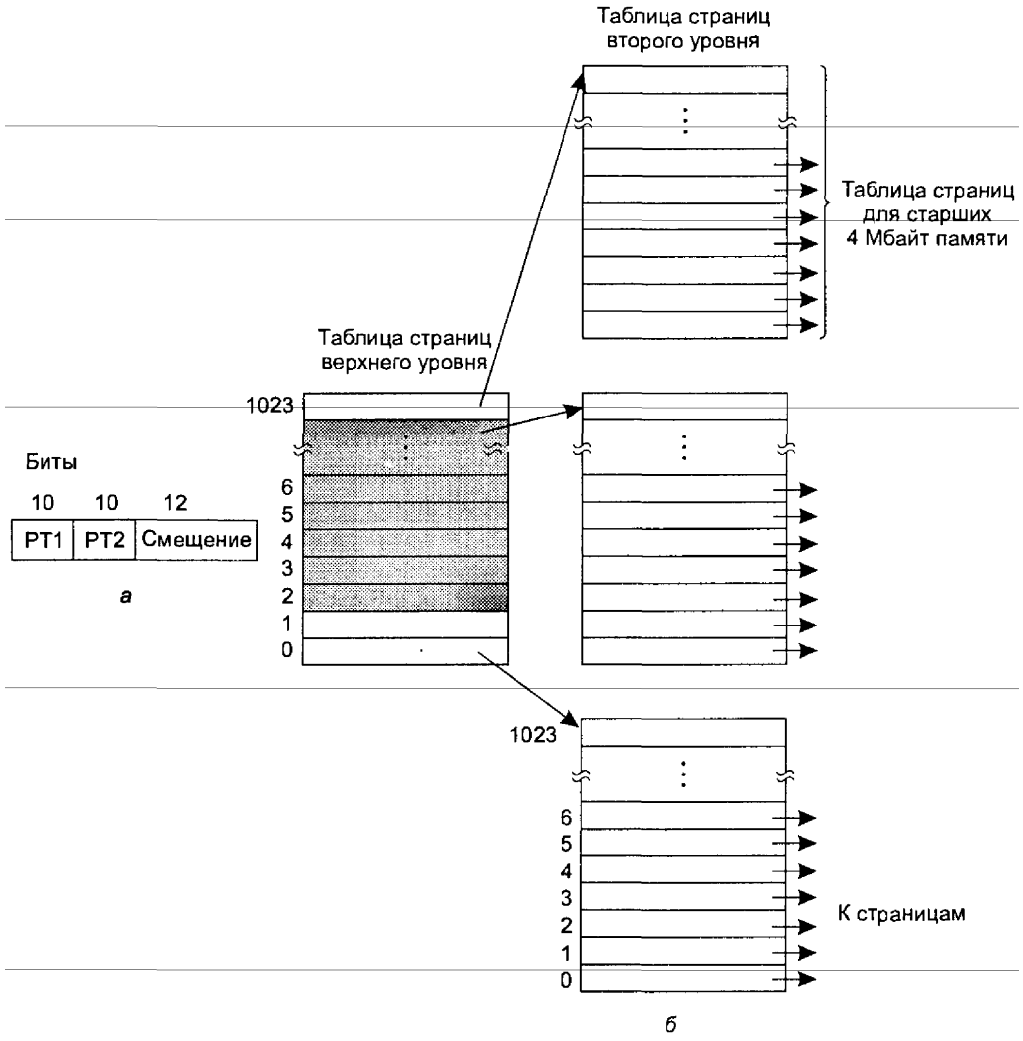

Рис. 2. 32-разрядные адреса с полями двух таблиц страниц (а); двухуровневая таблица страниц (б)

Секрет метода многоуровневой таблицы страниц заключается в том, чтобы избегать постоянного содержания в памяти всех таблиц страниц. В частности, те части, которые **не нужны в данный момент, не должны храниться в памяти**. Предположим, например, что процессу нужно 12 Мбайт, младшие 4 Мбайт памяти для текста программы, следующие 4 Мбайт для данных и старшие 4 Мбайт для стека. Между верхом данных и низом стека образуется гигантский свободный участок, который не используется.

На рис. 2(б) мы видим, как в данном примере работает двухуровневая таблица страниц. Слева находится таблица страниц верхнего уровня с 1024 записями, соответствующими 10-разрядному нолю PT1. Когда виртуальный адрес предстает перед MMU, он сначала выделяет поле PT1 и использует его значение как индекс таблицы верхнего уровня. Каждая из этих 1024 записей представляет 4 М, потому что целое 4-гигабайтное (то есть 32-разрядное) виртуальное адресное пространство было нарезано на куски по 1024 байта.

Запись, место которой определяется по индексу в таблице страниц верхнего уровня, выдает адрес или номер страничного блока таблицы страниц второго уровня. Запись 0 в таблице страниц первого уровня указывает на таблицу страниц для текста программы, запись 1 указывает на таблицу страниц для данных, запись 1023 указывает на таблицу страниц для стека. Другие (заштрихованные) записи не используются. Поле PT2 теперь используется как индекс в выбранной таблице второго уровня для поиска номера страничного блока самой страницы.

В качестве примера рассмотрим 32-разрядный адрес 0x00403004 (4 206 596 в десятичном виде), который соответствует байту 12 292 в данных. У этого виртуального адреса PT1=1, PT2=2 и Offset=4. MMU сначала использует поле РП, чтобы по индексу в таблице страниц верхнего уровня получить запись 1, которая соответствует адресам от 4 до 8 М. Затем он воспользуется полем PT2, чтобы по индексу из только что найденной таблицы второго уровня извлечь запись 3, которая соответствует адресам от 12 288 до

16 383 внутри своего участка размером 4 М (то есть абсолютным адресам от 4 206 592 до 4 210 687). Эта запись содержит номер физического блока страницы, содержащей виртуальный адрес 0x00403004. Если данная страница не находится в памяти, бит Присутствия/Отсутствия в записи таблицы страниц будет равен нулю, что приведет к страничному прерыванию. Если страница в памяти, то номер страничного блока, взятый из таблицы страниц второго уровня, присоединяется к смещению (4), создавая физический адрес. Этот адрес выставляется на шину и передается памяти.

Следует отметить одну интересную деталь на рис. 2. Хотя адресное пространство содержит больше миллиона страниц, фактически нужны только четыре таблицы: таблица верхнего уровня и таблицы нижнего уровня для памяти от 0 до 4 М, от 4 до 8 М и для верхних 4 М. Битам Присутствия/отсутствия для 1021 записи таблицы страниц верхнего уровня присвоено значение 0, что вызовет страничное прерывание при любом обращении к ним. Если это происходит, то ОС заметит, что процесс пытается обратиться к области памяти, не предполагающей ссылок на нее, и предпримет соответствующее действие, например, пошлет ему сигнал или уничтожит его. В описанном выше примере были выбраны круглые значения для различных величин и выбрали размер поля PT1, равный размеру поля PT2, но в реальной практике, конечно, возможны другие цифры.

Система двухуровневой таблицы на рис. 2 может быть расширена для трех, четырех и больше уровней. Дополнительные уровни дадут большую гибкость, но сомнительно, что следует усложнять систему больше, чем до трех уровней.

Структура элемента таблицы страниц

Теперь от структуры таблиц страниц в целом мы перейдем к деталям отдельного элемента записи таблицы. Точная структура элемента в значительной мере зависит от машины, но виды представленной информации примерно одни и те же. На рис. 3 представлен образец записи в таблице страниц. Ее длина изменяется от процессора к процессору, но 32 бита — это наиболее распространенный размер. Наиболее важным полем является **Номер страничного блока**. Прежде всего, задачей отображения страниц является определение этой величины. За этим полем следует бит **Присутствия/отсутствия**. Если этот бит равен 1, запись имеет силу и может использоваться. Если он равен 0, виртуальная страница, которой соответствует эта запись, в данный момент отсутствует в памяти. Обращение к записи в таблице страниц, у которой этому биту присвоено нулевое значение, приводит к страничному прерыванию.

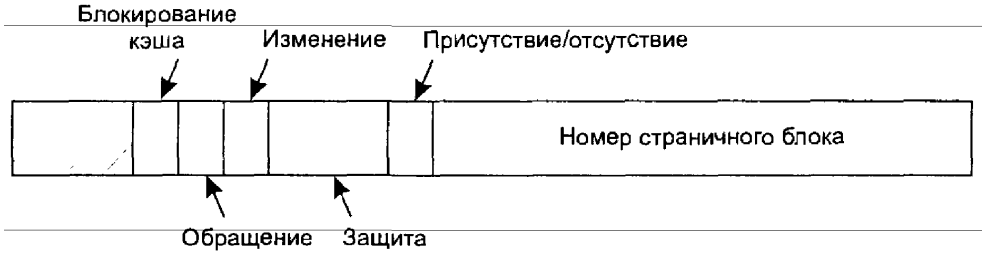

Рис. 3. Типичная запись в таблице страниц

Биты **Защиты** говорят о том, какие разрешены виды доступа к этой странице. В простейшей форме это поле содержит один бит, равный 1 для чтения/записи и равный 0 только для чтения. Более сложные схемы имеют три бита, по одному для допуска каждой из операций чтения, записи и выполнения страницы.

Биты **Изменения** и **Обращения** отслеживают использование страницы. Когда страница записывается, аппаратура автоматически устанавливает бит Изменение. Этот бит учитывается, когда ОС решает освободить страничный блок. Если страница в нем была изменена (то есть она «грязная»), то ее новая версия должна быть переписана на диск. Если она не была модифицирована (то есть страница «чистая»), ее можно просто удалить из памяти, так как все еще действительна копия на диске. Этот бит иногда называют грязным битом, так как он отражает состояние страницы.

Бит **Обращения** устанавливается всякий раз, когда происходит обращение к странице для чтения или записи. Его значение помогает ОС при выборе страницы для удаления из памяти, когда случается ошибка из-за отсутствия страницы. Страницы, не использующиеся в данный момент, являются лучшими кандидатами, чем находящиеся в

работе. Этот бит играет важную роль в нескольких алгоритмах перемещения страниц, которые будут рассмотрены немного позже.

Наконец, последний бит позволяет запретить кэширование страницы. Это свойство важно для страниц, отображающихся не на память, а на регистры устройств. Если ОС находится в цикле ожидания ответа от некоторого устройства ввода-вывода, которому была только что дана команда, существенно, чтобы аппаратура продолжала получать слово из устройства, а не использовало старую копию, находящуюся в кэш-памяти. При помощи этого бита кэширование можно отключить. Машины, имеющие отдельное пространство адресов ввода-вывода и не использующие отображения регистров ввода-вывода на память, не нуждаются в этом бите.

Заметим, что адрес места на диске, в котором хранится страница тогда, когда она не находится в памяти, не является частью таблицы страниц. Причина очень проста. Таблица страниц содержит только ту информацию, которая нужна аппаратуре для перевода виртуального адреса в физический. Информация, необходимая ОС для обработки страничных прерываний, хранится в программных таблицах внутри ОС. Аппаратуре она не нужна.

Буферы быстрого преобразования адреса (TLB)

В большинстве схем со страничной организацией памяти таблицы страниц хранятся в памяти из-за их значительного размера. Потенциально такое устройство оказывает колоссальное влияние на производительность. Рассмотрим, например, команду процессора, копирующую содержимое одного регистра в другой. В отсутствие страничной организации памяти эта команда приводит только к одному обращению к памяти для выборки самой команды. Если же память организована постранично, то будут необходимы дополнительные ссылки для доступа к таблице страниц. Так как скорость выполнения команд в основном ограничена скоростью, с которой центральный процессор выбирает команды и данные из памяти, необходимость двух обращений к таблице страниц на одну ссылку к памяти уменьшает производительность на 2/3. При таких условиях никто не стал использовать этот метод.

Разработчики процессоров многие годы размышляли об этой проблеме и в результате придумали решение. Оно основано на наблюдении, что большинство программ склонно делать огромное количество обращений к небольшому количеству страниц, а не наоборот. Таким образом, в таблице страниц только малая доля записей читается интенсивно, остальная часть едва ли вообще используется,

В результате принятого решения компьютер снабжается небольшим аппаратным устройством, служащим для отображения виртуальных адресов в физические без прохода по таблице страниц. Это устройство, называемое **буфером быстрого преобразования адреса** (TLB — Translation Lookaside Buffer) или иногда **ассоциативной памятью**, продемонстрировано в табл. 1. Оно обычно находится внутри MMU и состоит из нескольких записей. В примере их восемь, но фактически записей редко бывает больше 64. Каждая запись содержит информацию об одной странице, а именно:

- номер виртуальной страницы,
- бит, устанавливаемый при изменении страницы,
- код защиты (разрешения на чтение/ запись/выполнение)
- и номер физического страничного блока, в котором расположена эта страница.

Эти поля однозначно соответствуют полям в таблице страниц. Еще один бит служит признаком того, действительна ли запись (используется ли она в данный момент) или нет.

Таблица 1. Буфер быстрого преобразования памяти для увеличения скорости страничной подкачки

Действительный	Виртуальная страница	Изменение	Защита	Страничный фрейм
1	140	1	RW	31
1	20	0	RX	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	RX	50

1	21	0	RX	45
1	860	1	RW	74
1	861	1	RW	75

Пример, который мог бы сформировать TLB-буфер, изображенный на рис. 4, — циклический процесс, располагающийся в виртуальных страницах 19, 20 и 21, поэтому эти записи в буфере на рис. 6 имеют защитные коды для чтения и выполнения. Основные данные, используемые в настоящее время (скажем, обрабатываемый массив), находятся в страницах 129 и 130. Страница 140 содержит индексы, требующиеся для вычислений массива. И наконец, в страницах 860 и 861 находится стек.

Теперь рассмотрим, как же функционирует буфер быстрого преобразования адреса (TLB). Когда виртуальный адрес представляется MMU для отображения, аппаратура сначала убеждается в том, что номер его виртуальной страницы присутствует в буфере TLB путем сравнения адреса со всеми записями одновременно (то есть параллельно). Если найдено имеющее силу совпадение и обращение не нарушает биты защиты, страничный блок берется прямо из буфера TLB, без перехода к таблице страниц. Если номер виртуальной страницы присутствует в буфере TLB, но инструкция пытается записать что-то на страницу, доступную только для чтения, формируется ошибка защиты точно так же, как это происходило бы из самой таблицы страниц.

Интересная ситуация получается, если номер виртуальной страницы не находится в буфере быстрого преобразования адреса. MMU обнаруживает отсутствие страницы и выполняет обычный поиск в таблице страниц. Затем он удаляет одну из записей из буфера TLB и заменяет ее только что найденной записью из таблицы страниц. Таким образом, если эта страница снова вскоре будет затребована, во второй раз поиск окажется успешным, а не неудачным. Когда запись удаляется из буфера быстрого преобразования адреса, бит изменения копируется в запись таблицы страниц в памяти. Другие величины уже находятся там. Когда буфер TLB загружается из таблицы страниц, все поля берутся из памяти.

Инвертированные таблицы страниц

Традиционные таблицы страниц, тип которых описывались до сих пор, требуют по одной записи на каждую виртуальную страницу, так как они индексируются по номеру этой страницы. Если адресное пространство состоит из 232 байт с размером страницы 4096 байт, тогда в таблице страниц должно быть больше миллиона записей. При этом таблица страниц будет занимать минимум 4 Мбайт. В достаточно больших системах это, вероятно, выполнимо.

Однако поскольку 64-разрядные компьютеры встречаются все чаще, ситуация радикально меняется. Если теперь адресное пространство увеличилось до 2M байт с размером страницы 4 Кбайт, нам требуется таблица страниц с 252 записями. Если каждая запись равна 8 байтам, таблица займет больше 30 Тбайт. Выделение 30 Тбайт только для таблицы страниц нереально сейчас и не будет реальным когда-либо в будущем. Следовательно, для 64-разрядного страничного виртуального пространства необходимо другое решение.

Одним из таких решений является **инвертированная таблица страниц**. В этой модели таблица содержит по одной записи на страничный блок в реальной памяти, а не на страницу в виртуальном адресном пространстве. Например, при 64-разрядных виртуальных адресах, размере страниц 4 Кбайт и 256 Мбайт ОЗУ инвертированная таблица страниц потребует всего лишь 65 536 записей. Каждая запись отслеживает, что (процесс, виртуальная страница) расположено в данном страничном блоке.

Хотя инвертированные таблицы страниц экономят значительное количество места, по крайней мере, когда виртуальное адресное пространство намного больше, чем физическая память, они имеют **серьезный недостаток**: перевод виртуального адреса в физический становится намного сложнее. Когда процесс n обращается к виртуальной странице p, аппаратное обеспечение не может больше найти физическую страницу, используя номер p в качестве индекса в таблице страниц. Вместо этого оно должно производить поиск записи (n, p) во всей инвертированной таблице страниц. Более того, этот поиск должен выполняться при каждом обращении к памяти, а не только при страничном прерывании. Операция поиска в таблице размером 64 К при каждой ссылке к памяти вовсе не увеличит скорость вычислительной системы.

Выйти из этого затруднительного положения можно, используя **буфер быстрого преобразования адреса** (TLB). Если буфер TLB может содержать все часто используемые страницы, трансляция адреса будет происходить так же быстро, как и с обычными таблицами страниц. Но при неудачном поиске в буфере TLB поиск в инвертированной таблице страниц должен выполняться программно. Один из возможных **способов усовершенствовать** его — поддерживать хэш-таблицу виртуальных адресов. Все виртуальные страницы, находящиеся в данный момент в памяти и имеющие одинаковое значение хэш-функции, сцепляются друг с другом, как показано на рис. 4. Если хэш-таблица состоит из такого же количества ячеек, сколько в машине физических страниц, средняя цепочка будет длиной только в одну запись, что значительно увеличит скорость отображения адресов. Как только найден номер страничного блока, новая пара (виртуальная, физическая) помещается в буфер TLB.

Инвертированные таблицы страниц в настоящее время используются на некоторых рабочих станциях компаний IBM и Hewlett-Packard и будут встречаться все чаще, так как 64-разрядные машины получают все более широкое распространение.

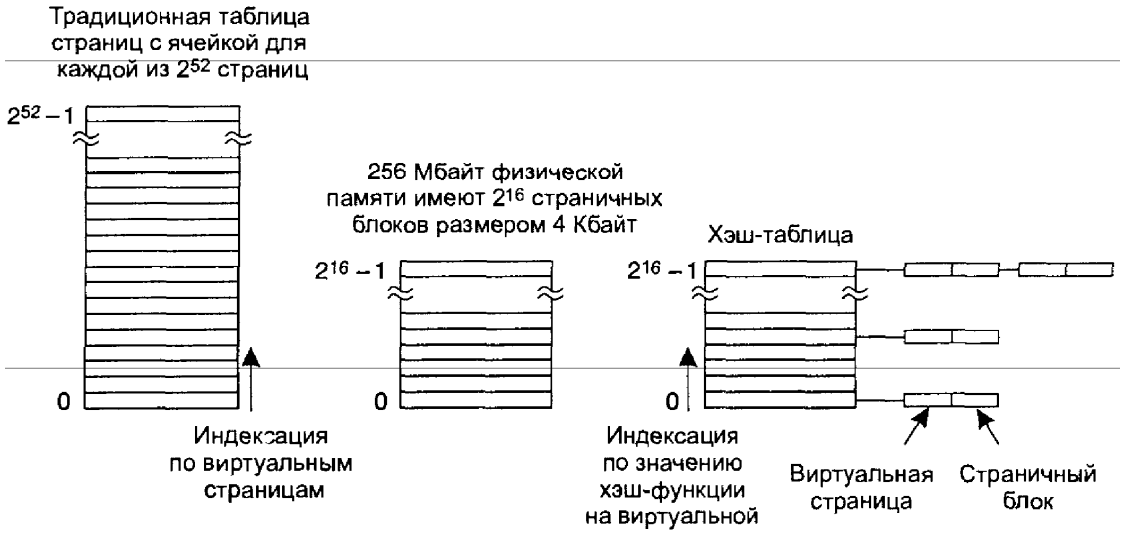

Рис. 4. Сравнение традиционной таблицы с инвертированной

Алгоритмы замещения страниц

Когда происходит страничное прерывание, ОС должна выбрать страницу для удаления из памяти, чтобы освободить место для страницы, которую нужно перенести в память. Если удаляемая страница была изменена за время своего присутствия в памяти, ее необходимо переписать на диск, чтобы обновить копию, хранящуюся там. Однако если страница не была модифицирована (например, она содержит текст программы), копия на диске уже является самой новой и ее не надо переписывать. Тогда страница, которую нужно прочитать, просто считывается поверх выгружаемой страницы.

Хотя в принципе можно при каждом страничном прерывании выбирать случайную страницу для удаления из памяти, производительность системы заметно повышается, когда предпочтение отдается редко используемой странице. Если выгружается страница, обращения к которой происходят часто, велика вероятность, что вскоре опять потребуется ее возврат в память, что даст в результате дополнительные издержки. Теме разработки алгоритмов замены страницы было посвящено много работ, как теоретических, так и экспериментальных. Ниже опишем некоторые из наиболее важных алгоритмов.

Следует отметить, что **проблема «страничного обмена»** также встает и в других областях конструирования компьютеров. Например, у большинства компьютеров есть один или несколько кэшей, состоящих из используемых в последнее время 32-байтовых или 64-байтовых блоков памяти. **Когда кэш заполнен, необходимо выбрать некоторые блоки для удаления**. Эта проблема практически аналогична замещению страниц лишь с одной разницей, заключающейся в меньшем масштабе времени (операция должна быть выполнена за несколько нс, а не мс, как для замены страниц). Причиной для более короткого промежутка времени является то, что неудачный поиск блока в кэше обрабатывается из основной памяти, в которой не тратится время на поиск нужного цилиндра диска и нет задержки из-за его вращения.

Второй пример встречается на web-серверах. Сервер может хранить определенное количество часто используемых web-страниц в своей кэш-памяти. Однако когда кэш-память заполняется целиком и происходит обращение к новой странице, должно приниматься решение о том, какую из страниц выгружать. Здесь применимы те же рассуждения, что и для страниц в виртуальной памяти, с той разницей, что web-страницы никогда не изменяются в кэше, поэтому для них всегда есть свежая копия на диске. В системе виртуальной памяти страницы в ОЗУ могут быть «чистыми» или «грязными».

Оптимальный алгоритм

Наилучший из возможных алгоритмов замещения страниц легко описать, но невозможно осуществить. Он действует так. В тот момент, когда происходит страничное прерывание, в памяти находится некоторый набор страниц. К одной из этих страниц будет обращаться следующая команда процессора (к странице, содержащей требуемую команду). На другие страницы, возможно, не будет ссылок в течение следующих 10, 100 или даже 1000 команд. Каждая страница может быть помечена количеством команд, которые будут выполняться перед первым обращением к этой странице.

Оптимальный страничный алгоритм просто сообщает, что должна быть выгружена страница с наибольшей меткой. Если одна страница не будет использоваться в течение 8 млн. команд, а другая — в течение 6 млн. инструкций, удаление первой отодвинет в будущее на возможно максимальный срок страничное прерывание, которое вернет ее назад. Компьютеры, подобно людям, пытаются отложить неприятные события настолько, насколько это возможно.

С этим алгоритмом **связана только одна проблема: он невыполним**. В момент страничного прерывания ОС не имеет возможности узнать, когда произойдет следующее обращение к каждой странице. Тем не менее, выполняя программу на модели и следя за всеми обращениями к страницам, **оптимальную замену можно осуществить при втором запуске**, используя информацию о ссылках на страницы, собранную во время первого запуска.

В этом случае можно сравнивать производительность реализуемых алгоритмов с наилучшим. Если ОС добивается производительности, скажем, всего на один процент ниже, чем при работе оптимального алгоритма, усилия, потраченные на поиск лучшего алгоритма, повысят продуктивность схемы максимум на 1%.

Чтобы избежать возможных недоразумений, следует прояснить, что полученный протокол обращений к страницам относится только к одной хорошо спланированной программе и, кроме того, к определенным входным данным. Таким образом, алгоритм замещения страниц, выведенный из него, будет характерен только для этой программы с именно этими входными данными. Хотя такой метод полезен для оценки алгоритмов замещения страниц, он не используется в практических системах. Ниже изучим алгоритмы, которые являются применимыми в реальных системах.

Алгоритм NRU — не использовавшаяся в последнее время страница

Чтобы дать возможность ОС 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. 5. 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 ОС 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 ОС 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 ОС 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 ОС 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, ОС 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 FIFO — первым прибыл — первым обслужен

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. ОС 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.

Алгоритм «вторая попытка»

В простейшем варианте алгоритма FIFO, который, позволяет избежать проблемы вытеснения из памяти часто используемых страниц, у самой старейшей страницы изучается бит R. Если он равен 0, страница не только находится в памяти долго, она вдобавок еще и не используется, поэтому немедленно заменяется новой. Если же бит R

равен 1, то ему присваивается значение 0, страница переносится в конец списка, а время ее загрузки обновляется, то есть считается, что страница только что попала в память. Затем процедура продолжается.

Работа этого алгоритма, называемого «второй попыткой» (second chance), показана на рис. 5(а). Здесь изображены страницы от А до Я, хранящиеся в связанном списке и отсортированные по времени их поступления в память. Числа над страницами обозначают их время загрузки в память.

Предположим, что в момент времени 20 происходит страничное прерывание. Самой старшей страницей является страница А, она была загружена в память во время 0, когда начал работу процесс. Если бит R страницы А 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 А передвигается в конец списка, а ее «загрузочное время» принимает текущее значение (20). При этом бит R очищается. Поиск подходящей страницы продолжается; следующей проверяется страница В.

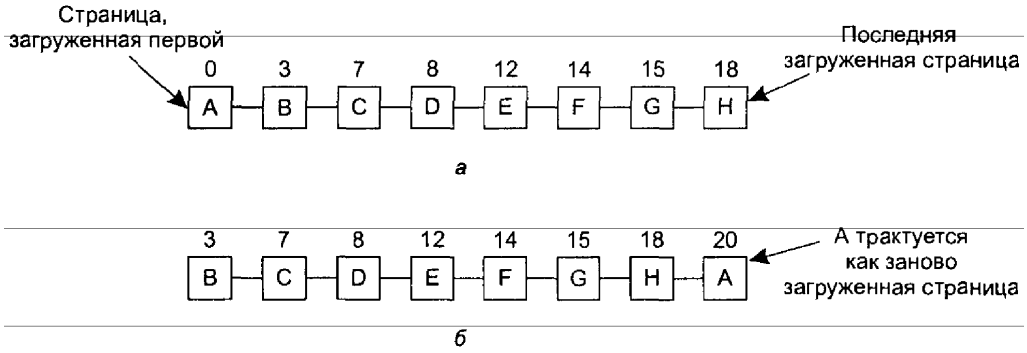

Рис. 5. Действие алгоритма «вторая попытка»: страницы, отсортированные в порядке очереди (FIFO) (а); список страниц, если страничное прерывание произошло во время 20, а страница А имеет бит R, равный 0 (б)

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. 5(а) бит R равен 1. Одну за другой передвигает ОС страницы в конец списка, очищая бит R каждый раз, когда она перемещает страницу в хвост. Наконец, она вернется к странице А, но теперь уже ее биту R присвоено значение 0. В этот момент страница А выгружается из памяти. Таким образом, алгоритм всегда успешно завершает свою работу.

Алгоритм «часы»

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. 6. 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.

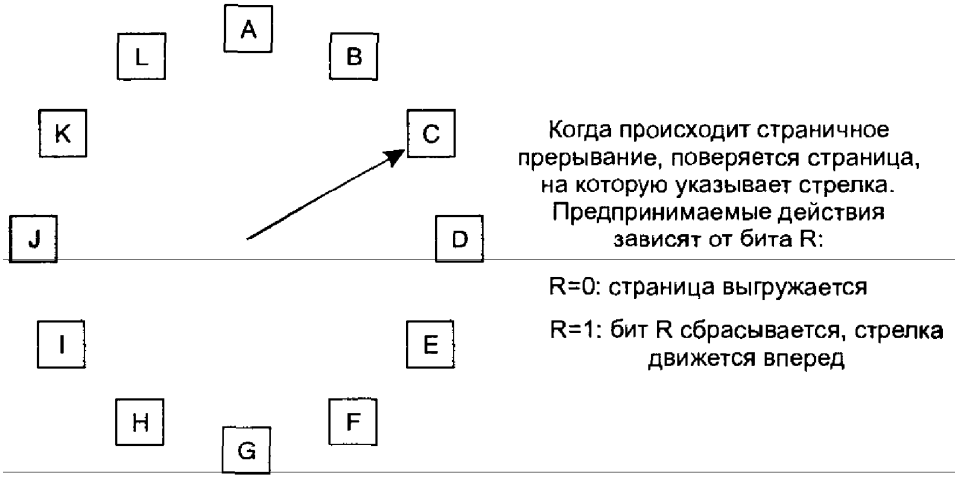

Рис. 6. Алгоритм замещения страниц «часы»

Алгоритм LRU — страница, не использовавшаяся дольше всего

В основе этой неплохой аппроксимации оптимального алгоритма лежит наблюдение, что страницы, к которым происходило многократное обращение в нескольких последних командах, вероятно, также будут часто использоваться в следующих инструкциях. И наоборот, можно полагать, что страницы, к которым ранее не возникало обращений, не будут употребляться в течение долгого времени. Эта идея привела к следующему реализуемому алгоритму: **когда происходит страничное прерывание, выгружается из памяти страница, которая не использовалась дольше всего.** Такая стратегия замещения страниц называется LRU (Least Recently Used — «менее недавно», то есть наиболее давно использовавшаяся страница).

Хотя алгоритм LRU теоретически реализуем, он не является дешевым. Для полного осуществления алгоритма LRU необходимо поддерживать связный список всех содержащихся в памяти страниц, где последняя использовавшаяся страница находится в начале списка, а та, к которой дольше всего не было обращений, — в конце. **Сложность заключается** в том, что список должен обновляться при каждом обращении к памяти. Поиск страницы, ее удаление, а затем вставка в начало списка — это операции, поглощающие очень много времени, даже если они выполняются аппаратно (если предположить, что необходимое оборудование можно сконструировать).

Однако существуют другие способы реализации алгоритма LRU с помощью специального оборудования. Для первого метода требуется оснащение компьютера 64-разрядным аппаратным счетчиком С, который автоматически возрастает после каждой команды. Кроме того, каждая запись в таблице страниц должна иметь поле, достаточно большое для хранения значения счетчика. После каждого обращения к памяти текущая величина счетчика С запоминается в записи таблицы, соответствующей той странице, к которой произошла ссылка. А если возникает страничное прерывание, ОС проверяет все значения счетчиков в таблице страниц и ищет наименьшее. Эта страница является не использовавшейся дольше всего.

Теперь рассмотрим второй вариант аппаратной реализации алгоритма LRU. На машине с n страничными блоками оборудование LRU может поддерживать матрицу $n \times n$ бит, изначально равных нулю. Всякий раз, когда происходит обращение к страничному блоку k, аппаратура сначала присваивает всем битам строки k значение 1, затем приравнивает к нулю все биты столбца k. В любой момент времени строка, двоичное значение которой наименьшее, является не использовавшейся дольше всего. Работа этого алгоритма продемонстрирована на рис. 7, где рассматриваются четыре страничных блока и следующий порядок обращения к страницам:

0123210323

После ссылки на страницу 0 мы получаем ситуацию, показанную на рис. 7(а); после обращения к странице 1 — рис. 7(б) и т. д.

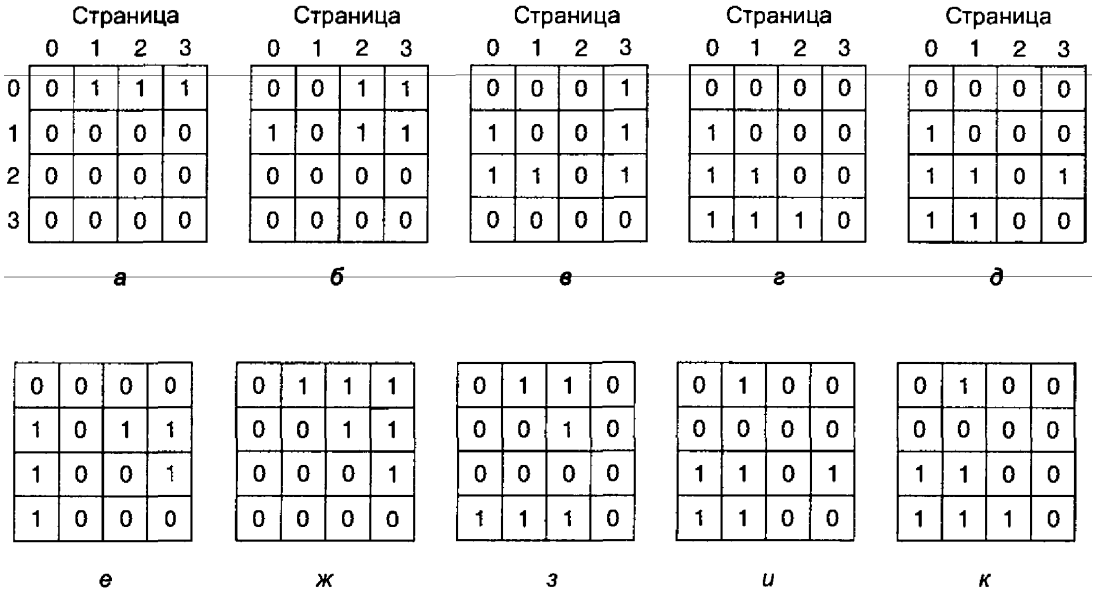

Рис. 7. Алгоритм LRU, использующий матрицу. Обращения к страницам происходят в порядке: 0, 1, 2, 3, 2, 1, 0, 3, 2, 3

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 ОС 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 команды, процесс обращается только к сравнительно небольшой части своих страниц. Каждый проход многопроходного компилятора, например, обращается только к части от общего количества страниц, и каждый раз к другой части.

Множество страниц, которое процесс использует в данный момент, называется **рабочим набором**. Если рабочий набор целиком находится в памяти, процесс будет работать, не вызывая большого количества ошибок, до тех пор пока он не перейдет к другой фазе выполнения (то есть к следующему проходу компилятора). Если доступная память слишком мала для того, чтобы содержать полный рабочий набор, процесс вызовет много страничных прерываний и будет работать медленнее, так как выполнение инструкции занимает несколько нс, а чтение страницы с диска обычно требует 10 мс. При скорости одна или две команды за 10 мс для завершения программы понадобятся века. Говорят, что программа, вызывающая страничное прерывание каждые несколько команд, **пробуксовывает (thrashing)**.

В многозадачных системах процессы часто перемещаются на диск (то есть все их страницы удаляются из памяти), чтобы позволить другим процессам получить доступ к центральному процессору. Возникает вопрос, что делать, когда процесс снова загружается в память. С формальной точки зрения делать ничего не нужно. Процесс будет вызывать одно за другим страничные прерывания до тех пор, пока не загрузится в память весь его рабочий набор. Проблема в том, что наличие 20, 100 или даже 1000 страничных прерываний при каждой загрузке процесса сильно замедляет работу системы и, кроме того, тратит впустую значительное количество времени работы центрального процессора, так как обработка страничного прерывания ОС требует нескольких миллисекунд работы процессора.

Поэтому многие системы со страничной организацией пытаются отслеживать рабочий набор каждого процесса и обеспечивают его нахождение в памяти до запуска процесса.

Такой подход носит название **модели рабочего набора**. Он разработан для того, чтобы значительно снизить процент страничных прерываний. Загрузка страниц перед тем, как разрешить процессу работать, также называется **опережающей подкачкой страниц (prepaging)**. Заметьте, что рабочий набор изменяется с течением времени.

Давно известно, что большинство программ не обращаются к своему адресному пространству равномерно; чаще всего ссылки группируются на небольшом количестве страниц. Обращение к памяти может быть выборкой команды, данных или сохранением данных. В любой момент времени t существует множество страниц, использовавшихся за k последних обращений к памяти. Это множество $w(k,t)$ и есть **рабочий набор**. Так как все недавние обращения к памяти для $k>1$ обязательно должны были обращаться ко всем страницам, использовавшимся для $k=1$ обращения к памяти, то есть к последней и, возможно, еще к некоторым страницам, $w(k,t)$ является **монотонно неубывающей функцией** от k. Функция $w(k,t)$ ограничена для больших k, потому что программа не может обращаться к большему количеству страниц, чем содержится в ее адресном пространстве, кроме того, редкие программы обращаются ко всем страницам адресного пространства. На рис. 8 изображена зависимость размера рабочего набора от k.

Тот факт, что большинство программ случайным образом обращается к небольшому числу страниц, но это множество медленно изменяется во времени, объясняет быстрое возрастание функции в начале и затем медленное увеличение для больших k. Например, программа, которая выполняет цикл, затрагивающий две страницы и использующий данные на четырех страницах, может обращаться ко всем шести страницам через каждые 1000 инструкций, но самое последнее обращение к некоторым другим страницам могло произойти миллионом команд раньше, во время начальной загрузки фазы. Вследствие этого асимптотического характера содержимое рабочего набора не чувствительно к выбранной величине k. Существует широкий диапазон значений k, для которых рабочий набор постоянен. Поскольку рабочий набор медленно меняется со временем, можно сделать разумное предположение, что те страницы, которые будут нужны для возобновления работы программы, составляют основу рабочего набора во время последней остановки процесса. Опережающая подкачка страниц заключается в загрузке рабочего набора до того, как процессу разрешается возобновиться.

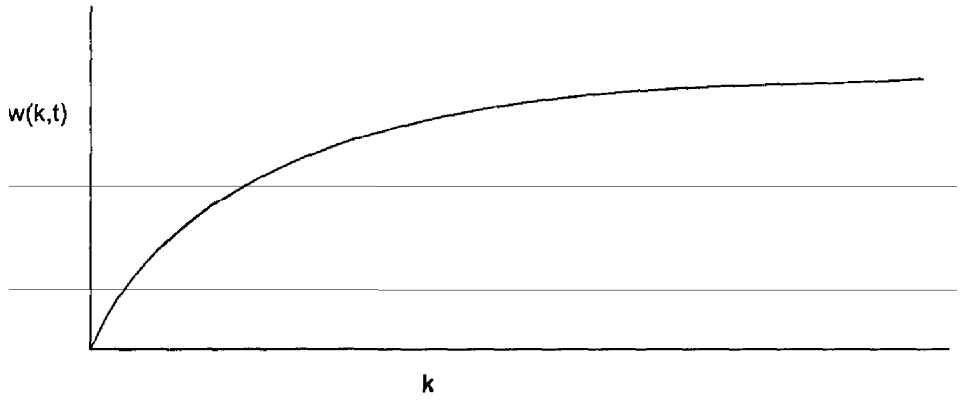

Рис. 8. Рабочий набор — это множество страниц, используемых к последними обращениями к памяти. Функция $w(k,t)$ представляет собой размер рабочего набора в момент времени t

Чтобы реализовать модель рабочего набора, необходимо, чтобы ОС отслеживала, какие страницы в нем находятся. Наличие этой информации также немедленно приводит к возможному алгоритму замещения страниц: когда происходит страничное прерывание, ищется и выгружается страница, не находящаяся в рабочем наборе. Для реализации такого алгоритма нужен точный метод определения того, какая страница находится в рабочем наборе, а какая в него не включена в любой заданный момент времени.

Как упоминали выше, рабочим набором является множество страниц, использовавшихся в k последних обращениях к памяти (некоторые авторы используют k последних страничных обращений, но выбор произволен). Для осуществления алгоритма «рабочий набор» заранее нужно назначить значение k. Как только

некоторая величина выбрана, после каждого обращения к памяти набор страниц, используемых за предыдущие k обращений, определяется единственным образом.

Конечно, наличие действующего определения рабочего набора вовсе не означает, что существует эффективный способ отслеживания его в реальном времени, то есть во время выполнения программы. Можно было бы придумать сдвигающийся регистр длины k, который смещается влево на одну позицию при каждом обращении к памяти и записывает номер последней использовавшейся страницы в крайнюю правую позицию. Все k номеров страниц в сдвигающемся регистре входили бы в рабочий набор. Теоретически в момент страничного прерывания содержимое этого регистра могло бы считываться и сохраняться. Затем удалялись бы дублирующиеся страницы. В результате мы бы получали рабочий набор. Однако поддержка сдвигающегося регистра и его обработка при страничном прерывании чрезвычайно дороги, поэтому данная техника никогда не употребляется на деле.

Вместо нее используются различные аппроксимации. Применяемый в большинстве случаев подход заключается в том, чтобы оставить идею подсчета k последних обращений к памяти и вместо этого использовать время выполнения программы. Например, вместо определения рабочего набора как множества страниц, на которые ссылались при предыдущих 10 млн обращений к памяти, можем определить его как множество страниц, использовавшихся в течение последних 100 мс времени выполнения. На практике это определение имеет тот же смысл, но намного упрощает реализацию алгоритма. Заметим, что для каждого процесса считается только его собственное время работы. Таким образом, если процесс стартовал во время T и занял процессор на 40 мс за реальное время T+100 мс, для определения рабочего набора его время равно 40 мс. Время работы процессора, которое фактически использовал процесс с момента запуска, часто называется текущим виртуальным временем. При таком приближении рабочий набор процесса — это множество страниц, к которому он обращался за последние τ секунд виртуального времени.

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. 9 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 и 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 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 т, это означает, что страница более не находится в рабочем наборе.

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

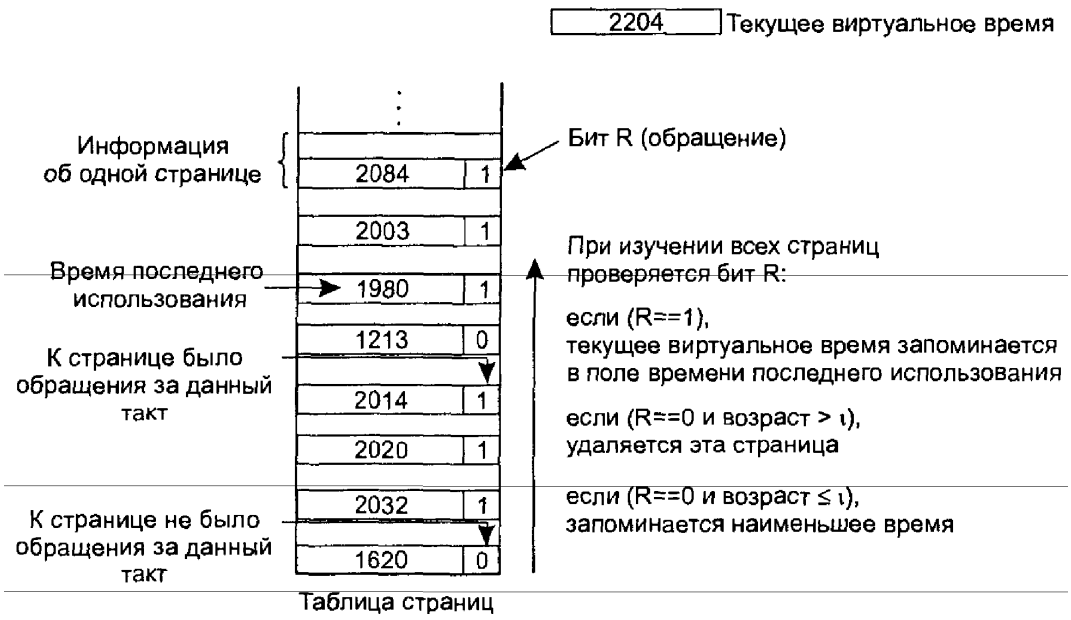

Рис. 9. Алгоритм «рабочий набор»

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.

Алгоритм WSCIock

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 **WSCIock**. 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. 10(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».

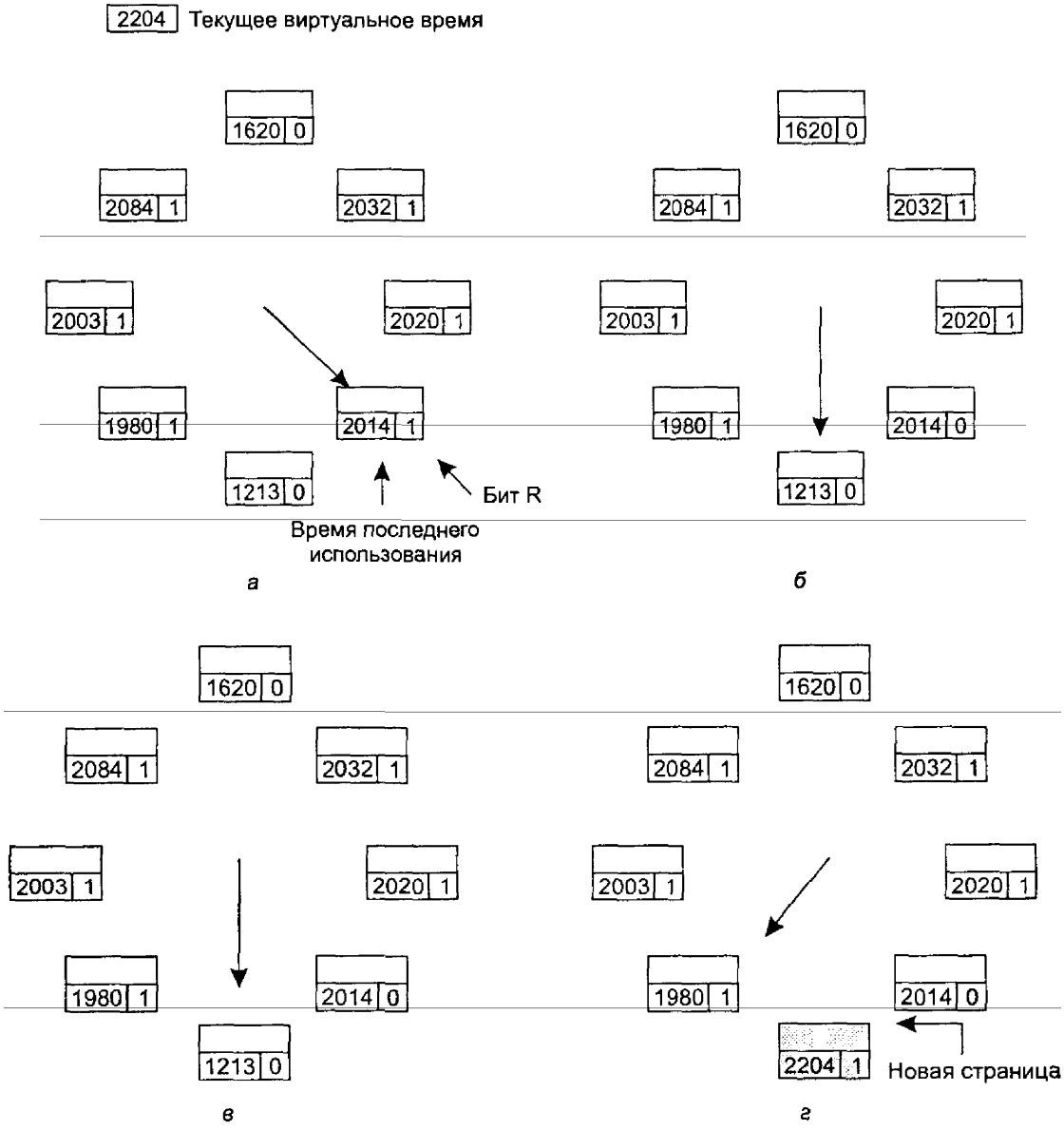

Рис. 10. Работа алгоритма WSCIock: пример того, что происходит при бите $R = 1$ (а) и (б); пример для бита $R = 0$ (в) и (г)

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. 10(б).

Теперь рассмотрим, что происходит, если страница, на которую указывает стрелка, имеет бит $R = 0$, как показано на рис. 10(в). 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. 10(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.

Выводы

Мы рассмотрели множество различных алгоритмов замещения страниц. В этом разделе мы кратко подведем итоги вышесказанного. Список обсужденных алгоритмов представлен в табл. 2.

Оптимальный алгоритм заменяет ту страницу, обращение к которой производилось раньше других, находящихся в данный момент в памяти. К сожалению, не существует способа определения того, какая страница будет последней, поэтому данный алгоритм не может использоваться на практике. Но он полезен в качестве тестовой задачи, относительно которой можно оценивать другие алгоритмы.

Алгоритм NRU (не использовавшаяся в последнее время страница) делит страницы на четыре класса в зависимости от состояния битов R и M. Выбирается любая страница из класса с наименьшим номером. Этот алгоритм легко реализуется, но он является очень грубым. Существуют лучшие схемы.

Таблица 2. Алгоритмы замещения страниц

Алгоритм	Комментарии
Оптимальный	Не осуществим, но полезен в качестве тестовой задачи
NRU (не использовавшаяся в последнее время страница)	Очень грубый
FIFO (первым прибыл, первым обслужен)	Может выгрузить важные страницы
Вторая попытка	Значительное усовершенствование FIFO
Часы	Реалистичный
LRU (страница, не использовавшаяся целиком дольше всего)	Отличный алгоритм, но его сложно осуществить
NFU (редко использовавшаяся страница)	Довольно грубое приближение алгоритма LRU
Старение	Эффективный алгоритм, хорошо аппроксимирующий алгоритм LRU
Рабочий набор	Немного дорог для реализации
WSCIock	Хороший рациональный алгоритм

Алгоритм FIFO (первым прибыл — первым обслужен) отслеживает порядок загрузки страниц в память, храня их в связном списке. При этом удаление старейшей страницы

становится тривиальным, но эта страница может использоваться в данный момент, поэтому алгоритм FIFO представляет собой плохой выбор.

Алгоритм «вторая попытка» — это модификация алгоритма FIFO, он перед удалением страницы из памяти проверяет, используется ли она в данный момент. Если да, то страница пропускается. Такое изменение сильно повышает производительность. Алгоритм «часы» представляет собой всего лишь другое осуществление алгоритма «второй попытки». Он имеет те же самые характеристики производительности, но требует немного меньше времени на выполнение алгоритма.

Алгоритм LRU (страница, не использовавшаяся дольше всего) — это отличный алгоритм, но его нельзя осуществить без специального аппаратного обеспечения. Если подобное оборудование недоступно, алгоритм невозможно использовать. Алгоритм NFU (редко использовавшаяся страница) представляет собой грубую попытку аппроксимации алгоритма LRU, Он не очень хорош. Но существует алгоритм «старения», который намного лучше аппроксимирует алгоритм LRU и может быть эффективно реализован. Это замечательный выбор.

Последние два алгоритма используют рабочий набор. Алгоритм «рабочий набор» обладает приемлемой производительностью, но дорог в реализации. Алгоритм WSClock — это вариант, который не только дает достойную производительность, но его также достаточно просто реализовать.

В итоге двумя наилучшими алгоритмами являются «старение» и WSClock. Они основаны на алгоритме LRU и понятии рабочего набора соответственно. Оба обеспечивают хорошую постраничную подкачку и могут быть реализованы за разумную цену. Существует еще несколько алгоритмов, но для практических целей эти два являются, вероятно, наиболее важными.

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. А. Шоу. Логическое проектирование операционных систем. Пер. с англ. –М.: Мир, 1981. –360 с.

3. С. Кейслер. Проектирование операционных систем для малых ЭВМ: Пер. с англ. –М.: Мир, 1986. –680 с.

4. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

5. Microsoft Development Network. URL: http://msdn.com