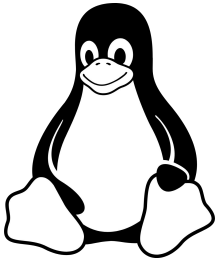
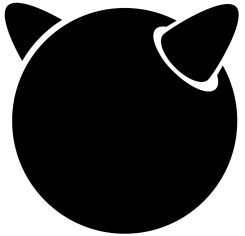
Operating systems



macOS



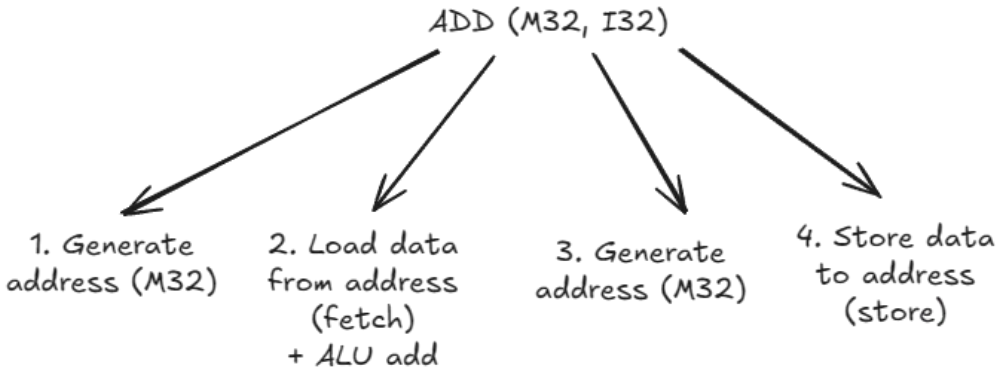
CPU, кэши и память

Компиляция кода в машинный код

- Программы, написанные на языках высокого уровня, компилируются или интерпретируются в машинный код
- Машинный код – набор низкоуровневых инструкций, которые процессор может исполнить
- Машинный код состоит из последовательности нулей и единиц. Каждая инструкция имеет свой битовый паттерн, так процессор знает, какую инструкцию выполнить
- ISA (instruction set architecture) — набор инструкций конкретного компьютера, их семантика и способ кодирования в памяти. (Примеры ISA: x86, ARM, MIPS, RISC-V)
- Компьютеры с одинаковой ISA могут быть по-разному устроены на уровне микроархитектуры — от этого может зависеть скорость или эффективность исполнения программ

Микрооперации (µOPs)

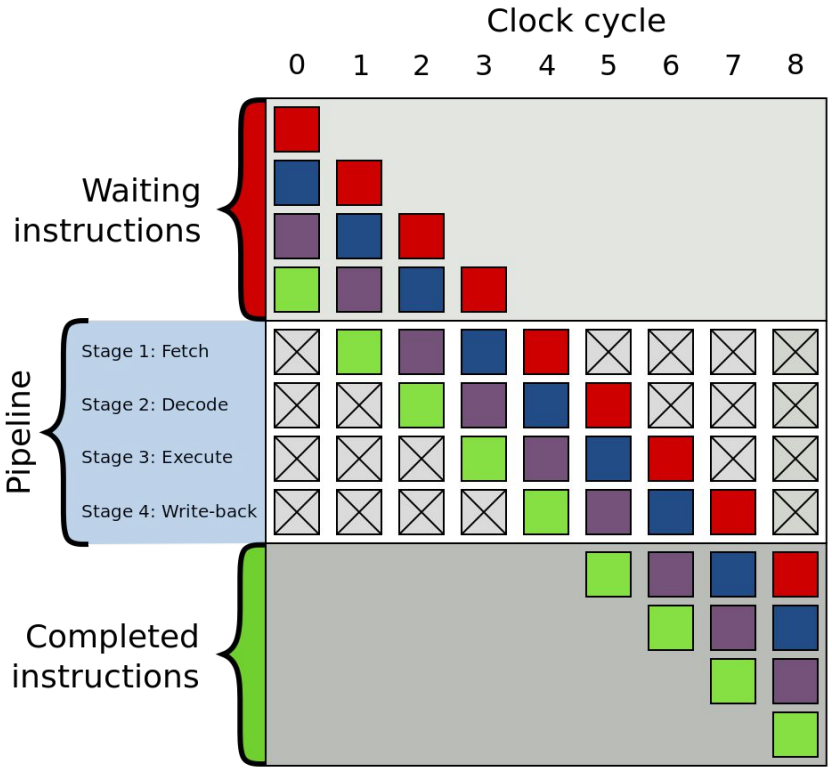
- Каждая инструкция машинного кода разбивается процессором на более мелкие операции – микрооперациями (µops)
- Это разбиение позволяет процессору оптимизировать выполнение, распределяя микрооперации на выполнение по разным его компонентам



Конвейер

- Чтобы ускорить обработку, процессор использует конвейер (pipeline), чтобы одновременно обрабатывать несколько инструкций, разделяя их выполнение на этапы:
  - Fetch (загрузка): процессор считывает инструкцию из памяти
  - Decode (декодирование): инструкция анализируется, чтобы понять, какие действия нужно выполнить, и преобразуется в микрооперации
  - Execute (выполнение): микрооперации отправляются на соответствующие исполнительные блоки (например, ALU или AGU)
  - Memory (доступ к памяти): если инструкция требует чтения или записи данных в оперативную память, это происходит на этой стадии
  - Write-back (запись результата): результат выполнения сохраняется в регистр или память

Конвейер


Clock cycle
0
1
2
3
4
5
6
7
8
Waiting instructions
Pipeline
Stage 1: Fetch
Stage 2: Decode
Stage 3: Execute
Stage 4: Write-back
Completed instructions

Порты или блоки исполнения

- Микрооперации выполняются на специализированных блоках, подключенных к портам, которые были распределены процессором
- **ALU** (Arithmetic Logic Unit, арифметико-логическое устройство) выполняет некоторый набор арифметических и логических операций
- **AGU** (Address Generation Unit, блок генерации адресов) вычисляет адреса для чтения или записи данных в память
- Каждый порт может обрабатывать определенные типы микроопераций, и суперскалярный процессор распределяет задачи по портам, чтобы одновременно выполнять как можно больше операций

Суперскалярность и выполнение вне порядка

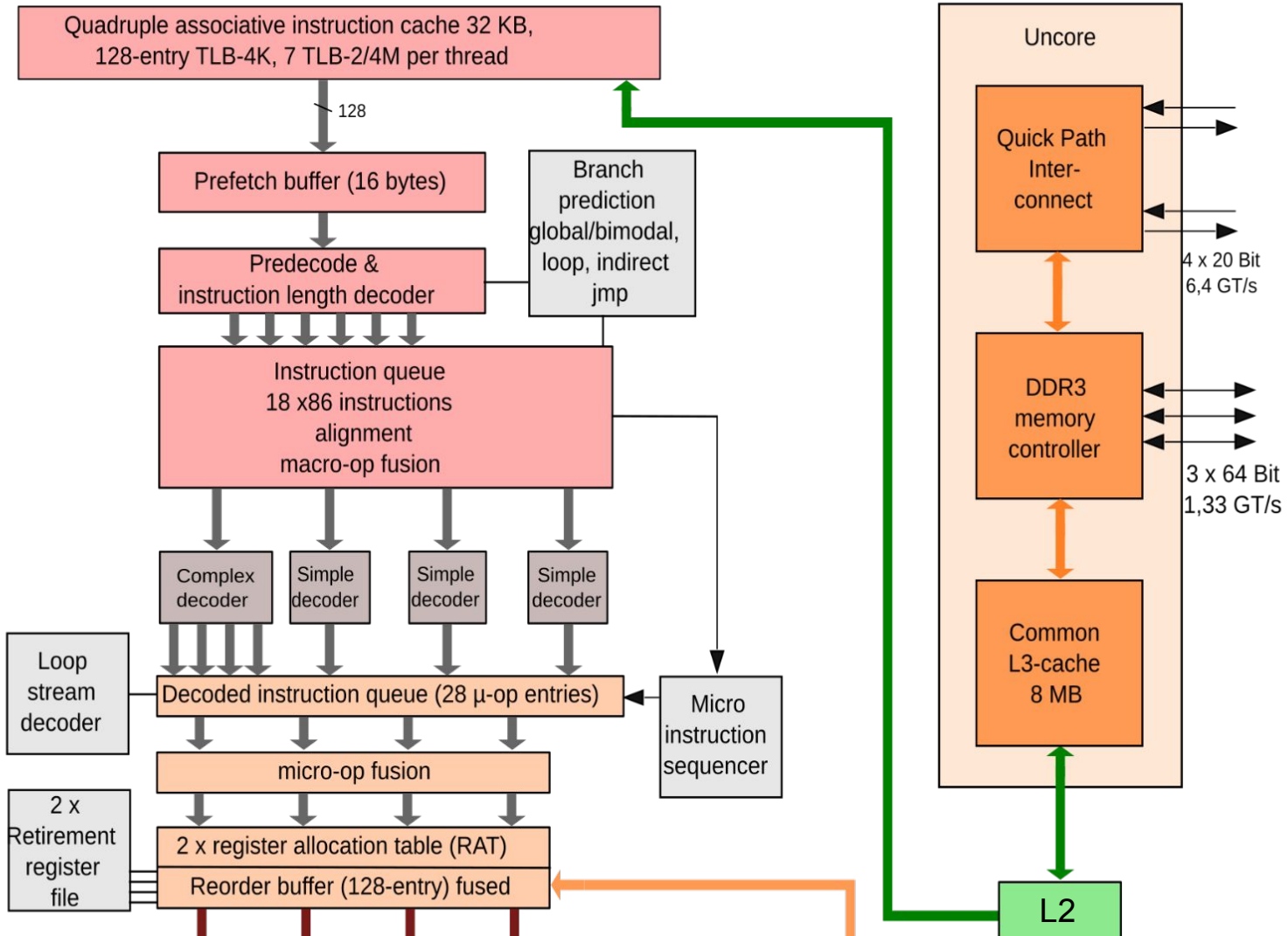
- Современные процессоры являются суперскалярными, то есть они могут обрабатывать несколько инструкций разными конвейерами в параллеле в один момент времени (instruction parallelism)
- Out-of-order выполнение применяется процессором, чтобы повысить пропускную способность вычислений. Если одна инструкция ожидает данные (например, из памяти), то процессор может начать выполнение следующей инструкции, если она готова на исполнение
- **Scheduler** — компонент процессора, который анализирует зависимости между микрооперациями и распределяет их по доступным исполнительным блокам, чтобы минимизировать простои.

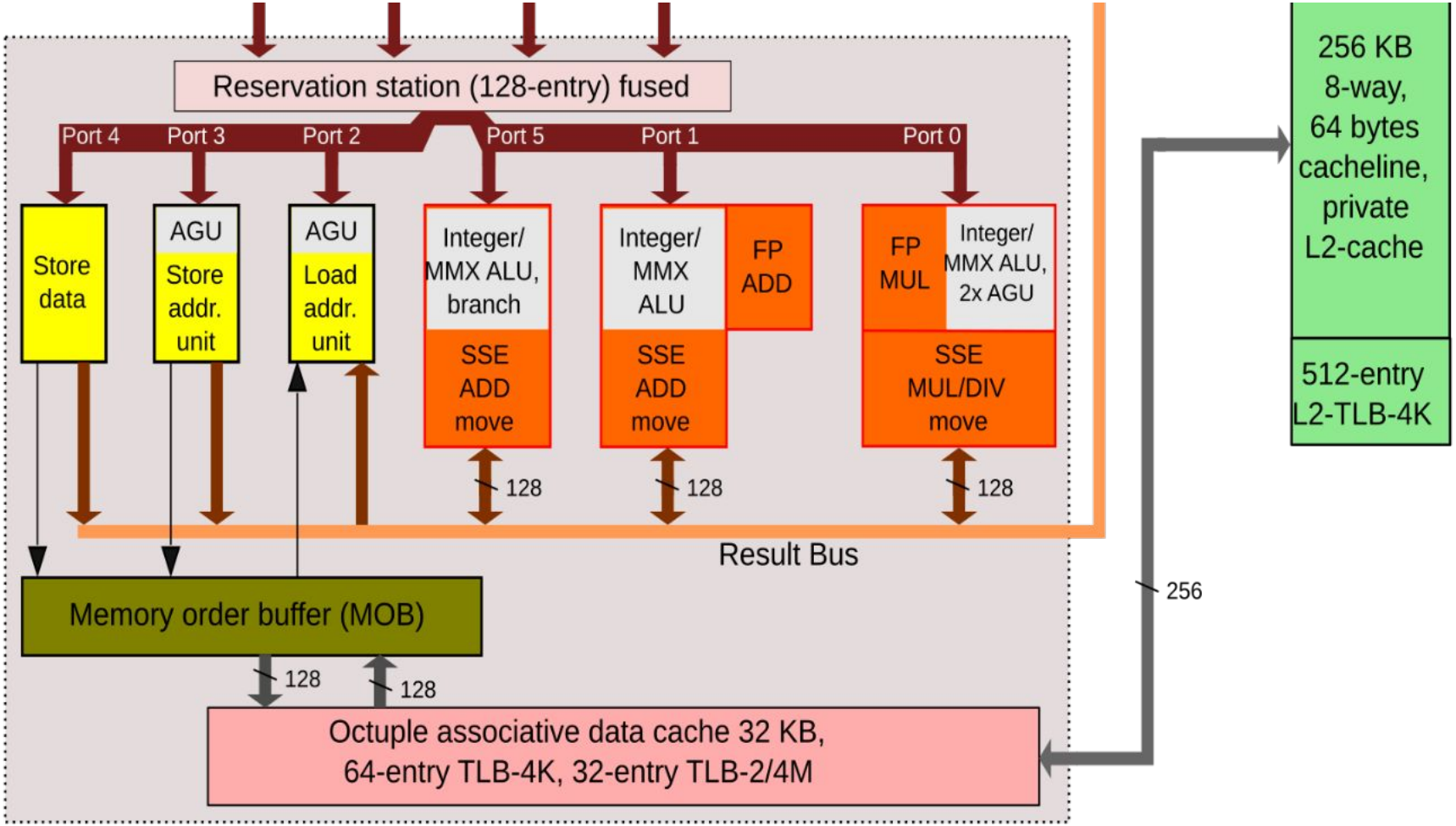
Переименование регистров

- Процессор использует **регистры** — небольшие ячейки сверхбыстрой памяти внутри процессора для хранения промежуточных данных
- Однако инструкции могут пытаться использовать один и тот же регистр, что вызывает конфликты при параллельном их исполнении
- **Register renaming** решает эту проблему: процессор создает временные “копии” регистров, позволяя нескольким инструкциям работать с данными параллельно без ожидания. Это увеличивает параллелизм и производительность
- **Регистровый файл** используется в процессоре, чтобы сопоставить физический регистр с его именем во время исполнения машинного кода

Предсказание ветвлений (branch prediction)

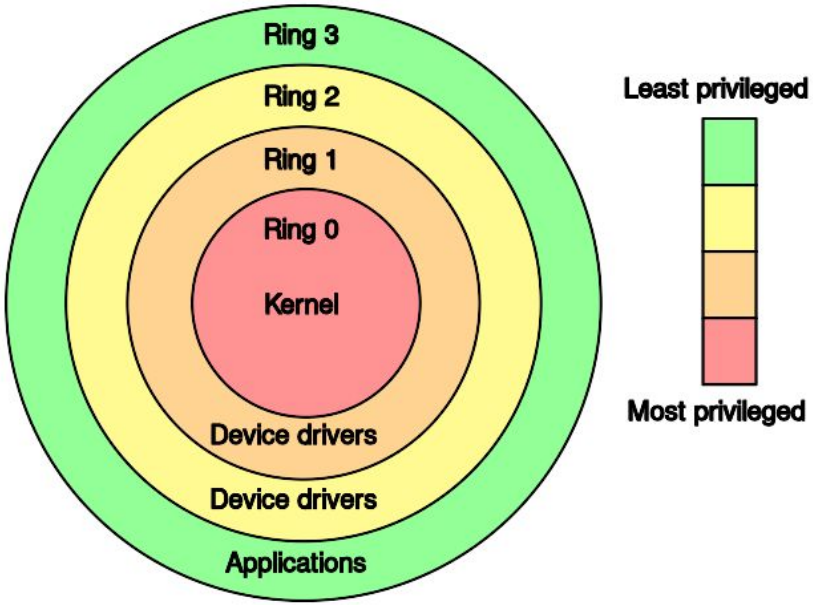
- Программы часто содержат ветки ветвления кода и чтобы не терять время на ожидание результата условия, процессор использует предсказание ветвлений.
- Он пытается “угадать”, какой путь программы будет выполнен, и начинает загружать и выполнять соответствующие инструкции заранее
- Если предсказание оказывается неверным, процессор откатывает изменения и исполняет инструкции той ветки, что была правильной
- Если удалось угадать, то правильные предсказания значительно ускоряют работу


Quadruple associative instruction cache 32 KB,
128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch buffer (16 bytes)
Branch prediction global/bimodal, loop, indirect jmp
Predecode & instruction length decoder
Instruction queue
18 x86 instructions
alignment
macro-op fusion
Complex decoder
Simple decoder
Simple decoder
Simple decoder
Loop stream decoder
Decoded instruction queue (28 µ-op entries)
Micro instruction sequencer
micro-op fusion
2 x Retirement register file
2 x register allocation table (RAT)
Reorder buffer (128-entry) fused
Uncore
Quick Path Inter-connect
4 x 20 Bit
6,4 GT/s
DDR3 memory controller
3 x 64 Bit
1,33 GT/s
Common L3-cache 8 MB
L2


Reservation station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store data
AGU
Store addr. unit
AGU
Load addr. unit
Integer/ MMX ALU, branch
SSE ADD move
Integer/ MMX ALU
FP ADD
SSE ADD move
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV move
128
128
128
Result Bus
Memory order buffer (MOB)
128
128
Octuple associative data cache 32 KB, 64-entry TLB-4K, 32-entry TLB-2/4M
256
256 KB 8-way, 64 bytes cacheline, private L2-cache
512-entry L2-TLB-4K

Кольца защиты

- Аппаратная реализация разделения системного (kernel mode) и пользовательского (user mode) уровней привилегий
- Обычно, на процессорных архитектурах реализуется 4 кольца (на Intel есть еще отрицательные кольца)
- На практике используется всего два: ring 0 и ring 3

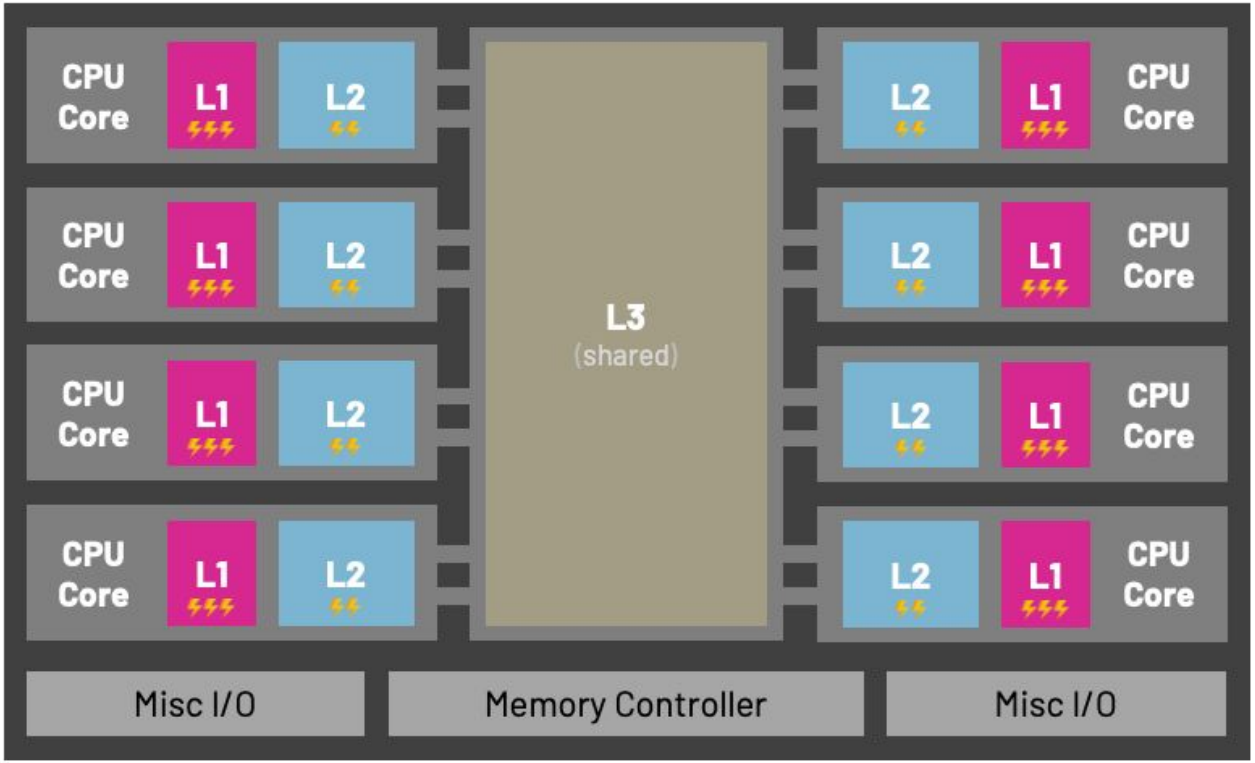


Привилегии колец

- **Ring 0**:
  - Доступ к железу (port mapped IO, memory mapped IO)
  - Служебные регистры (GDTR, IDTR, контрольные регистры)
  - Служебные инструкции (`lgdt`, `lidt`, `cli`/`sti`, ...)
- **Ring 3**:
  - Доступ к инструкциям/регистрам общего назначения
  - Полностью самостоятельно можно только пользоваться доступной памятью и что-то считать
  - за всем остальным придется звать ядро
- Актуальный уровень привилегий CPL (current privilege level) хранится в младших двух битах регистра `cs` (на Intel x86)

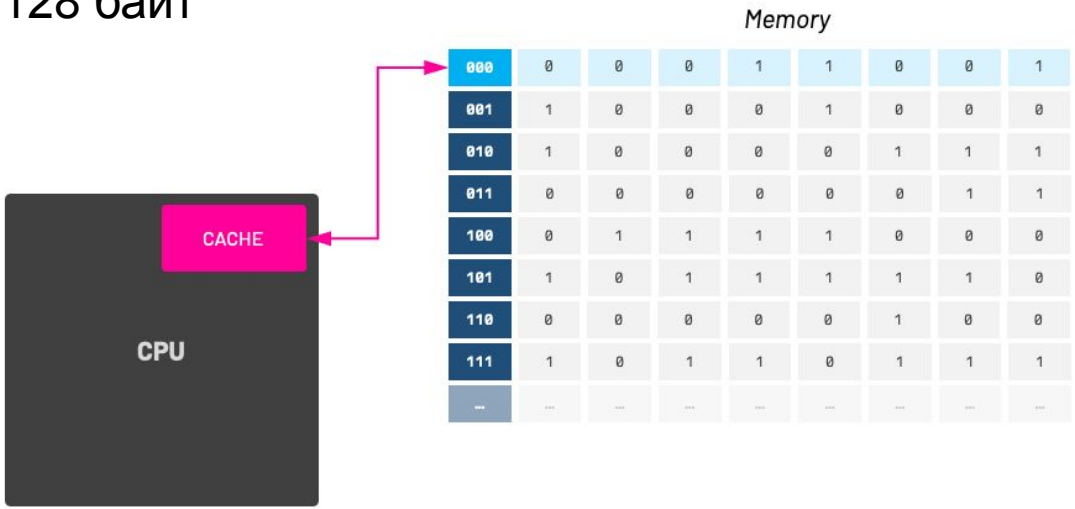
Кэши процессора

Кэши



Линии кэша

- Кэш может загружать N блоков памяти. Блоки кэша могут быть разного размера, и количество блоков в кэше обычно является степенью числа 2
- Блоки памяти кэша называют **линиями кэша**. Обычные размеры строк кэша составляют 32, 64 и 128 байт
- Линия кэша сохраняется в кэше, мы также сохраняем ее местоположение в памяти



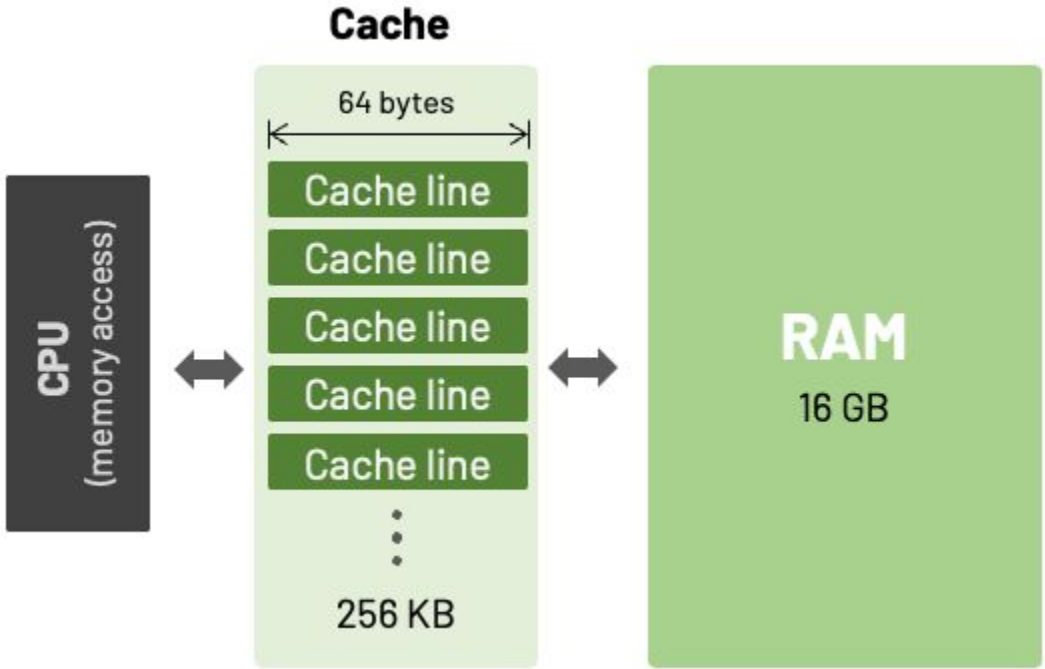
Попадания и промахи

- Когда мы успешно сохраняем и извлекаем строку кэша, мы называем это **попаданием в кэш (cache hit)**
- В противном случае – это **промах кэша (cache miss)**
- Как программисты, учитывающие кэш, мы должны стараться увеличить количество попаданий в кэш и избежать промахов кэша, чтобы доступ к памяти был всегда быстрый

Кэш линии

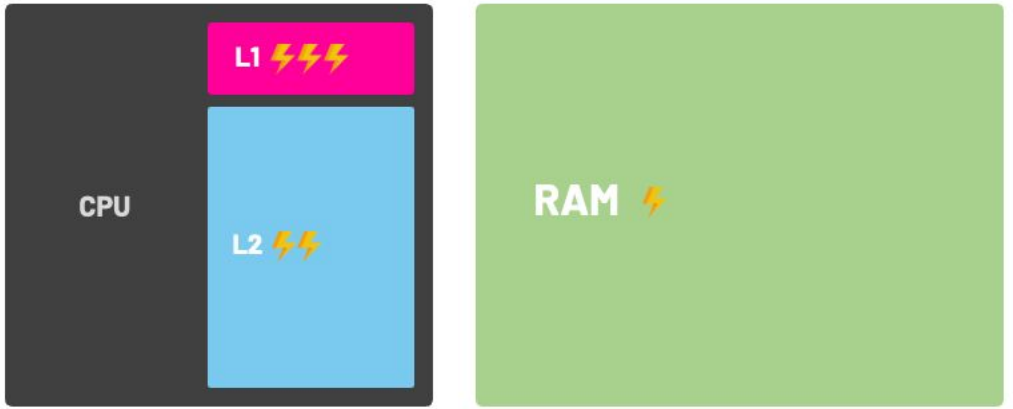
- Следует помнить, что различные микроархитектуры процессоров определяют разные размеры линий кэша
- Размер линии кэша (или размер блока кэша) является единицей данных, передаваемых в основную память и из нее
- Он значительно влияет на эффективность взаимодействия ЦПУ с подсистемой памяти. При использовании более крупных строк кэша можно более эффективно использовать *пространственную локальность* (близость) данных

Кэш линии


Cache
64 bytes
Cache line
Cache line
Cache line
Cache line
Cache line
256 KB
CPU
(memory access)
RAM
16 GB

Многоуровневые кэши

- Доступ к кэшу L1 может занять ~3 цикла ЦПУ
- Доступ к кэшу L2 — ~10 циклов
- Доступ к обычной оперативной памяти — ~250 циклов



Кэши разного типа

- Большинство процессоров имеют отдельный кэш для инструкций (**i-cache**) и другой кэш для данных (**d-cache**)
- Кэш инструкций хранит инструкции на языке ассемблера/коды операций, а кэш данных — данные, доступные из основной памяти
- Одно из основных отличий между ними заключается в том, что кэш данных должен быть способен выполнять как операции **чтения**, так и **записи**, в то время как кэш инструкций должен обеспечивать только операции чтения
- Существуют также некоторые ожидания в отношении того, как мы предполагаем, что данные и инструкции будут доступны (последовательно или случайным образом)

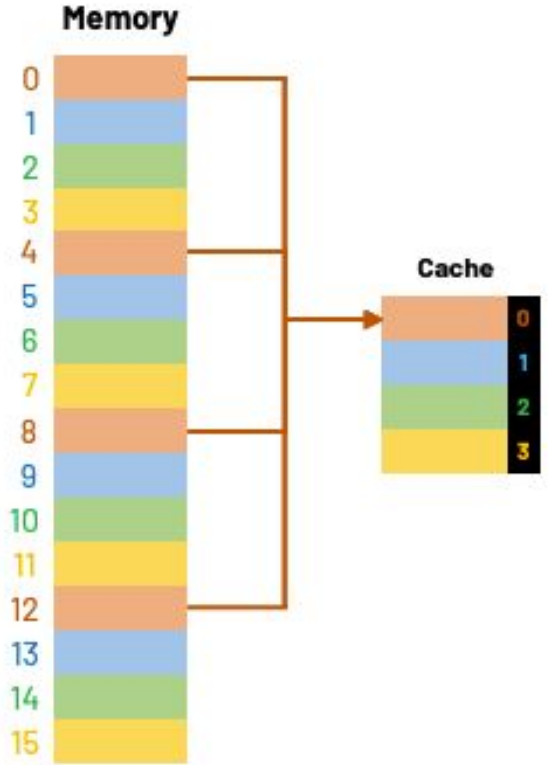
Как положить данные в кэш?

- Существует множество политик размещения кэша
  - кэш с прямым отображением
  - полностью ассоциативный кэш
  - наборный ассоциативный кэш
  - двусторонний ассоциативный кэш со смещением
  - псевдоассоциативный кэш

Кэш с прямым отображением

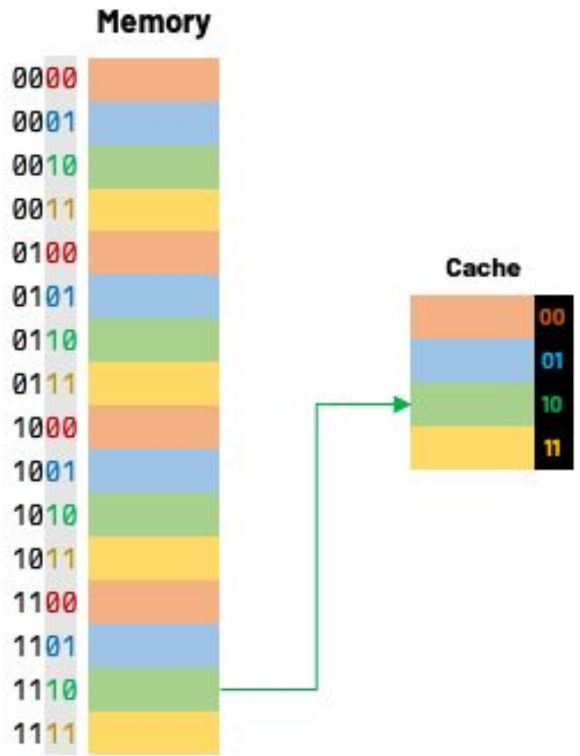
- Кэш с прямым отображением это самый простой подход, при котором каждый адрес памяти отображается точно на один блок кэша
- Кэш организован в несколько наборов с одной линией кэша на набор

Блоки памяти 0, 4, 8 и 12 => линия кэша 0
Блоки памяти 1, 5, 9 и 13 => линия кэша 1
Блоки памяти 2, 6, 10 и 14 => линия кэша 2
Блоки памяти 3, 7, 11 и 15 => линия кэша 3



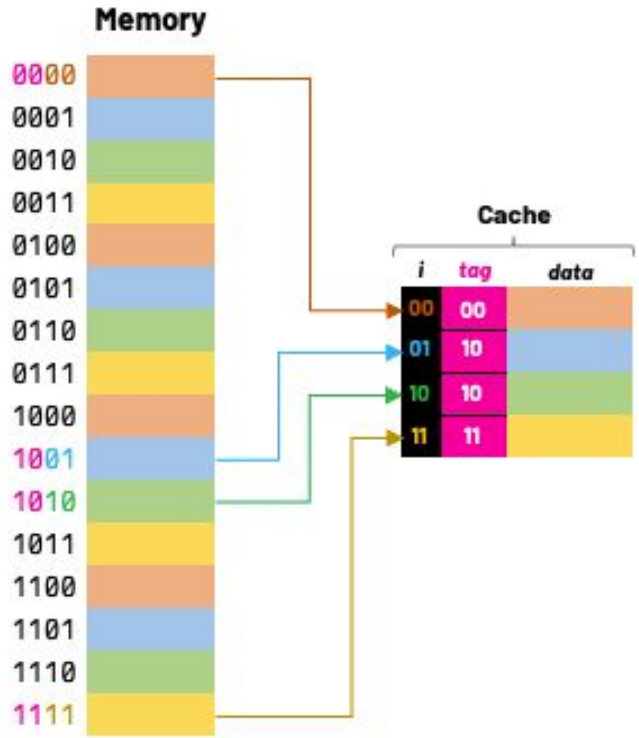
Кэш с прямым отображением

- Один из способов определить, в какой блок кэша должен попасть конкретный адрес памяти, — использовать оператор остатка от деления (mod)
- Если кэш содержит 2^k блоков, то данные по адресу памяти i попадут в блок кэша с индексом: $i \bmod 2^k$



Кэш с прямым отображением

- Как мы можем различить эти потенциальные адреса, которые все сопоставляются с одним и тем же индексом блока кэша?
- Один из подходов к решению этой проблемы заключается в добавлении **тегов** в кэш
- Эти теги предоставляют остальные биты адреса, позволяя нам различать различные ячейки памяти, которые сопоставляются с одним и тем же блоком кэша



Метаданные для кэша

- Кэш содержит не только необработанные копии данных в памяти, но и метаданные (дополнительные биты), которые помогают находить данные в кэше и проверять их действительность
- Последней частичкой информацией, которую мы также храним в кэше, является бит действительности данных
- В начале кэш пуст, и все биты действительности установлены в ноль. Когда данные загружаются в определенный блок кэша, соответствующий флаг бита действительности устанавливается в 1

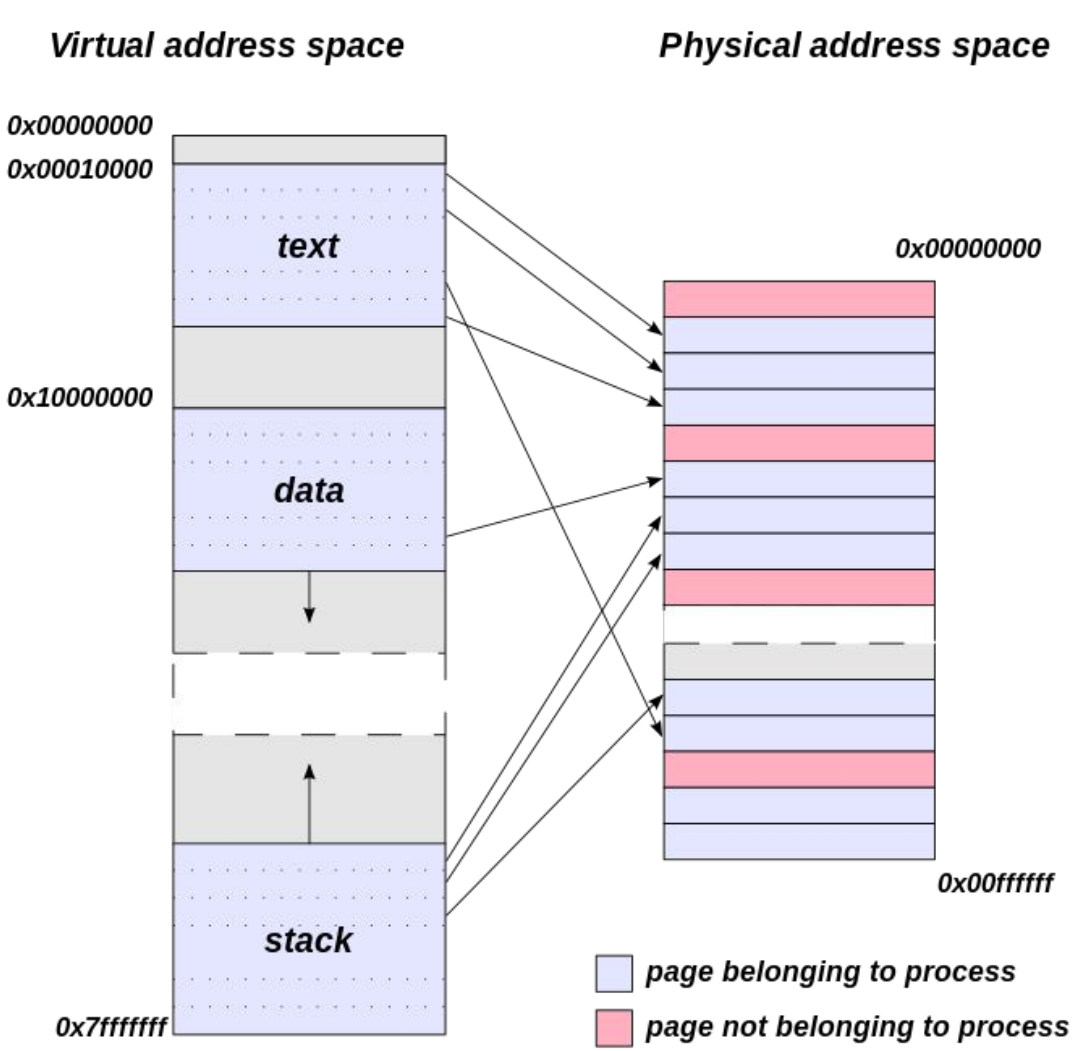
Виртуальная память

Виртуальная память

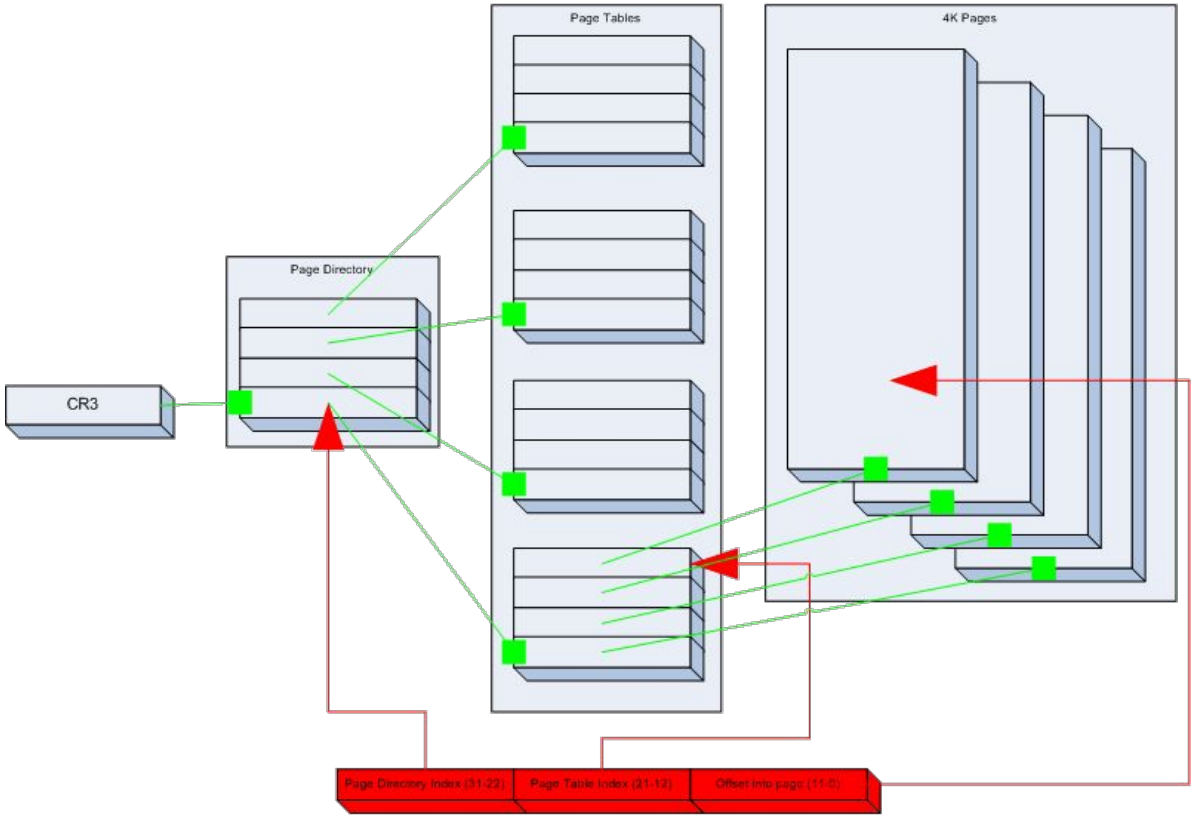
- **Виртуальная память** – это абстракция, позволяющая процессам использовать *виртуальные* адреса, которые ОС отображает на *физические* в RAM или диск
- Каждый процесс имеет свое **адресное пространство** (защита от доступа к чужой памяти и изоляция процессов друг от друга)
- Механизм overcommitment позволяет зарезервировать больше виртуальной памяти, чем физической
- Права доступа осуществляются механизмом RWX (как в ФС Unix)
- Поддержка ядра ОС для выгрузки неактивных страниц на диск (swap file)

Как работает виртуальная память?

- Виртуальные адреса транслируются в физические адреса через MMU (Memory Management Unit) и результат кешируется в TLB (translation lookaside buffer)
- Виртуальная память делится на блоки фикс. размера – **страницы** (memory pages, обычно 4KB), а блоки физ. памяти – **фреймы (frames)**
- Таблицы страниц (page tables) отображают виртуальные страницы на физические или диск (swap)
- Если программа трогает страницу, которой нет в адресном пространстве или не имеет прав на операцию, то происходит **page fault** (аппаратное прерывание
- Если страница памяти не в RAM происходит подкачка с диска


Virtual address space
Physical address space
0x00000000
0x00010000
text
0x10000000
data
stack
0x7fffffff
0x00000000
0x00ffffff
page belonging to process
page not belonging to process

Как устроена страничная вирт. память в x86

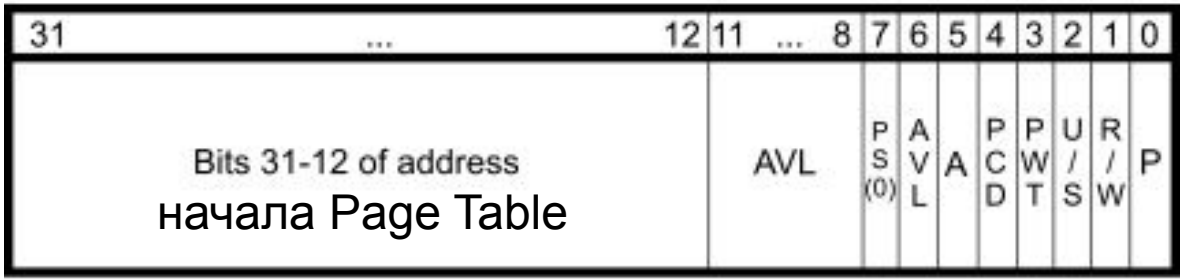


Как устроена страничная вирт. память в x86

- В регистре CR3 хранится физический адрес начала таблицы Page Directory
- В последних 10 битах виртуального адреса хранится индекс в таблицу Page Directory
- В следующих 10 битах вирт. адреса хранится индекс в таблицу Page Table, адрес который будет известен из записи Page Directory по индексу
- В первых 12 битах вирт. адреса хранится смещение физической страницы памяти, адрес которой будет известен из записи Page Table по индексу

Как устроена страничная вирт. память в x86

Page Directory Entry



P: Present	**D:** Dirty
R/W: Read/Write	**PS:** Page Size
U/S: User/Supervisor	**G:** Global
PWT: Write-Through	**AVL:** Available
PCD: Cache Disable	**PAT:** Page Attribute Table
A: Accessed	

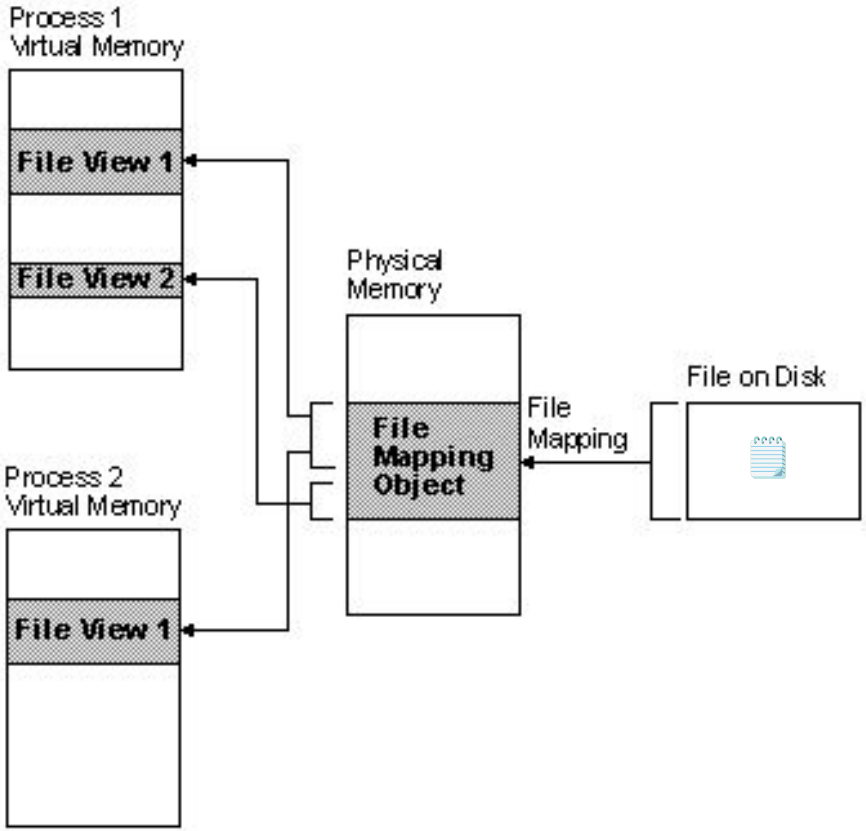
Как устроена страничная вирт. память в x86

Файловый маппинг

Отображение файла в память

- File mapping – техника ядра операционной системы, которая позволяет отобразить содержимое файла в виртуальное адресное пространство процесса, чтобы работать с ним как с обычной памятью
- Ядро выделяет область виртуальной памяти, связанную с файлом на диске
- Процесс может читать/записывать данные в эту область памяти, а ОС автоматически синхронизирует изменения с файлом
- Данные загружаются в физическую память только при первом обращении (page fault), то есть подгрузка содержимого файла будет происходить *лениво*

Отображение файла в память



Отображение файла в память

- Теперь не нужно читать/записывать файл через системные вызовы read/write!
- Работа с отображенным файлом выглядит, как обычная манипуляция с массивом по указателю
- Несколько процессов могут отображать один файл, используя общую память
- Не требует загрузки всего файла в RAM, его части будут подгружены по наставлению и выгружены из RAM и автоматически или по наставлению тоже!

Отображение файла в память

```
void *mmap(void *addr,     // Адрес по желанию
           size_t length,  // Сколько байт нужно отобразить
           int prot,       // Права доступа
           int flags,      // Флаги (MAP_SHARED, MAP_PRIVATE)
           int fd,         // Дескриптор файла
           off_t offset); // Смещение
```

CoW 🐄

- Copy-On-Write (COW) – техника оптимизации памяти, при которой несколько процессов/объектов изначально ссылаются на одну и ту же копию данных в памяти, а копирование происходит только при попытке модификации
- В Linux при создании дочернего процесса копируется только таблица страниц, а данные процесса – при записи в них
- Благодаря этому механизму гарантируется атомарность изменений, что другие процессы видят неизменные данные
- Копии данных не создаются, пока общие данные не изменяются

CoW 🐄

```
// Родительский процесс
int fd = open("file.txt", O_RDWR);
void* addr = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);

// Fork создает CoW-копию адресного пространства
// Оба процесса изначально ссылаются на общие страницы файла
pid_t pid = fork();

// При записи в дочернем процессе:
if (pid == 0) {
    *(char*)addr = 'X'; // CoW: создается частная копия страницы и файл на диске НЕ изменяется
}
```