# MMU в картинках (часть 1)

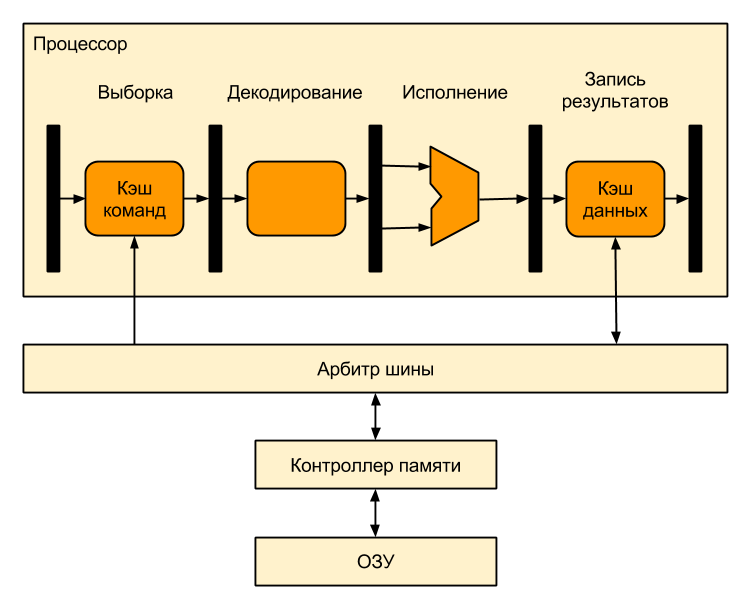
Хочу поговорить об устройстве управления памятью (Memory Management Unit, MMU). Как вы, разумеется, знаете, основной функцией MMU является аппаратная поддержка виртуальной памяти. Словарь по кибернетике под редакцией академика Глушкова говорит нам, что виртуальная память — это воображаемая память, выделяемая операционной системой для размещения пользовательской программы, ее рабочих полей и информационных массивов.  
  
У систем с виртуальной памятью четыре основных свойства:

1. Пользовательские процессы изолированы друг от друга и, умирая, не тянут за собой всю систему
2. Пользовательские процессы изолированы от физической памяти, то есть знать не знают, сколько у вас на самом деле оперативки и по каким адресам она находится.
3. Операционная система гораздо сложнее, чем в системах без виртуальной памяти
4. Никогда нельзя знать заранее, сколько времени займет выполнение следующей команды процессора

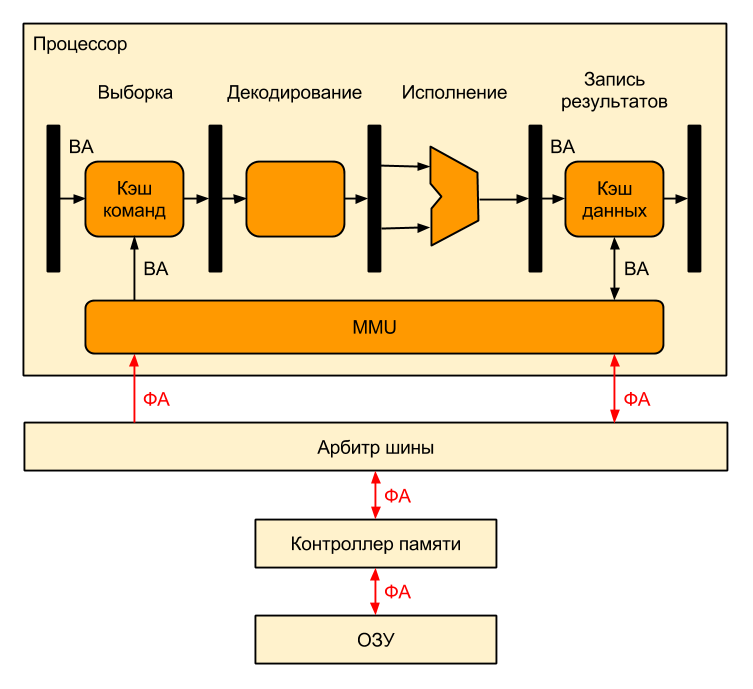
Выгода от всех вышеперечисленных пунктов очевидна: миллионы ~~криворуких~~ прикладных программистов, тысячи разработчиков операционных систем и несчетное число эмбеддеров благодарны виртуальной памяти за то, что все они до сих пор при деле.  
  
К сожалению, по какой-то причине все вышеперечисленные товарищи недостаточно почтительно относятся к MMU, а их знакомство с виртуальной памятью обычно начинается и заканчивается изучением страничной организации памяти и буфера ассоциативной трансляции (Translation Lookaside Buffer, TLB). Самое интересное при этом остается за кадром.  
  
Не хочу повторять Википедию, поэтому если вы забыли даже про то, что такое страничная память, то самое время перейти по[ссылке](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C). Про TLB будет пара строк ниже.

Устройство MMU

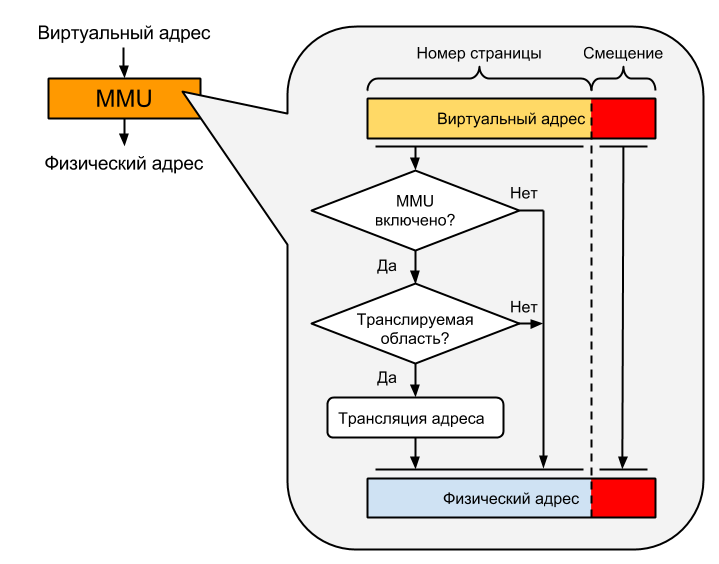
А теперь перейдем к делу. Вот так выглядит процессор без поддержки виртуальной памяти:



Все адреса, используемые в программе для такого процессора — реальные, «физические», т.е. программист, линкуя программу, должен знать, по каким адресам находится оперативная память. Если у вас припаяно 640 кБ оперативки, отображаемой в адреса 0x02300000-0x0239FFFF, то все адреса в вашей программе должны попадать в эту область.  
  
Если программист хочет думать, что у него всегда четыре гигабайта памяти (разумеется, речь идет о 32-битном процессоре), а его программа — единственное, что отвлекает процессор ото сна, то нам потребуется виртуальная память. Чтобы добавить поддержку виртуальной памяти, достаточно между процессором и оперативной памятью разместить MMU, которое будет транслировать виртуальные адреса (адреса, используемые в программе) в физические (адреса, попадающие на вход микросхем памяти):



Такое расположение очень удобно — MMU используется только тогда, когда процессор обращается к памяти (например, при промахе кэша), а все остальное время не используется и экономит электроэнергию. Кроме того, в этом случае MMU почти не влияет на быстродействие процессора.  
  
Вот что происходит внутри MMU:

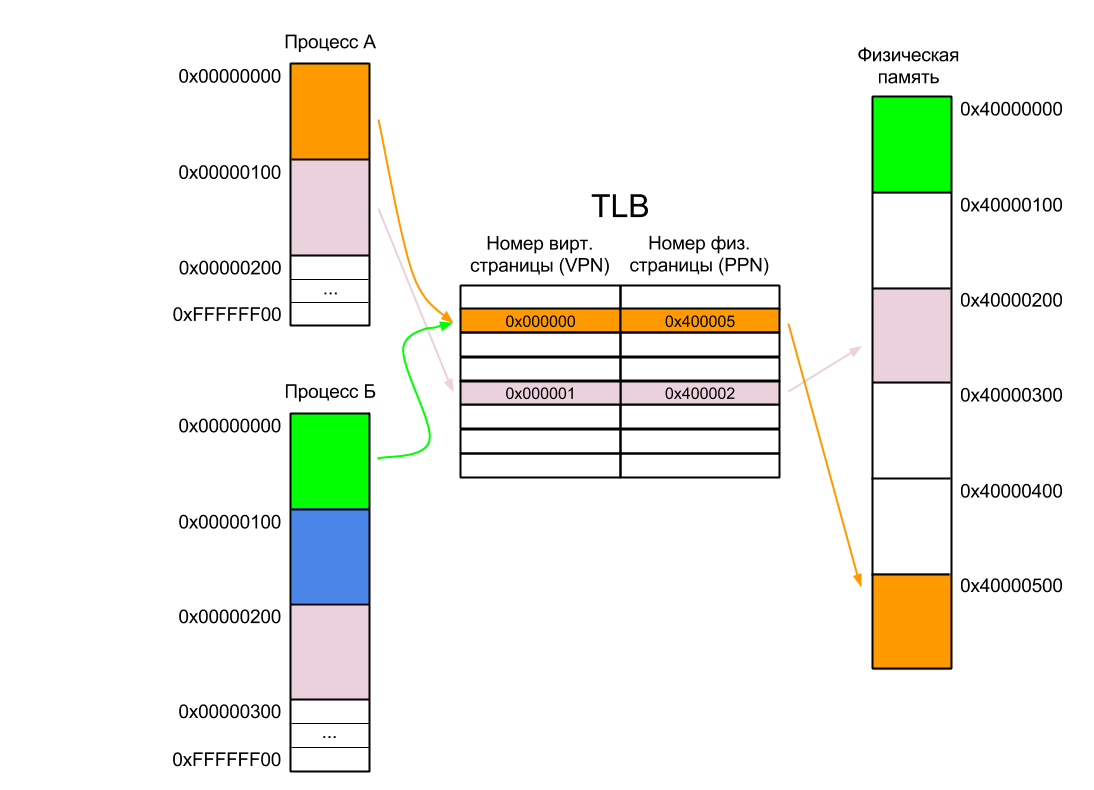


Выглядит этот процесс так:

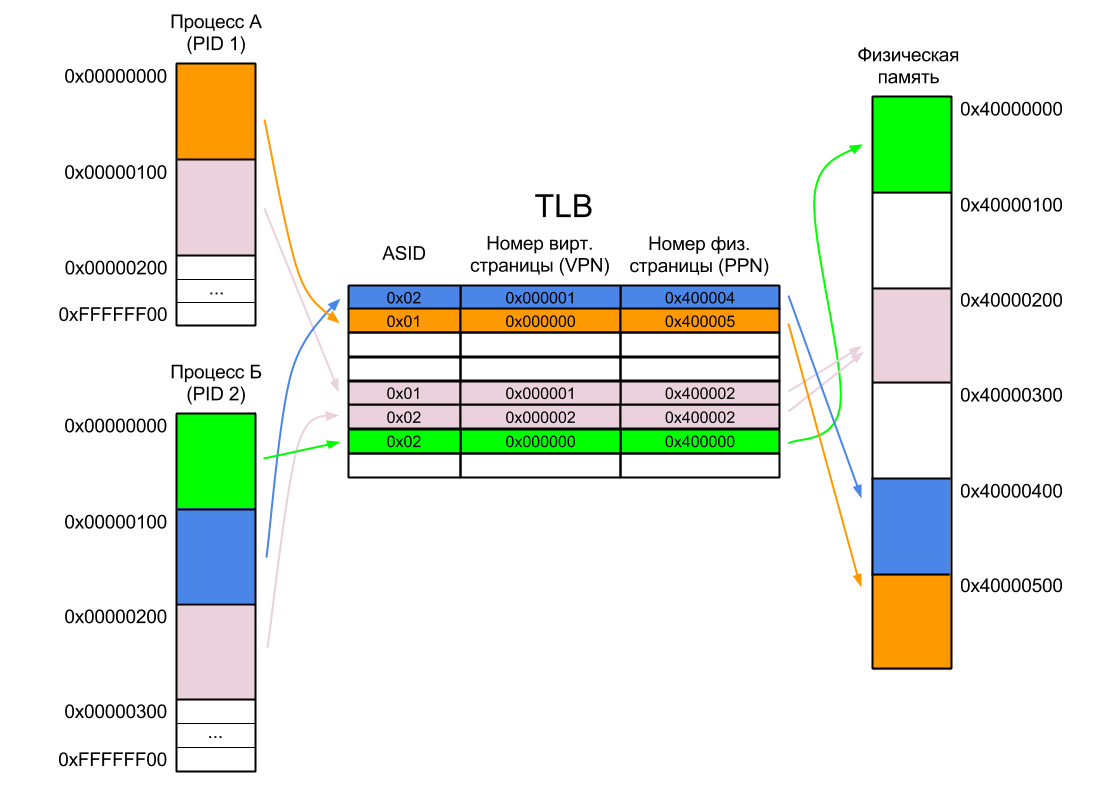
1. Процессор подает на вход MMU виртуальный адрес
2. Если MMU выключено или если виртуальный адрес попал в нетранслируемую область, то физический адрес просто приравнивается к виртуальному
3. Если MMU включено и виртуальный адрес попал в транслируемую область, производится трансляция адреса, то есть замена номера виртуальной страницы на номер соответствующей ей физической страницы (смещение внутри страницы одинаковое):  
   * Если запись с нужным номером виртуальной страницы есть в TLB, то номер физической страницы берется из нее же
   * Если нужной записи в TLB нет, то приходится искать ее в таблицах страниц, которые операционная система размещает в нетранслируемой области ОЗУ (чтобы не было промаха TLB при обработке предыдущего промаха). Поиск может быть реализован как аппаратно, так и программно — через обработчик исключения, называемого страничной ошибкой (page fault). Найденная запись добавляется в TLB, после чего команда, вызвавшая промах TLB, выполняется снова.

Лирическое отступление об MPU

Кстати, не путайте MMU и MPU (Memory Protection Unit), которое часто используют в микроконтроллерах. Грубо говоря, MPU — сильно упрощенное MMU, из всех функций обеспечивающее только защиту памяти. Соответственно, TLB там нет. При использовании MPU адресное пространство процессора делится на несколько страниц (например, на 32 страницы по 128 мегабайт для 32-битного процессора), для каждой из которых можно установить индивидуальные права доступа.

Рассмотрим работу TLB на простом примере. Допустим, у нас есть два процесса А и Б. Каждый из них существует в своем собственном адресном пространстве и ему доступны все адреса от нуля до 0xFFFFFFFF. Адресное пространство каждого процесса разбито на страницы по 256 байт (это число я взял с потолка — обычно размер страницы не меньше одного килобайта), т.е. адрес первой страницы каждого процесса равен нулю, второй — 0x100, третьей — 0x200 и так далее вплоть до последней страницы по адресу 0xFFFFFF00. Разумеется, процесс не обязательно должен занимать все доступное ему пространство. В нашем случае Процесс А занимает всего две страницы, а Процесс Б — три. Причем одна из страниц общая для обоих процессов.  
  
Также у нас есть 1536 байт физической памяти, разбитой на шесть страниц по 256 байт (размер страниц виртуальной и физической памяти всегда одинаков), причем память эта отображается в физическое адресное пространство процессора с адреса 0x40000000 (ну вот так ее припаяли к процессору).  
  
  
  
В нашем случае первая страница Процесса А расположена в физической памяти с адреса 0x40000500. Не будем вдаваться в подробности того, как эта страница туда попадает — достаточно знать, что ее загружает операционная система. Она же добавляет запись в таблицу страниц (но не в TLB), связывая эту физическую страницу с соответствующей ей виртуальной, и передает управление процессу. Первая же команда, выполненная Процессом А, вызовет промах TLB, в результате обработки которого в TLB будет добавлена новая запись.  
Когда Процессу А потребуется доступ ко второй его странице, операционная система загрузит ее в какое-нибудь свободное место в физической памяти (пусть это будет 0x40000200). Случится еще один промах TLB, и нужная запись снова будет добавлена в TLB. Если места в TLB нет — будет перезаписана одна из более ранних записей.  
  
После этого операционная система может приостановить Процесс А и запустить Процесс Б. Его первую страницу она загрузит по физическому адресу 0x40000000. Однако, в отличие от Процесса А, первая команда Процесса Б уже не вызовет промах TLB, так как запись для нулевого виртуального адреса в TLB уже есть. В результате Процесс Б начнет выполнять код Процесса А! Что интересно, именно так и работал широко известный в узких кругах процессор ARM9.  
  
Самый простой способ решить эту проблему — инвалидировать TLB при переключении контекста, то есть отмечать все записи в TLB как недействительные. Это не самая хорошая идея, так как:

* В TLB на этот момент занято всего две записи из восьми, то есть новая запись поместилась бы туда без проблем
* Удаляя из TLB записи, принадлежащие другим процессам, мы вынуждаем эти процессы генерировать повторные страничные ошибки, когда операционная система снова запустит их. Если в TLB не восемь записей, а тысяча, то производительность системы может значительно упасть.

Способ посложнее — линковать все программы так, чтобы они использовали разные части виртуального адресного пространства процессора. Например, Процесс А может занимать младшую часть (0x0-0x7FFFFFFF), а Процесс Б — старшую (0x80000000-0xFFFFFFFF). Очевидно, что в этом случае ни о какой изоляции процессов друг от друга речи уже не идет, однако этот способ действительно иногда применяют во встроенных системах. Для систем общего назначения по понятным причинам он не подходит.  
  
Третий способ — это развитие второго. Вместо того, чтобы делить четыре гигабайта виртуального адресного пространства процессора между несколькими процессами, почему бы просто не увеличить его? Скажем, в 256 раз? А чтобы обеспечить изоляцию процессов, сделать так, чтобы каждому процессу по-прежнему было доступно ровно четыре гигабайта оперативной памяти?  
Сделать это оказалось очень просто. Виртуальный адрес расширили до 40 бит, при этом старшие восемь бит уникальны для каждого процесса и записаны в специальном регистре — идентификаторе процесса (PID). При переключении контекста операционная система перезаписывает PID новым значением (сам процесс свой PID поменять не может).  
Если для нашего Процесса А PID равен единице, а для Процесса Б — двойке, то одинаковые с точки зрения процессов виртуальные адреса, например 0x00000100, с точки зрения процессора оказываются разными — 0x0100000100 и 0x0200000100 соответственно.  
Очевидно, что в нашем TLB теперь должны находиться не 24-битные, а 32-битные номера виртуальных страниц. Для удобства восприятия старшие восемь бит хранятся в отдельном поле — идентификаторе адресного пространства (Address Space IDentifier — ASID).  
  
Теперь, когда процессор подает на вход MMU виртуальный адрес, поиск в TLB производится по комбинации VPN и ASID, поэтому первая команда Процесса Б вызовет страничную ошибку даже без предшествующей инвалидации TLB.  
  
  
  
В современных процессорах ASID чаще всего или восьмибитный, или 16-битный. Например, во всех процессорах ARM с MMU, начиная с ARM11, ASID восьмибитный, а в архитектуре ARMv8 добавлена поддержка 16-битного ASID.  
  
Кстати, если процессор поддерживает виртуализацию, то помимо ASID у него может быть еще и VSID (Virtual address Space IDentificator), который еще больше расширяет виртуальное адресное пространство процессора и содержит номер запущенной на нем виртуальной машины.  
  
Даже после добавления ASID могут возникнуть ситуации, когда нужно будет инвалидировать одну или несколько записей или даже весь TLB:

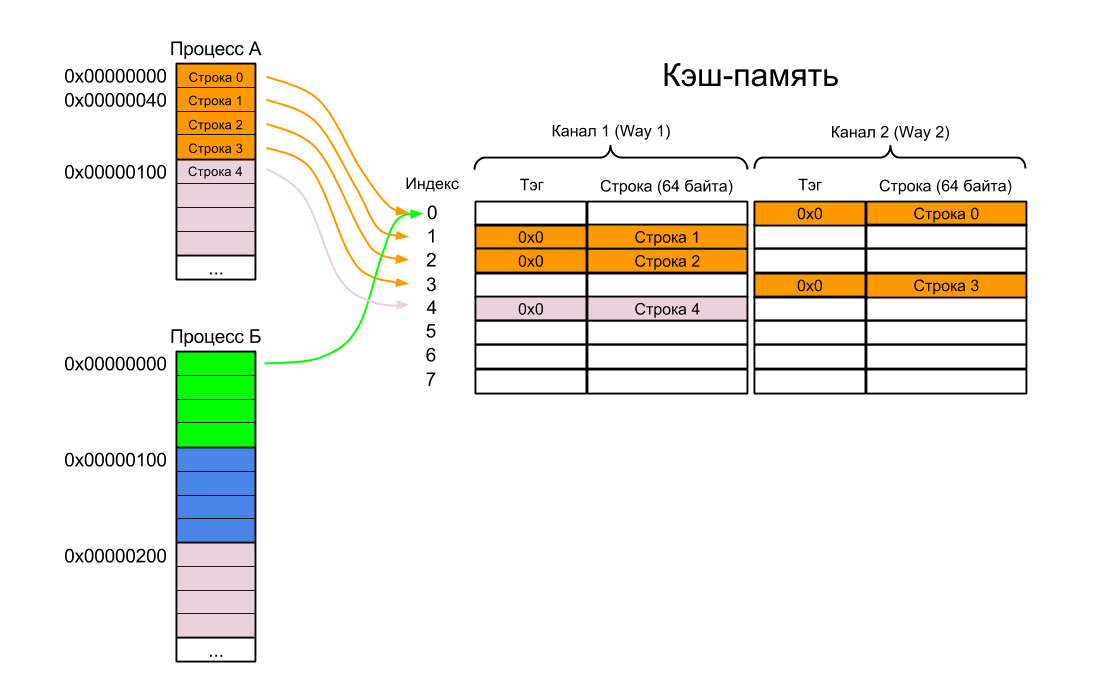
1. Если физическая страница выгружена из оперативной памяти на диск — потому что обратно в память эта страница может быть загружена по совсем другому адресу, то есть виртуальный адрес не изменится, а физический изменится
2. Если операционная система изменила PID процесса — потому что и ASID станет другим
3. Если операционная система завершила процесс

На этом можно было бы завершать рассказ об MMU, если бы не один нюанс. Дело в том, что между процессором и оперативной памятью помимо MMU находится еще и кэш-память.  
  
Те, кто забыл, что такое кэш-память и как она работает, могут освежить знания [тут](http://habrahabr.ru/post/179647/) и [там](http://habrahabr.ru/post/187654/).   
  
Для примера возьмем двухканальный (2-way) кэш размером один килобайт с размером строки кэша (или линии кэша — как вам угодно) в 64 байта. Сейчас не важно, кэш ли это команд, кэш данных или объединенный кэш. Поскольку размер страницы памяти у нас 256 байт, то в каждой странице помещается четыре строки кэша.  
  
Поскольку кэш находится между процессором и MMU, то очевидно, что и для индексации, и для сравнения тэгов используются исключительно виртуальные адреса (физические адреса появляются только на выходе MMU). По-английски такой кэш называется Virtually Indexed, Virtually Tagged cache (VIVT).

~~Паразиты в шасси~~ Омонимы в VIVT-кэше

В чем же, собственно, проблема? Самое время рассмотреть еще один пример. Возьмем все те же два процесса А и Б:

1. Предположим, что сначала выполняется Процесс А. В процессе выполнения в кэш-память одна за другой загружаются строки с командами или данными для этого процесса (Строки 0-4).
2. Операционная система останавливает Процесс А и передает управление Процессу Б.
3. По идее, в этот момент процессор должен загрузить в кэш первую строку из первой страницы Процесса Б и начать выполнять оттуда команды. На самом деле, процессор подает на вход кэша виртуальный адрес 0x0, после чего кэш отвечает, что ничего загружать не надо, ибо нужная строка уже закэширована.
4. Процесс Б начинает весело выполнять код Процесса А.

Подождите, скажете вы, мы же только что решили эту проблему, добавив ASID в TLB? А кэш-то у нас стоит до MMU — когда промаха кэша нет, то мы в MMU и не смотрим, и какой там ASID — знать не знаем.  
  
  
  
Это первая проблема c VIVT-кэшем — так называемые омонимы (homonyms), когда один и тот же виртуальный адрес может отображаться на разные физические адреса (в нашем случае виртуальный адрес 0x00000000 отображается на физический адрес 0x40000500 для Процесса А и 0x40000000 для Процесса Б).  
  
Вариантов решения проблемы омонимов множество:

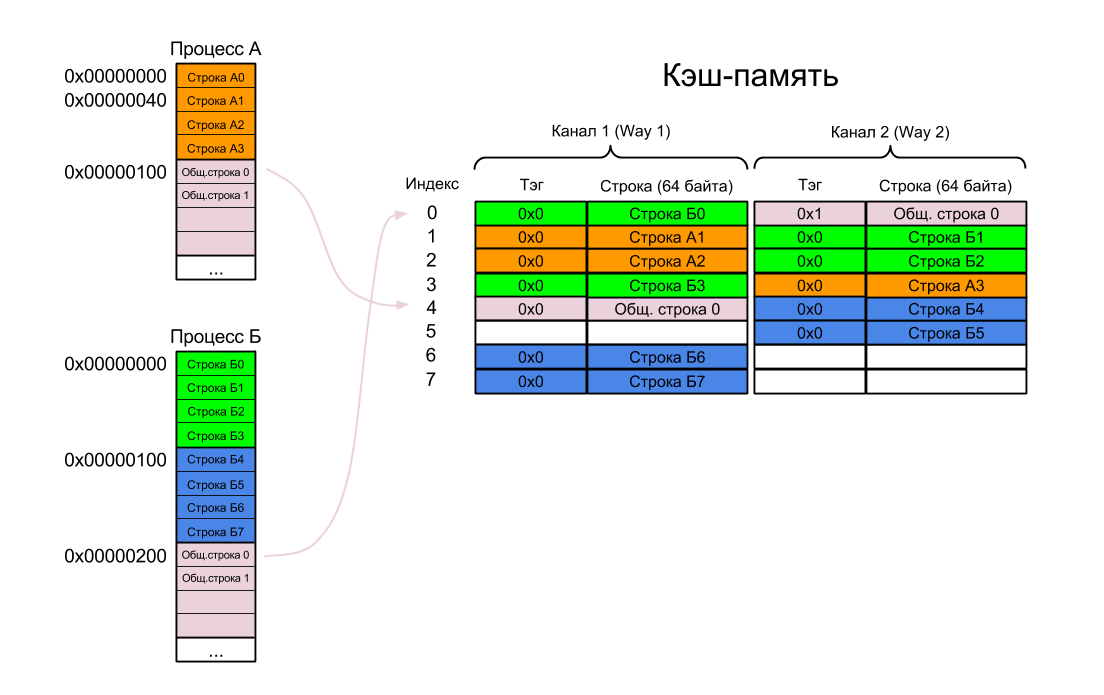
1. «Флашить» кэш (cache flush, т.е. записывать измененное содержимое кэша обратно в память) и инвалидировать кэш (то есть отмечать строки как недействительные) при переключении контекста. Если у нас раздельный кэш команд и данных, то кэш команд достаточно просто инвалидировать (то есть отметить все строки как пустые), потому что кэш команд доступен процессору только для чтения и флашить его нет смысла
2. Добавить ASID в тэг кэша
3. Обращаться в MMU каждый раз, когда мы обращаемся в кэш, а не только тогда, когда случился промах — тогда можем воспользоваться уже имеющейся логикой сравнения PID с ASID. Прощайте, энергосбережение и быстродействие!

Какой вариант вам нравится? Я бы, наверное, думал насчет второго. В ARM9 и некоторых других процессорах с VIVT-кэшами реализован третий.

Синонимы в VIVT-кэше

Если бы омонимы в кэше были бы единственной проблемой, то разработчики процессоров были бы самыми счастливыми людьми на свете. Существует и вторая проблема — синонимы (synonyms, aliases), когда несколько виртуальных адресов отображаются на один и тот же физический адрес.  
Вернемся к нашим ~~баранам~~ процессам А и Б. Предположим, что проблему омонимов мы решили каким-нибудь приличным способом, потому что каждый раз флашить кэш — это реально очень дорого!  
Итак:

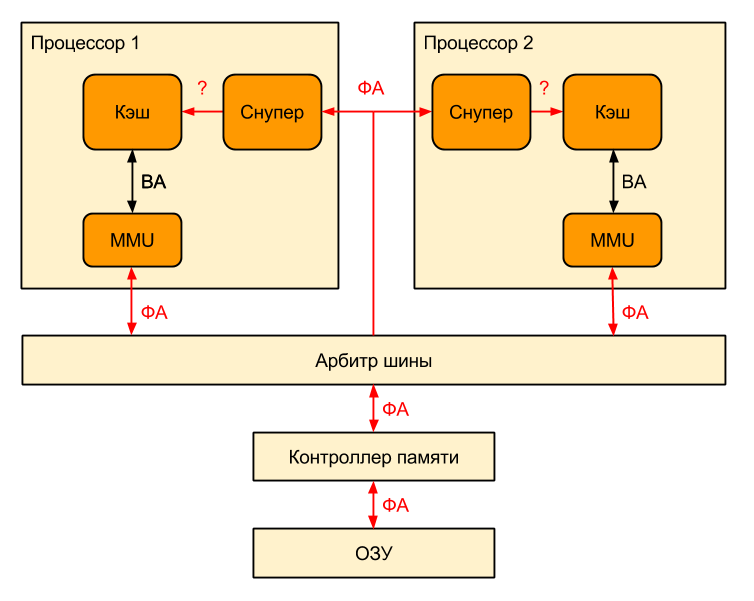
1. Сначала выполняется Процесс А — в кэш одна за другой загружаются Строки А0 — А3 и Общая строка 0 (помните, что у процессов А и Б одна страница общая)
2. Потом операционная система, не флаша кэш, переключает контекст
3. Начинает выполняться Процесс Б. В кэш загружаются строки Б0 — Б7.
4. Наконец, Процесс Б обращается к Общей строке 0. Эта строка уже загружена в кэш Процессом А, но процессор про это не знает, так как она фигурирует в кэше под другим виртуальным адресом (напомню, что в VIVT-кэше физических адресов нет)
5. Возникает промах кэша. Виртуальный адрес 0x00000200 транслируется в физический адрес 0x40000200, и Общая строка 0 повторно загружается в кэш. Ее расположение определяется виртуальным адресом — а адресу 0x00000200 соответствует Индекс 0 (биты 8-6) и Тэг 0x1 (биты 31-9).
6. Поскольку оба канала в наборе с индексом 0 уже заняты, приходится выкинуть одну из уже загруженных строк (А0 или Б0). Используя алгоритм LRU (Least Recently Used), кэш выкидывает Строку А0 и на ее место добавляет Общую строку 0.

  
  
В результате один и тот же кусок физической памяти находится в кэше в двух разных местах. Теперь если оба процесса изменят свою копию, а потом захотят сохранить ее в память, то один из них ждет сюрприз.  
Очевидно, что эта проблема актуальна только для кэша данных либо для объединенного кэша (повторюсь, кэш команд доступен только для чтения), но решить ее гораздо сложнее, чем проблему омонимов:

1. Самый простой способ, как мы уже выяснили — флашить кэш при каждом переключении контекста (кстати, для борьбы с синонимами, в отличие от омонимов, инвалидировать кэш не нужно). Однако, во-первых, это дорого, а во-вторых, это не поможет, если процесс захочет иметь в своем адресном пространстве несколько копий одной и той же физической страницы
2. Можно выявлять синонимы аппаратно:  
   * Либо при каждом промахе пробегать по всему кэшу, выполняя трансляцию адреса для каждого тэга и сравнивая полученные физические адреса с тем, который получен при трансляции адреса, вызвавшего промах (и позавидуют живые мертвым!)
   * Либо добавить в процессор новый блок, который будет выполнять обратную трансляцию — преобразование физического адреса в виртуальный. После этого при каждом промахе выполнять трансляцию вызвавшего его адреса, после чего при помощи обратной трансляции преобразовывать этот физический адрес в виртуальный и сравнивать его со всеми тэгами в кэше. Ей-богу, лучше бы вам просто флашить кэш!
3. Третий способ — избегать синонимов программно. Например, не использовать общие страницы. Или снова начать линковать программы в общее адресное пространство.

Когерентность и VIVT-кэш

Но и это еще не все! Мы живем в эпоху многоядерных процессоров с собственными L1-кэшами, которые так и норовят стать некогерентными. Для борьбы с некогерентностью кэшей были придуманы [хитрые протоколы](http://habrahabr.ru/post/183834/). Однако вот незадача: протоколы следят за внешней шиной, а адреса на ней какие? Физические! А в кэше используются виртуальные.



Как тогда узнать, какую строку кэша надо срочно записать в память, потому что ее ждет соседний процессор? ~~А никак~~ Пожалуй, это тема для отдельной статьи.

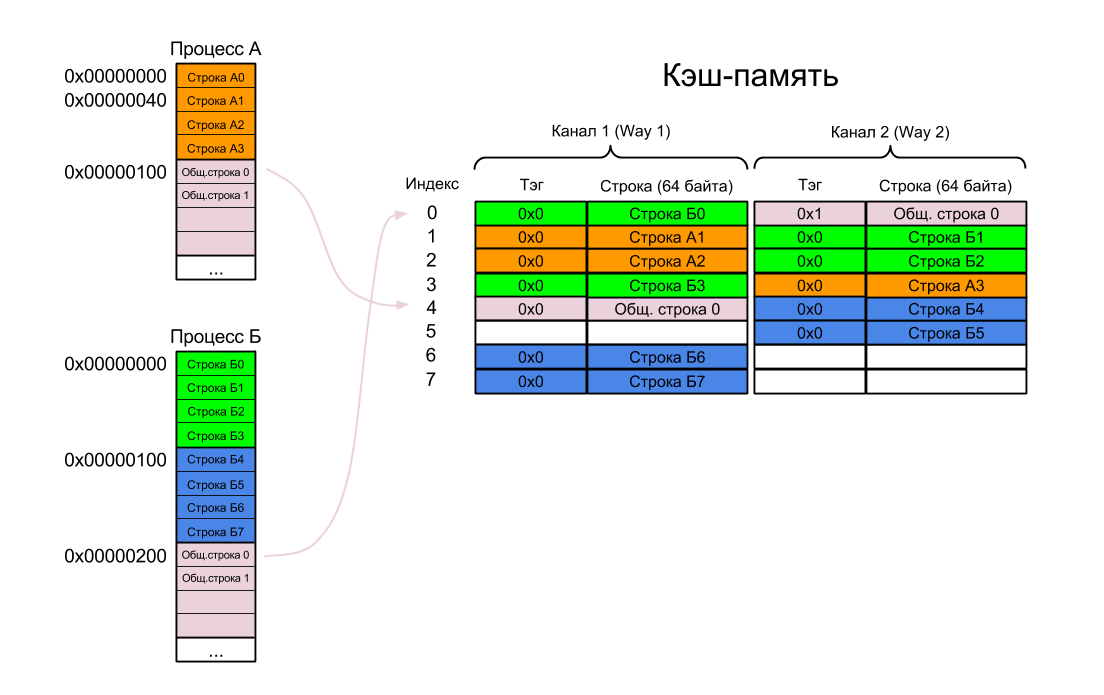
Достоинства VIVT-кэша

У VIVT-кэша есть единственный существенный плюс: если физическая страница выгружается из памяти, то соответствующие ей строки кэша нужно флашить, но не нужно инвалидировать. Даже если эта физическая страница через какое-то время будет загружена в другое место, на содержимом кэша это не отразится, потому что при доступе к VIVT-кэшу физические адреса не используются и процессору все равно, изменились они или нет.  
  
Лет дцать назад плюсов было больше. VIVT-кэши активно применялись в те времена, когда использовалось внешнее MMU в виде отдельной микросхемы. Доступ к такому MMU занимал гораздо больше времени, чем к MMU, расположенному на том же кристалле, что и процессор. Но поскольку обращение к MMU требовалось только в случае промаха кэша, то это позволяло обеспечивать приемлемую производительность системы.

Что делать?

Итак, у VIVT-кэшей слишком много недостатков и мало преимуществ (кстати, если я забыл какие-то еще грехи — пишите в комменты). Умные люди подумали и решили: перенесем-ка мы кэш-память за MMU. Пусть и индексы, и тэги получаются из физических, а не из виртуальных адресов. И перенесли. А сработало ли это — узнаете в следующей части.

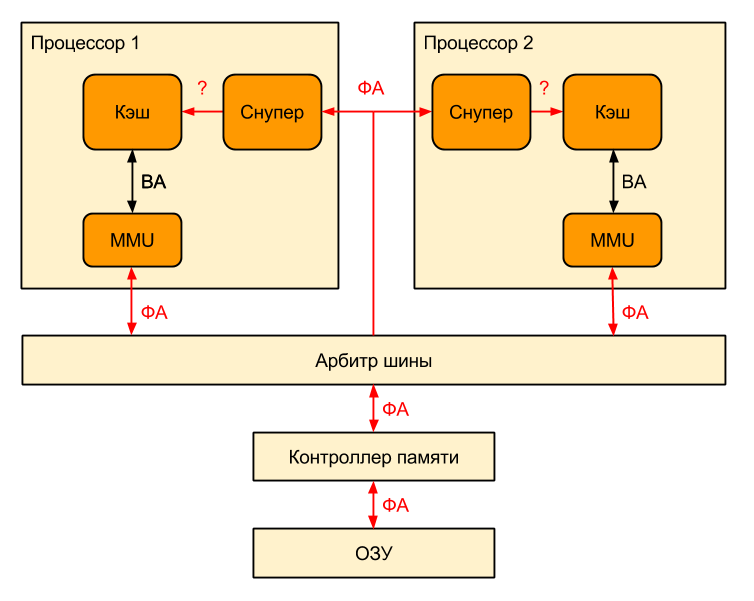
* [VIVT](http://habrahabr.ru/search/?q=%5BVIVT%5D&target_type=posts)
* , [MMU](http://habrahabr.ru/search/?q=%5BMMU%5D&target_type=posts)
* , [TLB](http://habrahabr.ru/search/?q=%5BTLB%5D&target_type=posts)
* , [кэш-память](http://habrahabr.ru/search/?q=%5B%D0%BA%D1%8D%D1%88-%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%5D&target_type=posts)
* , [кэш](http://habrahabr.ru/search/?q=%5B%D0%BA%D1%8D%D1%88%5D&target_type=posts)

  
  
В результате один и тот же кусок физической памяти находится в кэше в двух разных местах. Теперь если оба процесса изменят свою копию, а потом захотят сохранить ее в память, то один из них ждет сюрприз.  
Очевидно, что эта проблема актуальна только для кэша данных либо для объединенного кэша (повторюсь, кэш команд доступен только для чтения), но решить ее гораздо сложнее, чем проблему омонимов:

1. Самый простой способ, как мы уже выяснили — флашить кэш при каждом переключении контекста (кстати, для борьбы с синонимами, в отличие от омонимов, инвалидировать кэш не нужно). Однако, во-первых, это дорого, а во-вторых, это не поможет, если процесс захочет иметь в своем адресном пространстве несколько копий одной и той же физической страницы
2. Можно выявлять синонимы аппаратно:  
   * Либо при каждом промахе пробегать по всему кэшу, выполняя трансляцию адреса для каждого тэга и сравнивая полученные физические адреса с тем, который получен при трансляции адреса, вызвавшего промах (и позавидуют живые мертвым!)
   * Либо добавить в процессор новый блок, который будет выполнять обратную трансляцию — преобразование физического адреса в виртуальный. После этого при каждом промахе выполнять трансляцию вызвавшего его адреса, после чего при помощи обратной трансляции преобразовывать этот физический адрес в виртуальный и сравнивать его со всеми тэгами в кэше. Ей-богу, лучше бы вам просто флашить кэш!
3. Третий способ — избегать синонимов программно. Например, не использовать общие страницы. Или снова начать линковать программы в общее адресное пространство.

Когерентность и VIVT-кэш

Но и это еще не все! Мы живем в эпоху многоядерных процессоров с собственными L1-кэшами, которые так и норовят стать некогерентными. Для борьбы с некогерентностью кэшей были придуманы [хитрые протоколы](http://habrahabr.ru/post/183834/). Однако вот незадача: протоколы следят за внешней шиной, а адреса на ней какие? Физические! А в кэше используются виртуальные.



Как тогда узнать, какую строку кэша надо срочно записать в память, потому что ее ждет соседний процессор? ~~А никак~~ Пожалуй, это тема для отдельной статьи.

Достоинства VIVT-кэша

У VIVT-кэша есть единственный существенный плюс: если физическая страница выгружается из памяти, то соответствующие ей строки кэша нужно флашить, но не нужно инвалидировать. Даже если эта физическая страница через какое-то время будет загружена в другое место, на содержимом кэша это не отразится, потому что при доступе к VIVT-кэшу физические адреса не используются и процессору все равно, изменились они или нет.  
  
Лет дцать назад плюсов было больше. VIVT-кэши активно применялись в те времена, когда использовалось внешнее MMU в виде отдельной микросхемы. Доступ к такому MMU занимал гораздо больше времени, чем к MMU, расположенному на том же кристалле, что и процессор. Но поскольку обращение к MMU требовалось только в случае промаха кэша, то это позволяло обеспечивать приемлемую производительность системы.

Что делать?

Итак, у VIVT-кэшей слишком много недостатков и мало преимуществ (кстати, если я забыл какие-то еще грехи — пишите в комменты). Умные люди подумали и решили: перенесем-ка мы кэш-память за MMU. Пусть и индексы, и тэги получаются из физических, а не из виртуальных адресов. И перенесли. А сработало ли это — узнаете в следующей части.

* [VIVT](http://habrahabr.ru/search/?q=%5BVIVT%5D&target_type=posts)
* , [MMU](http://habrahabr.ru/search/?q=%5BMMU%5D&target_type=posts)
* , [TLB](http://habrahabr.ru/search/?q=%5BTLB%5D&target_type=posts)
* , [кэш-память](http://habrahabr.ru/search/?q=%5B%D0%BA%D1%8D%D1%88-%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%5D&target_type=posts)
* , [кэш](http://habrahabr.ru/search/?q=%5B%D0%BA%D1%8D%D1%88%5D&target_type=posts)