**1. Для чего используются базовый и ограничительный регистры?**

Классическим решением проблемы трансляции адресов является оснащение каждого центрального процессора двумя специальными аппаратными регистрами: базовым и ограничительным. При использовании этих регистров программы загружаются в последовательно расположенные свободные области памяти без модификации адресов в процессе загрузки.

При запуске процесса в базовый регистр загружается физический адрес, с которого начинается размещение программы в памяти, а в ограничительный регистр загружается длина программы.

При каждой ссылке процесса на память с целью извлечения команды или записи слова данных аппаратура центрального процессора перед выставлением адреса на шине памяти добавляет к адресу, сгенерированному процессом, значение базового регистра. Одновременно аппаратура проверяет, не равен ли предлагаемый адрес значению ограничительного регистра или не превышает ли он это значение,и в этом случае генерируется отказ и доступ прерывается.

Использование базового и ограничительного регистров – простой способ предоставления каждому процессу своего собственного закрытого адресного пространства, поскольку к каждому автоматически сгенерированному адресу перед обращением к памяти добавляется значение базового регистра. Многие реализации предусматривают такую защиту базового и ограничительного регистров, при которой изменить их значения может только операционная система.

Недостатком перемещений с использованием базовых и ограничительных регистров является необходимость применения операций сложения и сравнения к каждой ссылке на ячейку памяти. Сравнение может осуществляться довольно быстро, но сложение является слишком медленной операцией из-за затрат времени на вспомогательный сигнал переноса, если не используются специальные сумматоры.

**2. Для чего используется свопинг?**

Если компьютер обладает достаточным объемом памяти для размещения всех процессов, то все рассмотренные до сих пор схемы будут в той или иной степени работоспособны. Но на практике суммарный объем оперативной памяти, необходимый для размещения всех процессов, зачастую значительно превышает имеющийся объем ОЗУ.

Свопинг – размещение в памяти всего процесса целиком. Процесс запускается на некоторое время, а затем сбрасывается на диск. Бездействующие процессы большую часть времени хранятся на диске и в нерабочем состоянии не занимают пространство оперативной памяти.

Если предполагается, что большинство процессов по мере выполнения будут разрастаться, необходимо распределять небольшой объем дополнительной памяти при каждой загрузке процесса из области свопинга на диске в память (перемещении процесса внутри памяти), чтобы процессы помещались в отведенной им памяти.

**3. Для чего используются оверлеи?**

Проблема запуска программ, превышающих по объему размер имеющейся памяти, возникла еще на заре компьютерной эры.

В 60-е годы было принято решение разбивать программы на небольшие части, называемые оверлеями. При запуске программы в память загружался только администратор оверлейной загрузки, который тут же загружал и запускал оверлей с порядковым номером 0. Когда этот оверлей завершал свою работу, он мог сообщить администратору загрузки оверлеев о необходимости загрузки оверлея 1. Некоторые оверлейные системы имели довольно сложное устройство, позволяя одновременно находиться в памяти множеству оверлеев. Оверлеи хранились на диске, и их свопинг с диска в память и обратно осуществлялся администратором загрузки оверлеев.

Работа по свопингу оверлеев с диска в память и обратно выполнялась операционной системой, при этом разбиение программ на части выполнялось программистом в ручном режиме. Разбиение больших программ на небольшие модульные части было очень трудоемкой, скучной и не застрахованной от ошибок работой.

**4. Какие проблемы решает виртуальная память?**

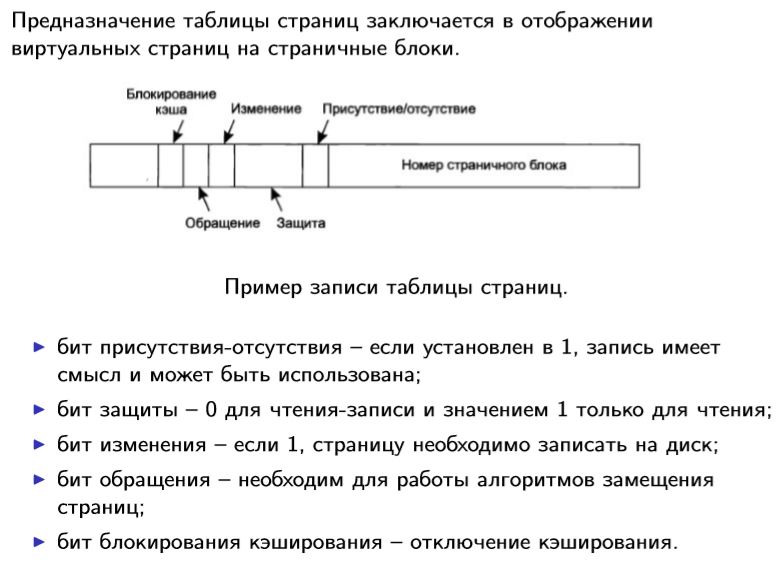
В основе виртуальной памяти лежит идея, что у каждой программы имеется свое собственное адресное пространство, которое разбивается на участки, называемые страницами.

Каждая страница представляет собой непрерывный диапазон адресов. Эти страницы отображаются на физическую память, но для запуска программы присутствие в памяти всех страниц не обязательно.

Когда программа ссылается на часть своего адресного пространства, находящегося в физической памяти, аппаратное обеспечение осуществляет необходимое отображение на лету.

Когда программа ссылается на часть своего адресного пространства, которое НЕ находится в физической памяти, операционная система предупреждается о том, что необходимо получить недостающую часть и повторно выполнить потерпевшую неудачу команду.

**5. Для чего используется страничная организация памяти?**



1. Буферы быстрого преобразования адреса. Буфер быстрого преобразования адреса (TLB) – небольшое устройство для отображения виртуальных адресов на физические без просмотра таблицы страниц.

2. Программное управление буфером TLB. Программное обновление индекса TLB в случае отсутствия страницы в TLB и ее наличии в ОЗУ.

3. Многоуровневые таблицы страниц.

4. Инвертированные таблицы страниц.

**6. Для чего используется сегментная организация памяти?**

Для решения многих проблем наличие двух и более отдельных виртуальных адресных пространств может быть более рациональным вариантом, чем наличие только одного адресного пространства.

Например, у компилятора имеется множество таблиц, выстраиваемых в процессе компиляции, в которые могут входить: 1. Исходный текст, сохраненный для печати листинга. 2. Таблица имен, содержащая имена и атрибуты переменных. 3. Таблица, содержащая все используемые константы, как целочисленные, так и с плавающей точкой. 4. Дерево разбора, в котором содержится синтаксический анализ программы. 5. Стек, используемый для вызовов процедур внутри компилятора.

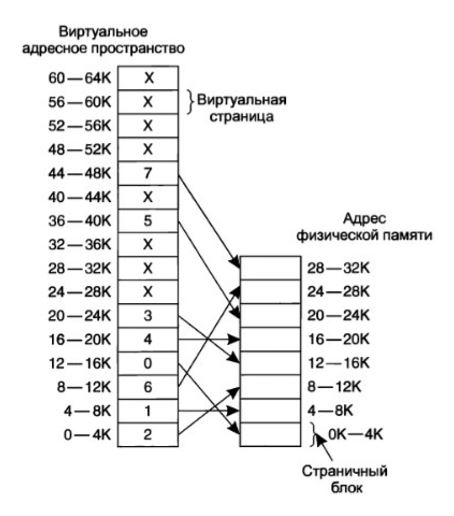
В процессе компиляции каждая из первых четырех таблиц постоянно растет. А последняя увеличивается и уменьшается в размерах совершенно непредсказуемым образом.

Каждый сегмент состоит из линейной последовательности адресов от 0 до некоторого максимума. Длина каждого сегмента может иметь любое значение от 0 до максимально разрешенного. Различные сегменты могут быть разной длины, как это обычно и случается. Кроме того, длина сегмента может изменяться в процессе выполнения программы. Длина сегмента стека может увеличиваться при поступлении в него данных и уменьшаться при их извлечении из него.

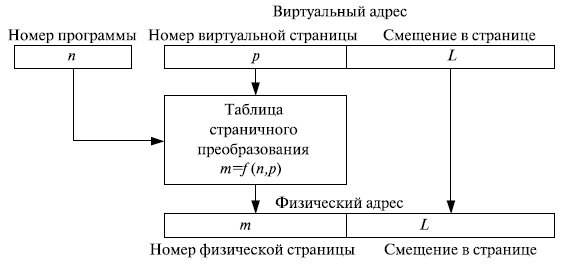
**Сегментированная память дает возможность каждой таблице разрастаться или сужаться независимо от всех остальных таблиц.**

5. Что такое страничное прерывание?

**Страничное прерывание** - происходит, если процесс обратился к странице, которая не загружена в ОЗУ (т.е. Х). Процессор передается другому процессу, и параллельно страница загружается в память.



**7. Как работает трансляция адресов?**



Для адресации операндов в физическом адресном пространстве программы используют логическую адресацию. Процессор автоматически транслирует логические адреса в физические, выдаваемые затем на системную шину.

Архитектура компьютера различает физическое адресное пространство (ФАП) и логическое адресное пространство (ЛАП). **Физическое адресное пространство**представляет собой простой одномерный массив байтов, доступ к которому реализуется аппаратурой памяти по адресу, присутствующему на шине адреса микропроцессорной системы.**Логическое адресное пространство**организуется самим программистом исходя из конкретных потребностей. Трансляцию логических адресов в физические осуществляет блок управления памятью MMU.

В архитектуре современных микропроцессоров ЛАП представляется в виде набора элементарных структур: байтов, сегментов и страниц. В микропроцессорах используются следующие варианты организации **логического адресного пространства**:

\* **плоское (линейное) ЛАП**: состоит из массива байтов, не имеющего определенной структуры; трансляция адреса не требуется, так как логический адрес совпадает с физическим;

\* **сегментированное ЛАП**: состоит из сегментов - непрерывных областей памяти, содержащих в общем случае переменное число байтов; логический адрес содержит 2 части: идентификатор сегмента и смещение внутри сегмента; трансляцию адреса проводит блок сегментации MMU;

\* **страничное ЛАП**: состоит из страниц - непрерывных областей памяти, каждая из которых содержит фиксированное число байтов. Логический адрес состоит из номера (идентификатора) страницы и смещения внутри страницы; трансляция логического адреса в физический проводится блоком страничного преобразования MMU;

\* **сегментно-страничное ЛАП**: состоит из сегментов, которые, в свою очередь, состоят из страниц; логический адрес состоит из идентификатора сегмента и смещения внутри сегмента. Блок сегментного преобразования MMU проводит трансляцию логического адреса в номер страницы и смещение в ней, которые затем транслируются в физический адрес блоком страничного преобразования MMU.

**8. Для чего используются алгоритмы замещения страниц?**

Большинство ОС используют сегментно-страничную виртуальную память. Для обеспечения нужной производительности менеджер памяти ОС старается поддерживать в оперативной памяти актуальную информацию, пытаясь угадать, к каким логическим адресам последует обращение в недалеком будущем. Это влияет на производительность системы. Ниже мы рассмотрим основные алгоритмы замещения страниц и проведем их сравнительный анализ.

Итак, наиболее ответственным действием страничной системы является выделение страницы основной памяти для удовлетворения требования доступа к отсутствующей в основной памяти виртуальной странице. Напомним, что мы рассматриваем ситуацию, когда размер каждой виртуальной памяти может существенно превосходить размер основной памяти. Это означает, что при выделении страницы основной памяти с большой вероятностью не удастся найти свободную (не приписанную к какой-либо виртуальной памяти) страницу. В этом случае операционная система должна в соответствии с заложенными в нее критериями найти некоторую занятую страницу основной памяти, переместить в случае надобности ее содержимое во внешнюю память, должным образом модифицировать соответствующий элемент соответствующей таблицы страниц и после этого продолжить процесс удовлетворения доступа к странице.

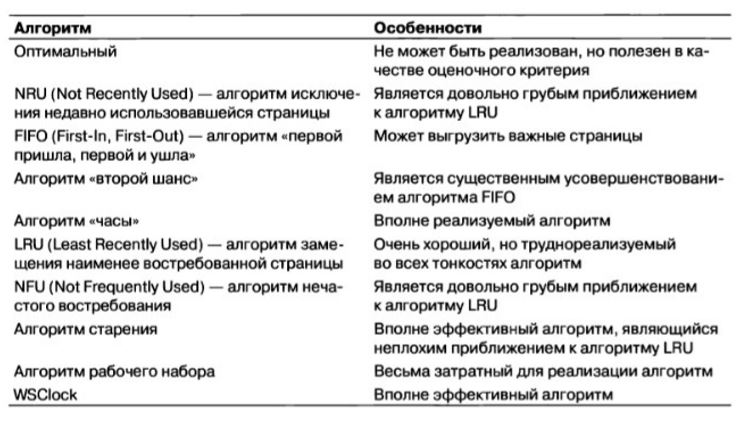
Заметим, что при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения может быть оптимизирован за счет использования бита модификации (один из атрибутов страницы). Бит модификации устанавливается компьютером, если хотя бы один байт записан на страницу.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

Существует большое количество разнообразных алгоритмов замещения страниц. Все они делятся на локальные и глобальные. Локальные алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему страницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов.  Глобальный же алгоритм замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют несколько недостатков. Во-первых, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе использует большое количество памяти, то все остальные приложения будут в результате ощущать сильное замедление из-за недостатка памяти. Во-вторых, некорректно работающее приложение может подорвать работу всей системы (если конечно в системе не предусмотрено ограничение на размер памяти, выделяемой процессу), пытаясь захватить все больше памяти. Поэтому в многозадачной системе лучше использовать более сложные, но эффективные локальные алгоритмы. Такой подход требует, чтобы система хранила список физических страниц каждого процесса. Этот список страниц иногда называют *рабочим множеством* процесса.  Рабочее множество и реализация алгоритма подкачки, основанного на понятиях локальности и рабочего множества описаны в последующих разделах.

**9. Перечислите алгоритмы замещения страниц.**

https://studfiles.net/preview/2567855/page:2/



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 он 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, то 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 предоставляет приемлемую производительность, но его реализация обходится слишком дорого.

Алгоритм **WSClock** является вариантом, который не только предоставляет неплохую производительность, но также может быть эффективно реализован.

В конечном итоге наиболее приемлемыми алгоритмами являются алгоритм **старения** и алгоритм **WSClock.** Они основаны соответственно на **LRU** и на **рабочем наборе**. Оба предоставляют неплохую производительность страничной организации памяти и могут быть эффективно реализованы. Существует также и ряд других алгоритмов, но эти два, наверное, имеют наибольшее практическое значение.

**Понятие оптимального алгоритма**

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 через 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 миллионов команд, а другая какая-нибудь страница не будет использоваться на протяжении 6 миллионов 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.

**Алгоритм исключения недавно использовавшейся страницы. NRU**

Алгоритм исключения недавно использовавшейся страницы — NRU(Not Re­cently Used) 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 интенсивно 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.

**Алгоритм «первой пришла, первой и ушла» FIFO**

Другим низкозатратным алгоритмом замещения страниц является алгоритм FIFO (First-In, First-Out — то есть «первой пришла, первой и ушла»). 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. Поэтому принцип FIFO в его чистом виде используется довольно редко.

**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 *А* сброшен, страница удаляется из памяти либо с записью на диск (если она измененная), либо просто удаляется (если она неизмененная). Но если бит *R* установлен, то *А* 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 *В.* 

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* был 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 *А,* у которой бит *R* теперь уже сброшен. И тогда страница *А* выселяется. Таким образом, алгоритм всегда завершает свою работу.

**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, как показано на рис. 3.15. Стрелка указывает на самую старую страницу



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.

**Алгоритм замещения наименее востребованной страницы. LRU**

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 невостребованные страницы наверняка еще долго так и останутся невостребованными. Эта мысль наталкивает на вполне осуществимый алгоритм: при возникновении ошибки отсутствия страницы нужно избавиться от той страницы, которая длительное время не была востребована. Эта стратегия называется замещением наименее востребованной страницы — LRU (Least Re­cently Used).

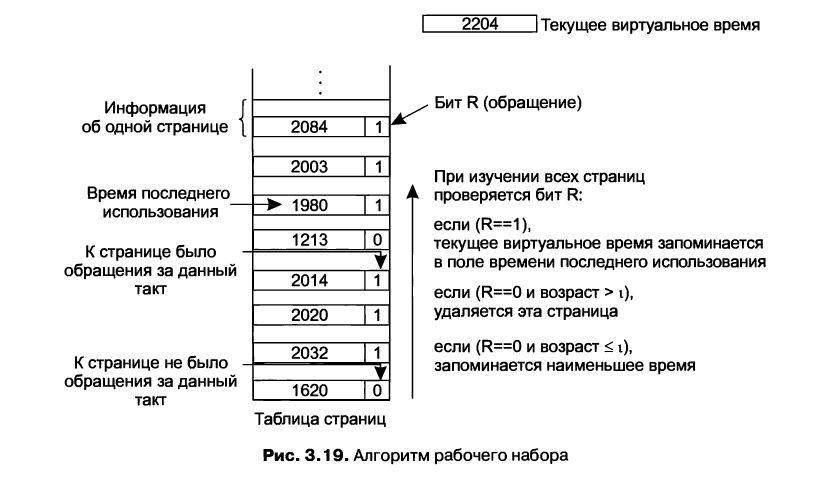
Теоретически реализовать алгоритм LRU вполне возможно, но его практическая реализация дается нелегко. Для его полной реализации необходимо вести связан­ный список всех страниц, находящихся в памяти. В начале этого списка должна быть только что востребованная страница, а в конце — наименее востребованная. 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).

**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 пытается 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. Ими применяется локальность обращений, означающая, 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 другой части.

Для реализации модели рабочего набора необходимо, чтобы операционная система отслеживала, какие именно страницы входят в рабочий набор. При нали­чии такой информации тут же напрашивается и 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, какие именно страницы относятся к рабочему набору По определению рабочий набор — это набор страниц, используемых в *k* самых последних обращений (некоторые авторы используют термин *k* самых последних страничных обращений, но это дело вкуса). Для реа­лизации любого алгоритма рабочего набора некоторые значения *k* должны быть выбраны заранее. Как только после каждого обращения к памяти будет выбрано некоторое значение, однозначно определяется и набор страниц, используемый при самых последних *k* обращениях к памяти.

****

**Алгоритм WSCIock**

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** (Carr and Hennessey, 1981). 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. 3.20, 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 М (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. 3.20, б.

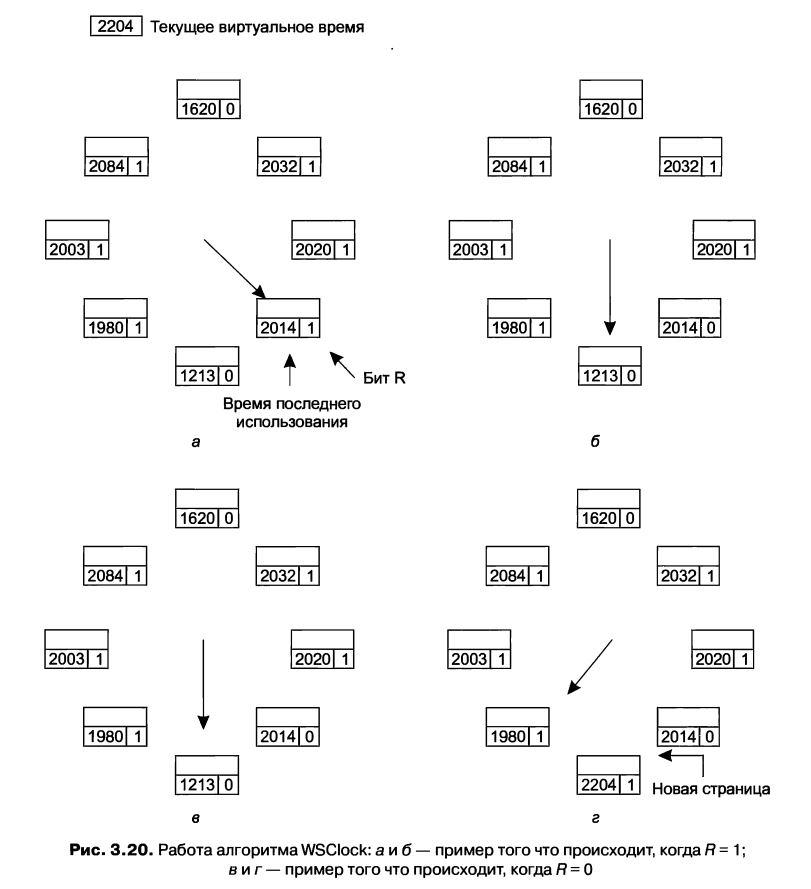
Теперь посмотрим, что получится, если у страницы, на которую указывает стрелка, бит R = 0, как показано на рис. 3.20, в. Если ее возраст превышает значение х и страница не изменена, она не относится к рабочему набору и ее точная копия присутствует на диске. Тогда страничный блок просто истребуется, и в него помещается новая страница, как и показано на рис. 3.20, г. С другой стороны, если страница была 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 уже не планируются.

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, — это 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.

**10. Перечислите алгоритмы выделения памяти.**

I первое подходящее. Диспетчер памяти сканирует список сегментов до тех пор, пока не найдет пустое пространство подходящего размера. Затем пустое пространство разбивается на две части: одна для процесса и одна для неиспользуемой памяти;

I следующее подходящее. Данный алгоритм работает так же, как и «первое подходящее», за исключением того, что отслеживает свое местоположение, как только находит подходящее пустое пространство. При следующем вызове для поиска пустого пространства он начинает поиск в списке с того места, на котором остановился в прошлый раз, а не приступает к поиску с самого начала, как в предыдущем алгоритме;

I наиболее подходящее. Данный алгоритм ведет поиск по всему списку от начала до конца и выбирает наименьшее соответствующее пустое пространство. Вместо того чтобы разбивать большое пустое пространство, которое может пригодиться чуть позже, данный алгоритм пытается подыскать пустое пространство, близкое по размеру к необходимому, чтобы наилучшим образом соответствовать запросу и имеющимся пустым пространствам;

I наименее подходящее. Данный алгоритм выбирает самое большое подходящее пустое пространство, чтобы вновь образующееся пустое пространство было достаточно большим для дальнейшего использования;

I быстро искомое подходящее. Его использование предусматривает ведение отдельных списков для некоторых наиболее востребованных искомых размеров.