Управление памятью

Задачи управления памятью у ОС

- Распределение ресурса типа «память» между различными, конкурирующими за нее процессами (т.к. памяти всегда не хватает, это ограниченный ресурс по своей сути)
- Максимизировать использование памяти
- Получить дополнительные «бонусы» в виде изоляции процессов (защита доступа одного процесса от другого)
- Абстрагировать доступ к памяти для программистов.

Загрузку ОП можно посмотреть в Taskmanager

Инструменты управления памятью

- Регистры база-предел
- Своп
- Страницы (также таблицы страниц)
- Сегменты (таблицы сегментов)
- Страничное прерывание (page fauet) и виртуальная память

Современные ОС

Основным механизмом абстракции в современных ОС является виртуальная память(**virtual memory**), используется повсеместно, так как:

-**позволяет эффективно использовать реальную память**

- VM позволяет программам работать без необходимости загружать все их адр.пространство в физическую память (используется свопинг)
- Большинству программ не нужны сразу все их данные и код

-**Гибкость программ**

Сама программа «не знает» сколько физ.памяти осталось в системе, а сколько – свопа. Объем памяти для любого процесса должен быть организован по принципу: сколько ему нужно, а не сколько есть всего в системе.

- **Позволяет организовать защиту**

  Виртуальная память изолирует адресное пространство процессов друг от друга

Аппаратная поддержка для VM

Виртуальная память требует аппаратной поддержки:

- MMU (*memory management unit*) -Блок управления памятью
- TLB (*Translation lookaside buffe*) - Буфер ассоциативной трансляции
- Таблицы страниц
- Обработка страничных прерываний

Обычно есть поддержка свопинга и ограниченной сегментации.

Далее мы будем рассматривать разные алгоритмы организации памяти. Часто будем обращаться к понятию **фрагментация памяти.**

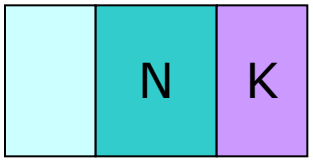
Фрагментация

По сути это неэффективное использование памяти.

Очевидный минус – снижается объем доступной памяти.

Существует 2 типа фрагментации:

- **Внутренняя**: когда выделяется больше памяти, чем запрашивалось, избыток памяти не используется;
- **Внешняя**: свободная память в процессе выделения или освобождения разделяется на мелкие блоки и в результате не обслуживаются некоторые запросы на выделение памяти.



Внутренняя фрагментация

Память (RAM) –random access memory

Поступает запрос в ОС на выделение блока памяти, длиной N-байт

Система неким образом(любым алгоритмом) выделяет кусок памяти.

В силу того, что алгоритмы выделения кусков памяти разные, часто реально выделается не N-байт, а N+K байт, где К- значение или 0 или вполне реальное.

Все «выделители» памяти работают таким образом, обычно никогда не выделяется ровно столько памяти, сколько запрашивается процессом, т.е. внутри выделенного блока памяти есть неиспользованное пространство (К) - это есть **внутренняя фрагментация –** фрагментация внутри блока. Эти К при использовании многих блоков накапливаются, они вроде бы и есть, но использовать их нельзя.

Память (RAM) –random access memory

занятый кусок ОП

В ОП выделяется много кусков памяти и какие то из них освободились (процессы закончили работать и освободили ОП). В результате получилось 4 занятых куска и 1и2 свободные.

Поступает запрос на выделение большого куска памяти. Если суммировать 1+2 блоки памяти, то вполне хватит, но они разбросаны. Поэтому процессу память не выделится, будет получен отказ.

Возникла **внешняя фрагментация –** по отношению к блоку выделенной памяти она располагается снаружи.

Эволюция памяти

Данный вопрос рассматривается из-за того, что современные аспекты управления ОП сформировались исторически

С самого начала программы напрямую пользовались физической памятью. ОС загружала задание, оно выполнялось, затем ОС выгружала его и загружала следующее.

Большинство встраиваемых систем не имело виртуальной памяти. Во встраиваемых системах обычно работает только одна программа.

Свопинг

По сути это сохранение полного состояния программы на диске. При этом он позволяет запустить другую программу, выполнить ее , а предыдущую сохранить, потом загрузить обратно предыдущую и продолжить ее выполнение.

Исторически свопинг – это замена одной программы на другую.

Эволюция памяти. Мультипрограммирование

Рассмотрим, как это складывалось исторически.

При мультипрограммировании одновременно выполняется несколько процессов и заданий. При этом возникают требования к менеджеру памяти:

Защита: ограничить адресное пространство, используемое процессами.

Быстрая трансляция адресов – это защита не должна тормозить процесс трансляции, не должна вносить задержку.

Быстрое переключение контекста.

Эволюция памяти. Мультипрограммирование

Мультипрограммирование

Введем понятие виртуальных адресов.

Виртуальный адрес – это независимость от физического расположения данных в памяти, т.е. как данные располагаются в памяти как угодно, мы их можем адресовать, используя некоторый виртуальный адрес.

Виртуальный адрес упрощает управление памятью нескольких процессов. Процессорные инструкции используют виртуальные адреса. ЦП преобразует эти виртуальные адреса в физические, используя некоторую помощь от ОС.

Адресное пространство – это множество виртуальных адресов, которые могут использовать процессы. Это было самое начало того, что сейчас называется «виртуальной памятью». Но в данном случае, это гораздо примитивнее.

Эволюция памяти. Метод фиксированных разделов.

Самый простой: ***метод разбивки физической памяти на разделы фиксированной длины***.

Фиксированные – значит заранее определенные, и их размер в процессе работы изменить нельзя.

Аппаратная поддержка в виде регистров база-предел.

Преобразование адресов осуществляется по формуле:

Физический адрес = виртуальный адрес + база

Базовый регистр загружается ОС при переключении процесса.

Эволюция памяти. Метод фиксированных разделов.

Простая защита: Если виртуальный адрес больше база+предел, тогда наступает определенное системой событие – отказ в доступе или выводится ошибка. Есть механизм, который позволяет это отследить.

+(Преимущества):

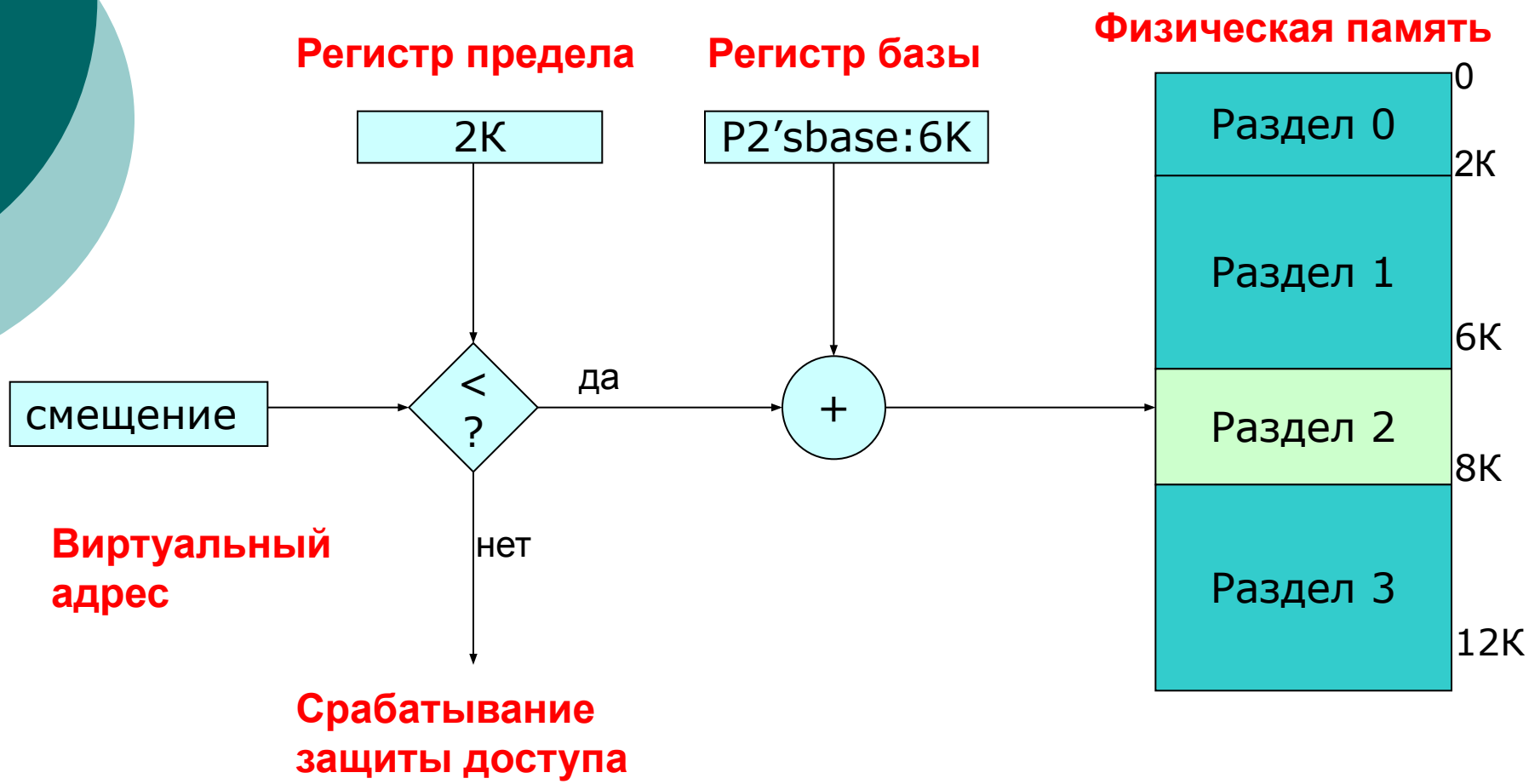
Простой метод

-(Недостатки)

-внутренняя фрагментация – доступный раздел выделяется, как правило больше, чем требуется.

-внешняя фрагментация – когда требуется большой объем памяти, но осталось только 2 маленьких раздела (кусочка)

Метод фиксированных разделов



Метод фиксированных разделов.

Есть виртуальный адрес, он дает нам смещение. Есть регистр предела с которым сравнивают.

Если виртуальный адрес больше регистра предела, то срабатывает защита доступа.

Если меньше, то к нему прибавится регистр базы и получится адрес физической памяти.

Регистр базы на рис. Равен 6Кб. Процесс будет располагаться между 6и8Кб.

Данную предложенную схему необходимо улучшить, а именно: разбивать физическую память на разделы динамически (разделы переменной длины). Аппаратные требования те же: регистр база-предел

Физ.адрес = виртуальный адрес + база

Защита – проверять если физ.адрес больше, чем виртуальный адрес + предел

+(Преимущества)

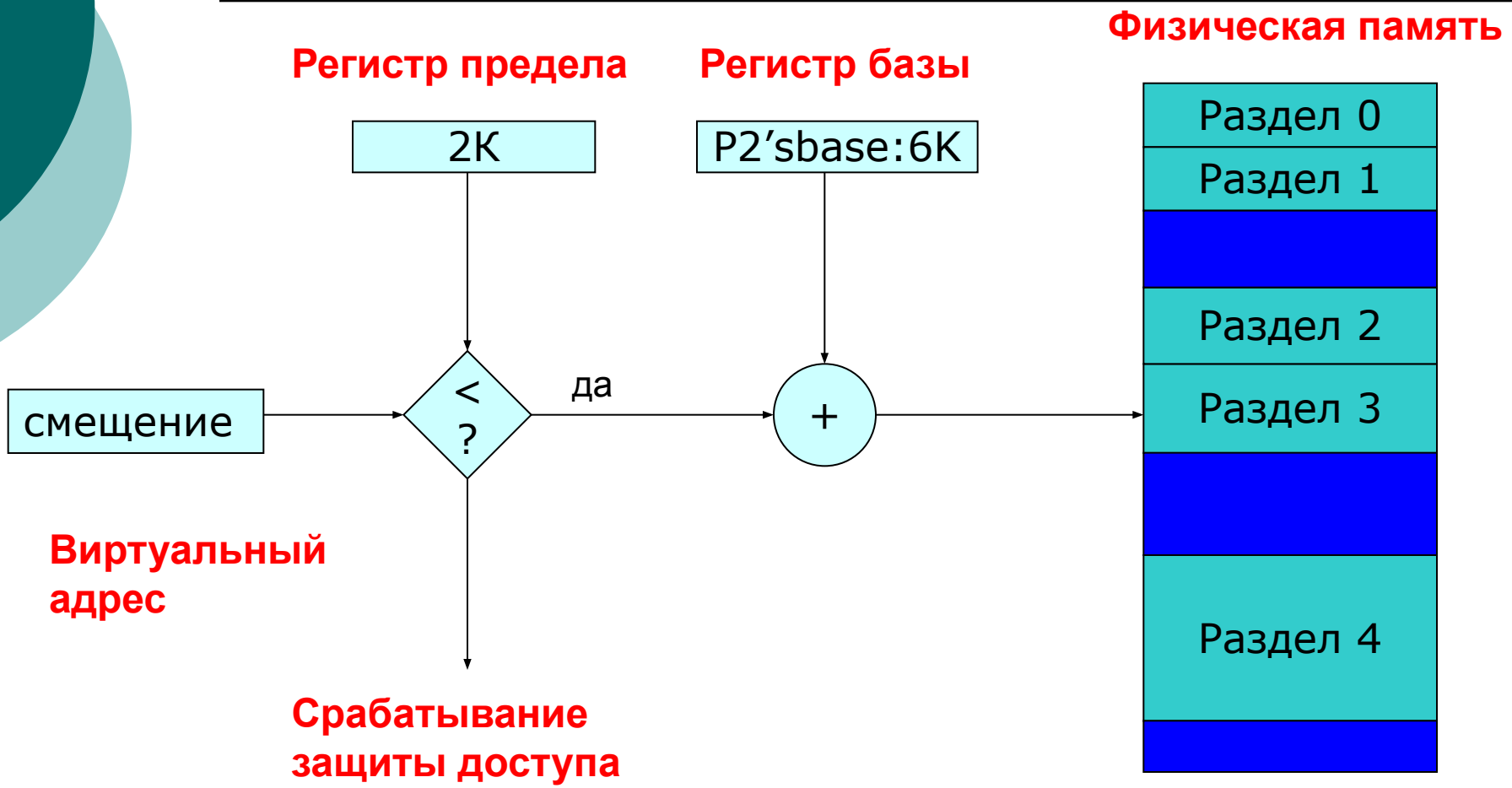
-нет внутренней фрагментации – выделяется столько, сколько запрашивается

-(Недостатки)

-внешняя фрагментация: загрузка/выгрузка задач оставляет необъединяемые «дыры» в памяти.

Все тоже самое, но в памяти появились свободные пространства.

Метод фиксированных разделов

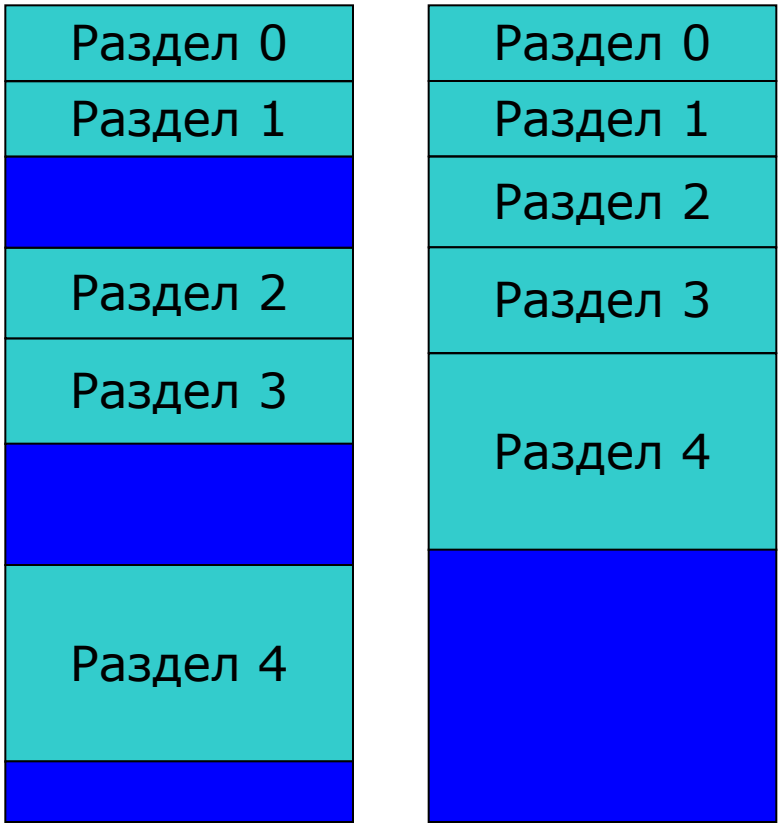


Как бороться с внешней фрагментацией?

На помощь приходит свопинг.

1. Выгрузить программу
2. Загрузить ее по другому адресу
3. Исправить регистр базы

Все поднимается и остается большой кусок памяти для загрузки большой задачи.



Современный подход к решению этой проблемы – **организация памяти в виде страниц.**

Страничная организация памяти

Организация памяти в виде страниц борется с двумя проблемами

- Внешней фрагментацией – используются блоки фиксированного размера в виртуальной и физической памяти, т.е. все запросы на выделение памяти будут кратны, не будет оставаться некратных зон.
- Внутренняя фрагментация – блоки достаточно малого размера, поэтому (К) будет мал.

Страничная организация памяти

С точки зрения программиста:

- Процессам виртуальное адресное пространство предоставляется непрерывным, от байта 0 до байта N
- N зависит от аппаратной поддержки (например 32бит. - адр.пространство 4Гб), делится соответственно.
- В реальности виртуальные страницы распределены по страницам физической памяти далеко не непрерывно и не один к одному. Это два разных мира – физические страницы и виртуальные страницы. Это ключевой аспект, который надо понимать.

Страничная организация памяти

С точки зрения менеджера памяти(+):

- Эффективное использование памяти из-за очень низкой внутренней фрагментации
- Внешняя фрагментация полностью отсутствует и не нужно дефрагментировать

С точки зрения защиты(+):

- Процесс имеет доступ только к своему адресному пространству.

Допущение – все страницы виртуальной памяти всегда находятся в страницах физической памяти. Не будем думать, что есть только виртуальные страницы, а физических – их нет, т.е. полное отображение виртуальной и физической памяти.

Предположим, что все страницы резидентно в памяти, это необходимо, чтобы понять как работает **трансляция адресов**.

Трансляция адресов

Трансляция виртуального адреса:

Виртуальный адрес состоит из двух частей: номер виртуальной страницы (VPN) и смещение внутри страницы

Номер виртуальной страницы (**VPN**- virtual page number) это индекс в таблице страниц (Pagetable)

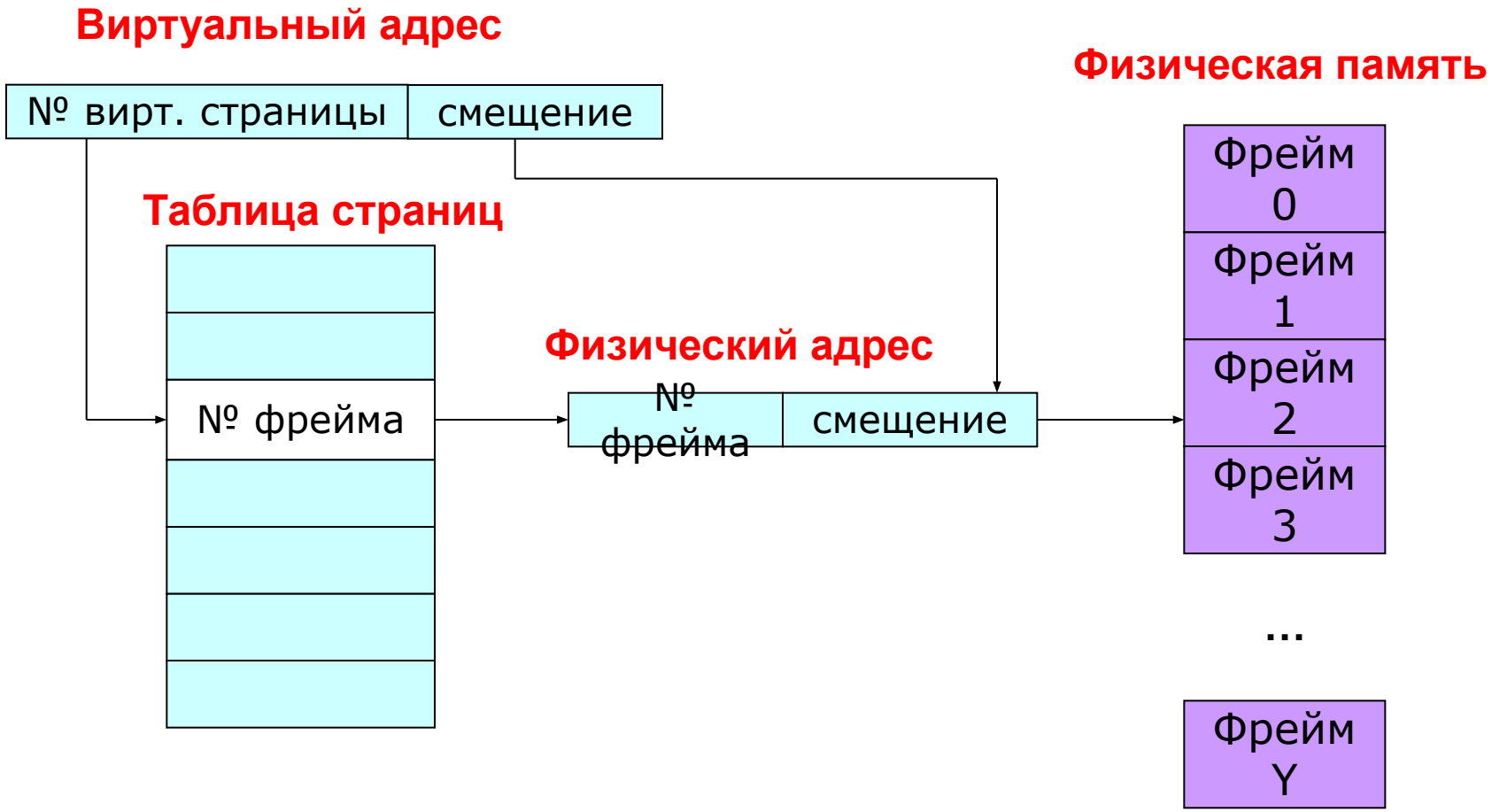
Запись в таблице страниц (**PTE** – page table entry) содержит номер фрейма (**PFN** –page frame number)

Номер фрейма – это номер физической страницы.

Фрейм – это страница физической памяти.

Смысл таблицы страниц – одна запись в таблице страниц (PTE) на одну страницу виртуального адресного пространства (VPN), отображает VPN на PFN. Какая **виртуальная страница** соответствует какому **фрейму физической** памяти.

Трансляция адресов



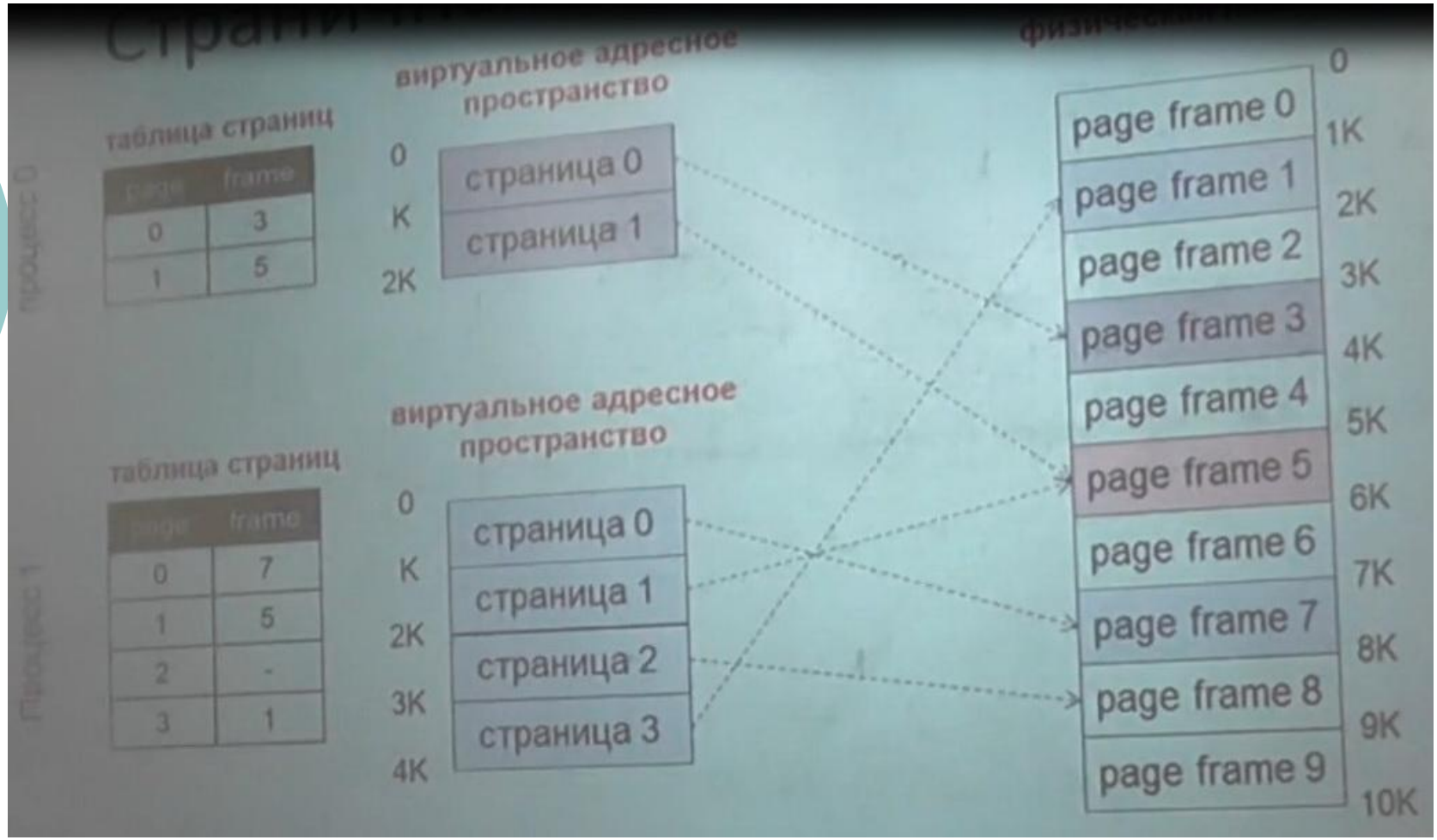
Трансляция адресов

Есть виртуальный адрес, состоящий из двух частей:

1) №вирт.страницы, по нему идет поиск в таблице страниц и находится № фрейма, он составляет одну часть физического адреса
2) Смещение – берется напрямую – вторая часть физического адреса.

По новому адресу осуществляется поиск уже в физической памяти и доступ к данным

Рис. Страничная организация памяти



Существуют 2 процесса 0 и 1, у 0 есть 2 страницы у 1процесса 4 страницы. И Мы видим, что они могут отображаться как угодно, вплоть до того, что 2 виртуальные страницы могут отображаться на одной физической. Зачастую это бывает полезно, простор для манипуляций ограниченный и в этом состоит вся прелесть виртуального адресного пространства.

Страничная организация памяти. Пример

-32битная разрядность адресов

-Размер страницы 4096байт

-VPN длиной 20бит, смещение 12бит (20+12=32бита)

-Преобразуем виртуальный адрес **0* 43456 323**

- **№ вирт. страницы смещение внутри страницы**

-VPN=43456 смещение=323

-Допустим, что в ячейке таблицы страниц по индексу 0*43456 находится значение фрейма PFN= 0*1002. Получаем физический адрес 0*01002323

таблица страниц (PTE)

Если есть таблица страниц, которая содержит преобразование адреса, то необходимо воспользоваться и нагрузить ее дополнительными функциями:

- Добавить защиту доступа
- Добавить дополнительную вспомогательную информацию (например, используется эта вирт.страница или нет, был ли к ней когда-либо осуществлен доступ, была ли в нее осуществлена запись…)

Таблица страниц превращается в сложную структуру данных, которые начинает использоваться множеством всяких дополнительных полей.

Все это нужно, чтобы сохранить память для других процессов, делать все быстро и не плодить десятки новых таблиц с данными – все по возможности хранить внутри таблицы PTE.

таблица страниц (PTE)

Если мы посмотрим на запись в PTE, то мы увидим, что туда можно поместить.

V - может ли использоваться данная запись PTE (valid or not) – бит валидности, бит присутствия

R - был ли доступ к этой странице

M – была ли страница модифицирована

P – какие операции разрешены (битовая маска операций)

PFN – номер фрейма (как основной, как осн.нагрузка)

Преимущества страничной памяти

- Легко выделять физическую память
  - Списки свободных фреймов, выделить фрейм – просто удалить из списка свободных
  - Внешняя фрагментация не проблема, т.к. фреймы одного размера
- Естественный подход
  - Всей программе не нужно быть резидентной – это «побочный» продукт
  - Все страницы одного размера
  - Основа – устранение внешней фрагментации

Недостатки страничной памяти

- Внутренняя фрагментация
  - Процессам может быть нужны размеры, некратные размеру страницы
  - По сравнению с размером адресного пространства, размер страницы очень мал.
- Накладные расходы при обращении к памяти – вначале к таблице страниц, а затем уже к памяти.

Решение: аппаратный КЭШ для обращений к таблице страниц (**TLB** translation lookaside buffer – буфер внутри процессора)

Недостатки страничной памяти

- Большой объем памяти, требуемый для хранения таблиц страниц
- 1 PTE на 1 страницу в виртуальном адресном пространстве

Пример: Архитектура x86

- 32-битное АП с 4КБ страницами = 2^{20} PTE= 1048576записей PTE
- 4 байта на PTE = 4Мб памяти на таблицу страниц
- В Ос создаются отдельные таблицы страниц для каждого процесса, итого, например 25процессов * 4Мб = 100Мб
- Соответственно нужны отдельные таблицы страниц для каждого процесса.

Решение: хранить таблицы страниц в страничной памяти)

Страничная организация памяти (обобщение)

- Решает разные проблемы, типа фрагментации
- Адресное пространство – линейный массив байтов
- Разделяется на страницы одинакового размера (например 4Кб)
- Использует таблицы страниц для отображения виртуальной страницы на физический фрейм

Сегментация

- Разделяет адресное пространство на логические блоки (стек, код, куча…)
- Виртуальный адрес в виде – сегмент + смещение

Страничная организация предполагает куски памяти, как в примере выше, 4096байт (4Кб), таких кусков можно брать сколько угодно.

Сегментация подходит к распределению памяти более «продвинуто» - есть разные логические сегменты памяти: стек, код программы - имеют свой размер, расположение и права доступа, куча – для динамической памяти программы – все это **отдельные логические блоки памяти**. Их не нужно мешать в одно, поэтому следующим этапом было разделение всего названного и ввод виртуального адреса в виде пары: **Сегмент, смещение**

Сегментация. Зачем?

- Позволяет разделить разные участки памяти в соответствии с их назначением
- Динамический (изменяемый) размер у сегментов

2варианта

- 1 сегмент на процесс – переменный раздел
- Много сегментов на процесс - сегментация

Аппаратная поддержка сегментации

- Одна пара база-предел(лимит) на сегмент
- Сегменты идентифицируются №, который является индексом в таблице
- Виртуальный адрес = пара <СЕГМЕНТ, СМЕЩЕНИЕ>
- Физический адрес = база сегмента + смещение

Недостатки (-)

- Все недостатки, присущие организации памяти разделами переменной длины присущи и сегментной организации
- Внешняя фрагментация

Аппаратная поддержка сегментации

- Лучшим, как можно заметить является организация этих подходов в один. Давайте объединим страничную и сегментную адресацию
- Архитектура x86 поддерживает и страничную и сегментную адресацию
- Сегменты используются для управления логическими блоками, обычно сегменты большие (помещают множество страниц)
- Сегменты разбиваются на страницы, у каждого сегмента своя таблица страниц

В современных ОС достаточно ограниченно применяются сегменты, много их не применяется.

Пример ОС LUNIX

- 1 сегмент кода ядра, 1 сегмент данных ядра
- 1 сегмент кода пользователя, 1 сегмент данных пользователя
- Все сегменты организованы странично.

Страничная виртуальная память

- Мы предполагали ранее, что вся память резидентная (не рассматривался вариант, что какой-либо страницы может не быть в физ.памяти.
- Сейчас, допустим, что адресное пространство может и не быть полностью резидентным. На практике так в основном и есть – память никогда не резидентна. Процессов сотни, на них выделяется много вирт.памяти и ее не хватит на отображение в физ. Памяти.
- Например 100 процессов 100П*4Гб=400ГБ ОП.
- Соответственно какая то часть адресного пространства находится в физ.памяти, какая то часть в некоторой вторичной памяти (понимается дисковая подсистема), абстрагируется от физ. реализации.

Страничная виртуальная память

- С точки зрения ОС важно, что ОС использует основную память, как КЭШ. У ОС есть медленная память (это дисковая подсистема), она может туда загружать процессы и выгружать их оттуда, а основная память используется как ограниченный ресурс, сродни классическому кэшированию.
- Принцип работы достаточно простой – нужная страница перемещается в свободный фрейм физической памяти из вторичной памяти.
- А если свободных фреймов нет, то какая-либо страница выгружается на диск, т.о. освобождается драгоценный фрейм физ. Памяти.
- Важно отметить, запись во вторичной памяти происходит только тогда, когда страница в основной памяти была модифицирована, если она не была модифицирована, то соответственно и выгружать нечего.
- Весь процесс происходит прозрачно для программы. Никакая программа не управляет напрямую, какую страницу ей выгрузить в основную память, какую загрузить.
- Менеджер памяти ОС все абстрагирует и делает все прозрачным для программиста, чтобы он не «напрягался» на данную тему, это сложно реализовывать самостоятельно.

Page fault – Страничное прерывание

- Процесс обращается к виртуальному адресу на выгруженной (или загруженной) странице, т.е. страница в принципе отсутствует, при этом обращении и происходит страничное прерывание.

Как этот процесс весь работает:

- Когда страница выгружается, ОС устанавливает **бит присутствия** (valid – является ли валидной ячейка) PTE=**0**. И там же в PTE записывает куда она была соответственно выгружена.
- Когда же процесс обращается к этой странице, то происходит **исключение**, т.к. Valid=0,т.е. бит валидности установлен в 0 – страница не использовалась.
- После того, как произошло исключение ОС передает управление **обработчику страничного прерывания**
- Обработчик находит то **место**, куда была выгружена страница
- Считывает эту страницу в фрейм физической памяти, обновляет PTE, ставит бит валидности (присутствия) в 1.
- В физической памяти появляется новая страница.

Процесс достаточно простой, если страницы нет, то сама же ОС ее загрузит обратно, или выгрузит, когда ей это надо.

Загрузка по требованию

Еще один ключевой механизм работы менеджера памяти. Практически все страницы памяти загружаются по требованию.

Смысл: страницы загружаются в основную память только тогда, когда к ним происходит **непосредственное обращение**.

Предсказать заранее, какая страница потребуется в будущем – сложно(это как гадать на кофейной гуще).

Сегодня для этого разработано множество алгоритмов, есть разные подходы.

Самый логичный из них - **кластеризация страниц.**

ОС ведет учет страниц, которые обычно загружаются вместе, даже если они расположены в различных местах виртуальной памяти или физической памяти.

Если идет обращение к одной их них, то ОС **загружает все страницы их этого кластера**. Это естественный подход к тому, как можно провести исходную оптимизацию этого алгоритма.

Можно даже предоставить программисту возможность определять такие кластеры.

Механизм замещения страниц

Допустим у нас заняты все фреймы(страницы) физической памяти, а нам нужно еще загрузить одну страницу.

? – какую из страниц физ. памяти выгрузить? Ведь страниц много и это выбора многое зависит.

Алгоритм замещения страниц простой:

1. Выбрать страницу, которую не понадобится в ближайшем бедующем
2. Выбрать страницу, которая не была модифицирована, чтобы обойтись без записи ее на диск.
3. Чтобы обойтись без замены в ОС есть пул свободных страниц.

Но проблема замещения страниц существует.

Посмотрим последовательность действий в ОС, когда у нас загружается программа.

Как загружается программа?

1. Для создания нового процесса создается новый PCB (процесс контрол.блок).
2. Создается таблица страниц для этого процесса, которая описывает его виртуальное адресное пространство
3. Образ программы на диск размещается блоками в адресное пространство (он не копируется, а неким образом размещается)
4. Строится таблица страниц, указатель на которую уже есть в PCB

   Все PTE имеют бит присутствия =0 (виртуальное адресное пространство в начале процесса полностью пустое, в плане отображения на физическую память – ни одна страница не присутствует)

   В эти же PTE заносится информация о нахождении этих физических страниц уже во вторичной памяти, т.е. на диске
5. Передает управление на точку входа этого процесса – исполнение первой же инструкции приводит к страничному прерыванию (page fault)
6. Обработчик прерывания загружает эти страницы из дисковой памяти

Т.О. никто, ничего заранее не подгружает, все загружается через «page fault» по требованию. Именно так все работает.

Механизм замещения страниц

Принцип локальности определяется 2 подходами:

- Одна загрузка –много обращений (**локальность по времени)**
- Обращение к страницам рядом с уже загруженной (**локальность по расположению**).

Загрузка страниц по требованию может быть частой, а может и не такой частой, **зависит от**:

- Локальности
- Политики замещения страниц
- Объема физической памяти
- «рабочего множества», так называемого

Смысл алгоритма замещения страниц – уменьшить число страничных прерываний (page fault) путем выбора **лучшей страницы** для выгрузки.

Лучшая – та, к которой **вообще больше не будет обращений**, т.е. освобождает место в памяти и решаем все вопросы, все будет работать быстро.

Далее рассмотрим разные алгоритмы, которые были предложены к решению данной проблемы (фундаментальной и основополагающей, как оказалось). Это как в алгоритмах планирования, они существенно влияют на производительность, отзывчивость системы. В менеджере памяти тоже есть ниша, которая оказывает большое влияние.

Алгоритм Belady

Оптимальность его доказана по критерию наименьшего числа страничных прерываний.

-**Выгрузить страницу, которая дольше всех не будет использоваться в будующем**

- Проблема – предсказать будущее

Физический смысл АВ на практике реализовать сложно, а по сути нельзя. Его используют в качестве **эталона для сравнения.**

Если Вы реализовали какой-то другой алгоритм практически в ОС и он оказался ненамного хуже, чем АВ, что значит Вы сделали очень хороший алгоритм.

Но на самом деле не существует «лучшего» алгоритма, т.к. все зависит от задачи. Например, при планировании при интерактивных процессов один алгоритм, для фоновых – другой. Чаще используется объединение этих всех алгоритмов.

Не существует худшего алгоритма, но в качестве эталона «худшего» можно указать: - Случайный выбор страницы для замены, но случайный не всегда худшее.

Рассмотрев теоретический АВ первым был сделан алгоритм FIFO

FIFO

Достаточно прост в реализации:

- **При загрузке страницы мы помещаем ее в конец списка**
- **Выгружаем страницы из начала списка.**

Преимущества (+)

- Выгружается самая старая страница, может быть и редко использованная

Недостатки (-)

- Выгруженная страница может все еще использоваться, хотя и давно загрузилась, нет уверенности что она редко используемая.
- Плохая производительность

Существует понятие **аномалия Belady** – если дать процессу больше физической памяти, то определенные последовательности обращений к страницам приведет к ***увеличению числа страничных прерываний***.

Есть определенная ограниченная физическая память, когда выполняется процесс он генерирует страничные прерывания. При использовании алгоритма FIFO при определенных случаях, если увеличить размер физической памяти, то этих страничных прерываний станет еще больше. Это доказал Belady и названо явление аномалией Belady.

Алгоритм LRU (Last recently used) – наименее используемое

Использует информацию об обращениях к страницам памяти для принятия решения о ее выгрузке

Смысл: угадать будущее на основе прошлого. При завершении выгрузить ту **страницу**, которая **дольше всех не использовалась**.

- При сравнении с АВ, то АВ заглядывает в будущее, LRU в прошлое.

Реализация
Алгоритм LRU (Last recently used)

В идеале нужно **сохранять время** при каждом обращении к памяти в PTE, сортировать на основе этих данных. И та страница, к которой дольше всех не было обращений выбирается первым кандидатом на выгрузку.

Если каждое обращение к памяти будет сопровождаться записью значения времени последнего обращения к памяти, то система начнет **тормозить**. Преимущество LRU сойдет на нет, все окажется слишком долго.

На практике LRU «в чистом виде» не реализуется. Но существуют аппроксимации, спасает математический подход ко всему, можно реализовать **различные приближения** к алгоритму LRU.

Аппроксимации алгоритма LRU

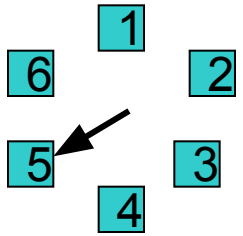
Существует бит в PTE, который называется **referenced** (0 или 1). Вводим счетчик для каждой страницы.

Алгоритм аппроксимации: регулярно для каждой страницы через какие-то равные промежутки времени смотрим:

- если бит **referenced =0,** то мы увеличим некий счетчик
- если бит **referenced =1,** то счетчик обнулим
- И через равные промежутки времени всегда обнуляем бит **referenced.** Т.О. эта аппроксимация приводит к тому, что счетчик будет содержать количество временных интервалов с момента последнего обращения к памяти к этой странице.
- В некоторых архитектурах этого бита в PTE нет, поэтому применяются всякие «некрасивые» ухищрения. Бит **referenced** устанавливается автоматически процессором, когда происходит доступ к памяти, чтобы было достаточно быстро.

Not Frequently Used (алгоритм clock – часовой)

Суть Алгоритма: он замещает достаточно старую страницу. Фреймы физ.памяти организуются в виде часов. Есть часовая стрелка и по кругу фреймы физ.памяти. Стрелка выбирает кандидата на выгрузку.



Если бит ref=0, то выгружает эту страницу
Если бит ref=1, то обнуляет его и идет дальше
Действия происходят через регулярные промежутки времени.
Если памяти много, то ниже затраты и ниже точность.
Изначально в список «часов» помещаются только те страницы, которые используются, т.е. те, у которых ref=1. Стрелка идет по кругу и начинает обнулять ref и соответственно выгружать страницы при необходимости. Поэтому из круга выпадают только те, которые не были недавно использованы.

Задача. Распределение фреймов физ.памяти между процессами.

Существуют 2 подхода

- **Локальный алгоритм замещения**
  - У каждого процесса свой лимит на число страниц, который он может использовать
  - Выгружает только свои страницы
- **Глобальный алгоритм замещения**
  - Неважно, чьи страницы кому принадлежат – алгоритм замещения идет по большому кругу и выгружает независимо от владельца

Примеры из практики:

Lunix использует глобальное замещение

Существуют гибридные алгоритмы – локальное замещение и механизм явного добавления/удаления фреймов

Рабочее множество

РМ-**Woking set** – необходимо для моделирования локальности использования памяти.

Раб. множество – это множество тех страниц процесса, которые ему сейчас нужны.

Формальное определение РМ ввел Петр Деннинг в 1968г.

Эти алгоритмы используются во всех современных ОС.

WS(t,w) ={страницы P, к которым были обращения за интервал (t-w;t) }

t - время w - окно рабочего множества

«Сейчас» - в определении раб. множества это имеет какую-то точку во времени, «когда сейчас?» и имеет некоторую длительность.

Страница находилась в раб. множестве тогда, когда к ней обращались за последние **W** единиц времени.

Раб. множество (**WS**) изменяется во времени и размер WS также меняется, это плавающие величины.

Размер рабочего множества

- Введем обозначение **|WS(t,w)|** - зависит от локальности
- При плохой локальности, подгружается много страниц.
- |WS(t,w)| в этот момент увеличивать

Очевидно, что WS должно быть в памяти резидентно, иначе случится «пробуксовка»(trashing), компьютер затормаживается, т.е. постоянные страничные прерывания для того, чтобы подгрузить страницу, так как процесс без нее работать не сможет, он будет к ней обращаться.

Перед тем как запустить процесс мы должны оценить размер его исходного рабочего множества.

1) Оценить |WS(0,w)| для процесса
2) Запустить этот процесс только тогда, если есть столько фреймов физ. Памяти, чтобы туда поместилось это исходное раб. Множество.
3) Используя алгоритм замещения страниц загружать нужные страницы
4) Динамически отслеживать размеры рабочего множества процессов.

Алгоритм Page Fault Frequency

- **Page Fault Frequency** – частоты возникновения страничных прерываний
- Он в корне отличается от всех остальных, т. к. пытается уровнять число страничных прерываний между всеми процессами и снизить их общее число.

За счет чего он это делает?

- Следит за частотой страничных прерываний каждого процесса
- Если оно выше определенного порога – предоставляет процессу больше памяти
- Если оно меньше, то забирает у него память.
- Т.О. он перераспределяет память между процессами и регулирует число страничных прерываний.

Пробуксовка (thrashung)

- Она происходит, когда система проводит большую часть времени, обслуживая страничные прерывания, а не занимаясь полезной работой – вычислениями.

Из-за чего это происходит, причины:

- ✔ Из-за неправильного алгоритма замещения страниц при достаточном объеме памяти
- ✔ Либо если объема физ. Памяти недостаточно, слишком много активных процессов

Это ненормально состояние, которое должно решаться каким-то образом.