OTUS

ОНЛАЙН-ОБРАЗОВАНИЕ

OTUS

Кэширование

Михаил Горшков
разработчик

ВКЛЮЧИТЬ ЗАПИСЬ!!!

Заполните опросник перед началом занятия: https://docs.google.com/forms/d/e/1FAIpQLSe_X0-ueYIVqPEnTOPcbVGkrteLmCjpDBFfsgiQj7JSvEBKvA/viewform (https://docs.google.com/forms/d/e/1FAIpQLSe_X0-ueYIVqPEnTOPcbVGkrteLmCjpDBFfsgiQj7JSvEBKvA/viewform)

Общая информация про кэширование

Вопрос к аудитории: приведите примеры кэшей

Кэш - от англ. *cache* (не "наличные деньги" - cash, а от фр. *cacher* — "прятать");

Что такое кеширование, зачем применяется, что дает.

Примерное объяснение - вы обращаетесь к более медленному ресурсу, а результат обращения храните на более быстром ресурсе для того, чтобы впоследствии воспользоваться этим результатом быстрее. Для пользователя этот более быстрый ресурс "спрятан", он его не замечает.

Размещение данных в специально отведенном месте для ускоренного доступа к ним в будущем.

Как работает кэш

- когда пользователь системы с кэшированием запрашивает данные, сначала проверяется кэш;
- если он содержит требуемые данные, то используются они без обращения к основному хранилищу данных;
- если данных в кэше нет, они запрашиваются в хранилище и добавляются в кэш;
- когда кэш заполняется, данные из него каким-то образом очищаются, чтобы освободить место для новых данных.

Вопросы к аудитории:

- Что такое 'cache hit'?
- Что такое 'cache miss' (промах)?
- Что такое 'hit rate'?
- Что такое 'miss rate'?
- Что такое 'hit time'?
- Что такое 'вымывание' или 'загрязнение' кэша (cache pollution)?
- Что такое 'cache coherence'?
- Что такое 'cache oblivious'?

Термины

- cache hit - объект найден в кэше;
- cache miss - объект не найден в кэше;
- hit rate - total number of hits / (total number of hits + total number of misses) * 100%; Если 100% - все нашли в кэше, если 0% - ничего не нашли, если 50% - нашли половину
- miss rate - total number of misses / (total number of hits + total number of misses) * 100%;
- hit time - время хита;
- cache pollution - вымывание/загрязнение кэша - быстрое удаление элементов из него ввиду постоянной записи новых данных, недостаточного размера кэша, неподходящей стратегии его реализации или иных причин, ввиду чего кэш дает почти всегда 100% miss rate;
- cache coherence - поддержание одинаковости разных локальных кэшей, кэширующих одно и то же - например, кэш процессора в многопроцессорной системе: в одном кэше изменились данные => надо обновить остальные;
- cache oblivious алгоритм - алгоритм, учитывающий и использующий особенности кэша;

Когда может использоваться кэширование?

- вы производите "дорогие" вычисления (например, сложные запросы к БД или ресурсоемкие вычисления) - тогда можете закэшировать результат;
- когда можно переиспользовать ранее вычисленное значение и это целесообразно;
- MRU кэш для пользовательских запросов (последних открытых имен файлов в редакторах).

Когда не нужно писать свой кэш?

- не нужно дублировать уже существующий кэш, например, файловый;
- бывают ситуации, когда кэшировать нецелесообразно, например, вычислить что-либо стоит столько же, сколько взять из кэша.

Вопрос к аудитории: можно ли кэшировать результат функции, использующей для своих вычислений текущее время?

Нельзя кэшировать!!

- результат вычисления функций с сайд-эффектами, функций, которые создают какие-либо объекты на каждом вызове;
- результат функций, предназначенных для генерирования недетерминированных данных, например time() или random();
- поговорим о кэшировании функций далее.

Стратегии и алгоритмы общецелевого кэширования (процессорный, дисковый, БД, веб)

Стратегии "выталкивания" элементов из кэша

Когда требуемых данных нет в кэше, это cache miss. Над принять решение: хранить ли данные, полученные из основного хранилища, в кэше, и, если да, то какие данные убрать из кэша, чтобы дать место новым? Это т.н. политика замены, replacement policy. Наиболее эффективная стратегия - это всегда убирать объект, который не будет нужен в будущем. В 1966 Belady разработал replacement policy, минимизирующую число промахов. Доказано, что этот алг-м оптимальный (см ссылку в конце). На практике этот оптимальный алгоритм невозможно применить, можно только сравнить эффективность разных алгоритмов кэширования в данных условиях.

- FIFO - первым добавлен, первым вытолкнется
- LIFO - последним добавлен, первым вытолкнется
- MRU - выталкивается последний используемый
- LRU - выталкивается наиболее давно используемый
- LFU - выталкивается наименее часто используемый
- LRU2 - выталкивается наименее используемый 2 раза
- 2Q - 2 очереди
- Time-based expiration - выталкиваются давно размещенные

FIFO

(First In First Out):

- объекты добавляются в кэш по мере доступа к ним, располагаются в очереди (буфере) и не меняют своего положения в буфере;
- когда кэш заполняется, объекты удаляются в порядке, в котором они были добавлены;
- время доступа к кэшу - константное независимо от размера кэша (можно реализовать через массив с прямым поиском).

Преимущества:

- простой и быстрый;

Недостаток:

- не очень "интеллектуальный"; нет функционала для хранения часто используемых объектов.

LIFO

(Last In First Out): похож на FIFO, но при заполнении кэша объекты выталкиваются в обратном порядке.

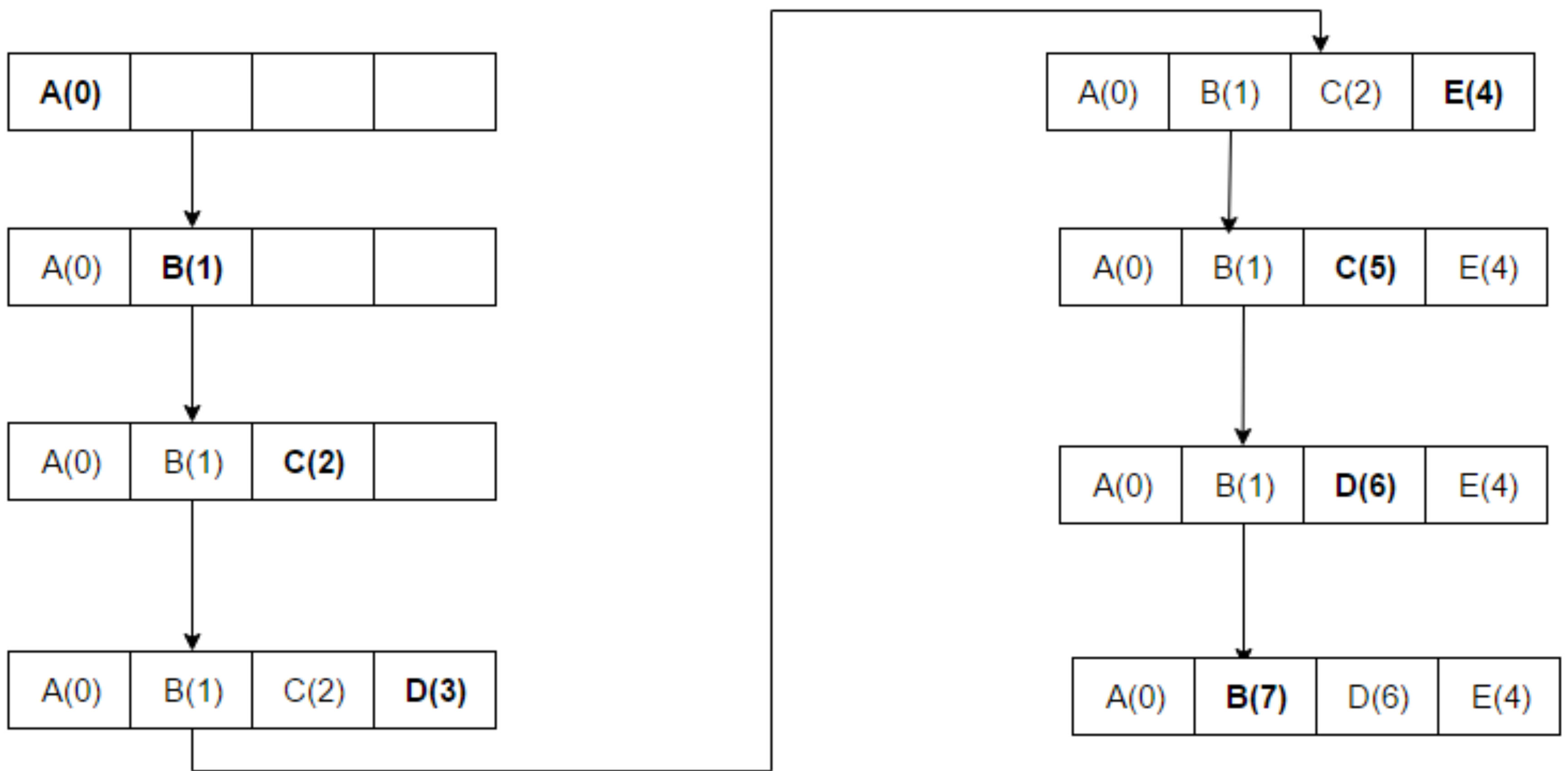
- для хранения используется стек;

MRU

Выталкивается последний использованный

- подходит для сканов - один раз прошли по объекту, больше не вернемся к нему, можно выталкивать;
- используется в ситуациях, когда чем старее объект, тем больше вероятность, что он будет затребован.

На диаграмме показана обработка последовательности A B C D E C D B



LRU

(Least Recently Used)

Наиболее часто используемый алгоритм кэширования.

- объекты добавляются в кэш по мере доступа к ним;
- когда кэш заполняется, выталкивается объект, который читался (использовался) максимально давно.
- обычно реализуется как связный список, так что объект кэша, который достается, перемещается в голову; объекты удаляются с конца списка;

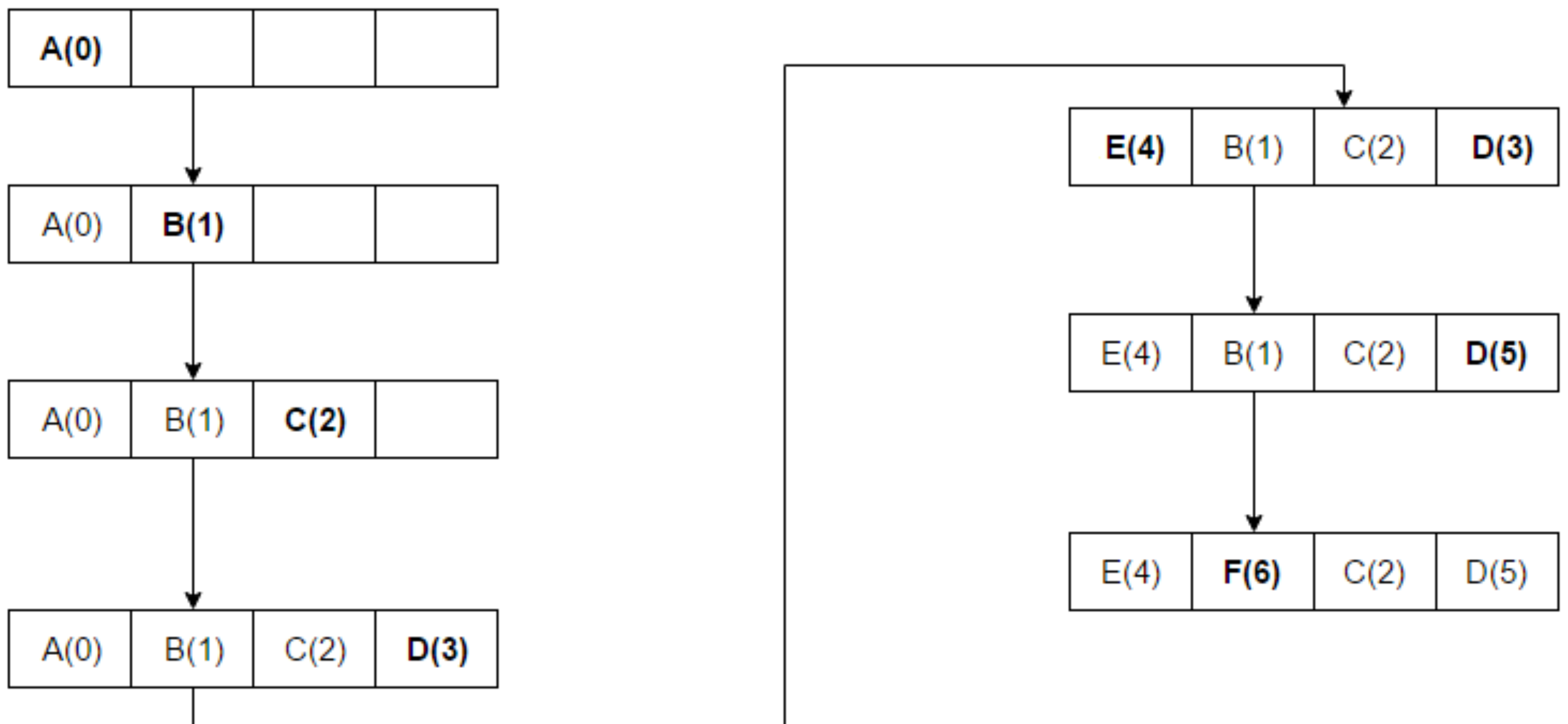
Достоинства:

- простой, константное время доступа
- преимущество по сравнению с FIFO в подстройке под режим доступа к данным; часто используемые объекты скорее всего не вытолкнутся из кэша. Например, подходит для кэширования часто используемых "передовиц" новостных статей.

Недостатки:

- в отличие от MRU не подходит для сканов: может заполниться объектами, которые могут и не понадобиться в ближайшем будущем, например, если делать скан по бОльшему числу объектов, чем помещается в кэш. Тогда выталкиваем не объекты, которые только что обработали, а объекты, которые обработали давно, что не имеет смысла в данном случае.

На диаграмме показана обработка последовательности A B C D E D F.



LFU

LFU - (Least Frequently Used)

- для всех объектов хранится частота их использования;
- в кэше остаются наиболее часто используемые объекты.

Недостатки:

- оверхэд на хранение доп информации растет с размером кэша; также надо отслежить все объекты, независимо от того, в кэше они или нет;
- инертность при смене паттерна использования;
- небыстрый.

Преимущество:

- долгоживущие объекты хорошо покрываются этим сценарием, сканы обрабатываются также хорошо.

LRU2

(Least Recently Used Twice)

- объекты добавляются в кэш только на 2-й раз
- когда кэш заполнен, удаляется 2-й по счету самый недавний объект.
- надо хранить 2 самых последних времени доступа для каждого объекта. Надо хранить инфо по объектам, которых еще нет в кэше.

Преимущества:

- адаптируется к смене паттернов доступа как LRU
- не заполняется при сканировании так как объекты не остаются в кэше после 1-го доступа.

Недостаток:

- не очень быстрый

2Q

(Две очереди) Объекты помещаются в LRU-кэш, когда за ними приходят. Если еще раз приходят, они помещаются во второй, бОльший, LRU-кэш. Объекты обычно чистятся так что размер первого кэша около 1/3 размера второго. Этот алгоритм пытается воспользоваться преимуществами LRU2 при константном оверхэде, без увеличения размера кэша.

Time based expiration

Простой

- Данные в кэше инвалидируются через фиксированный промежуток времени. Объекты добавляются в кэш и остаются там фиксированное время.

Достоинства:

- быстрый

Недостатки:

- не адаптивный
- не защищает от сканов

Расширенный

- Данные инвалидируются через относительные промежутки времени. Например, каждые 5 мин.

Достоинства:

- быстрый

Недостатки:

- не адаптивный
- не защищает от сканов

Скользящий

- Данные в кэше инвалидируются через заданное время, прошедшее после последнего доступа к ним.

Достоинства:

- быстрый

Недостатки:

- адаптивный
- не защищает от сканов

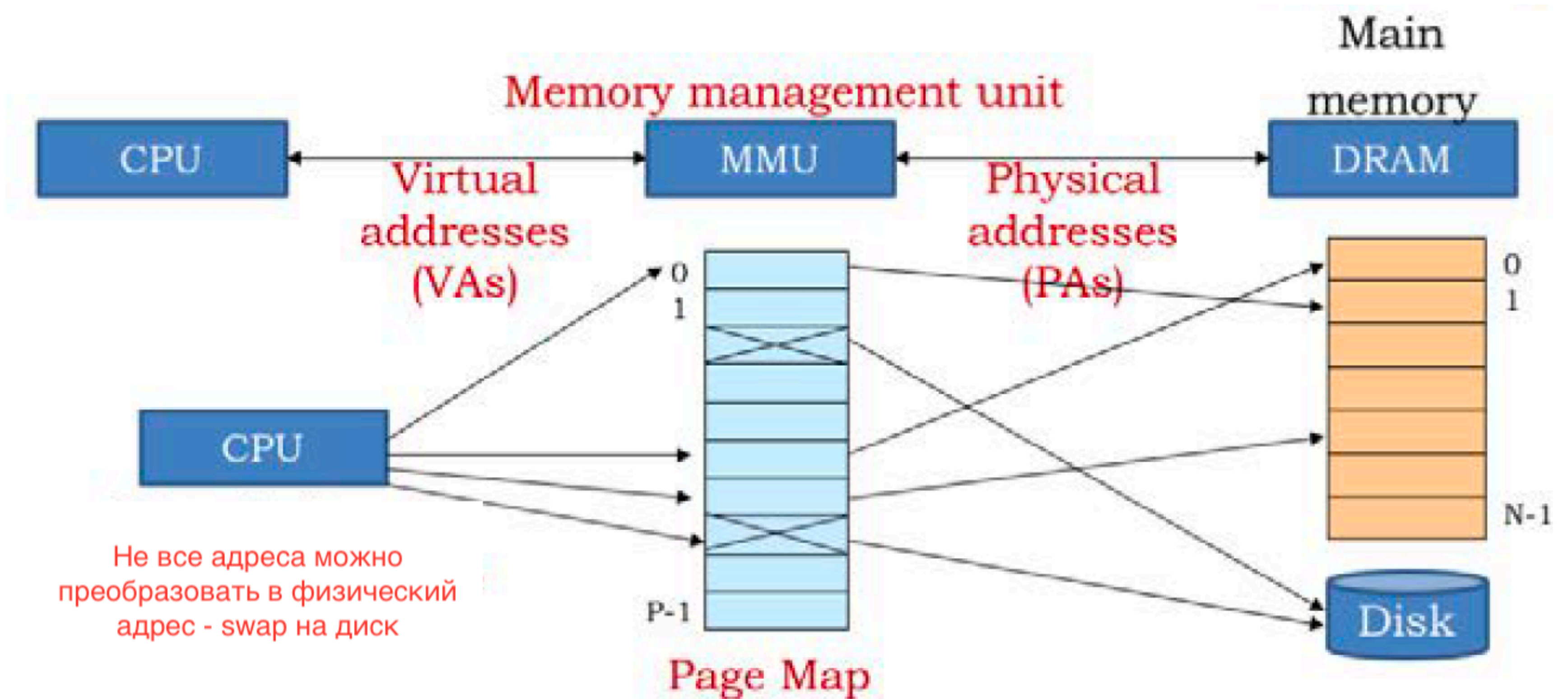
Стратегии записи в кэш

Подробнее рассмотрим позже на примере процессорных кэшей

- write through - запись в кэш сразу приводит к записи в backed storage;
- write back - запись в кэш не сразу приводит к записи в backed storage.

Алгоритмы вытеснения страниц

Как работает менеджер виртуальной памяти в OS.



- менеджеру виртуальной памяти приходит запрос на аллокацию страницы;
- если запрошенная страница отсутствует в памяти, и нет свободной страницы, происходит вытеснение одной из страниц;
- при вытеснении страница виртуальной памяти, "закэшированная" системой, сбрасываются на диск, происходит swap out (page out);
- если на вытесненную страницу происходит обращение, надо сделать page in (чтение с диска);
- чем быстрее будут происходить page-ins, тем лучше.

Алгоритмы вытеснения страниц:

- FIFO - вытесняется самая долгоиспользуемая страница;
- Second Chance - страница вытесняется только на 2-й раз;
- CLOCK - с итератором на страницы;
- Random - вытесняется случайная страница;

FIFO

- работает аналогично FIFO для обычных кэшей;
- хранится очередь из страниц в памяти;
- вытесняется страница в начале очереди, т.е. фрейм, который был в памяти наиболее долгое время, вытесняется.

Достоинства:

- простой;

Недостаток:

- плохой performance.

Применяется в системах VAX/VMS

Second chance

- работает по аналогии с FIFO;
- в отличие от FIFO, вытесняет страницу, только если у нее не установлен reference bit, то есть никто ее не использует;
- если установлен, бит очищается и страница перекидывается в начало очереди;
- очередь работает как circular queue;
- если все страницы имеют установленный reference bit, на второй раз страница убирается, так как у нее уже нет reference bit;
- если у всех страниц нет reference bit, то алгоритм вырождается в обычный FIFO;
- дает странице "второй шанс": если старая страница, возможно, используется, не убирать ее в swap сразу.

CLOCK

- более эффективная версия FIFO, чем Second-chance: страницы не нужно постоянно пушить в конец очереди;
- содержит кольцевой список страниц в памяти, с итератором, указывающим на последнюю проверенную страницу;
- когда нужно вытеснить страницу, проверяется R-бит в позиции итератора;
- если он 0, вытесняется текущая страница;
- иначе бит очищается, итератор инкрементируется и процесс повторяется, пока какая-то страница не вытеснится.

Random

- вытесняется случайная страница;
- работает лучше, чем FIFO.

Что используется в реальности в операционных системах?

Linux

- Linux использует частичную имплементацию CLOCK-PRO;

Windows

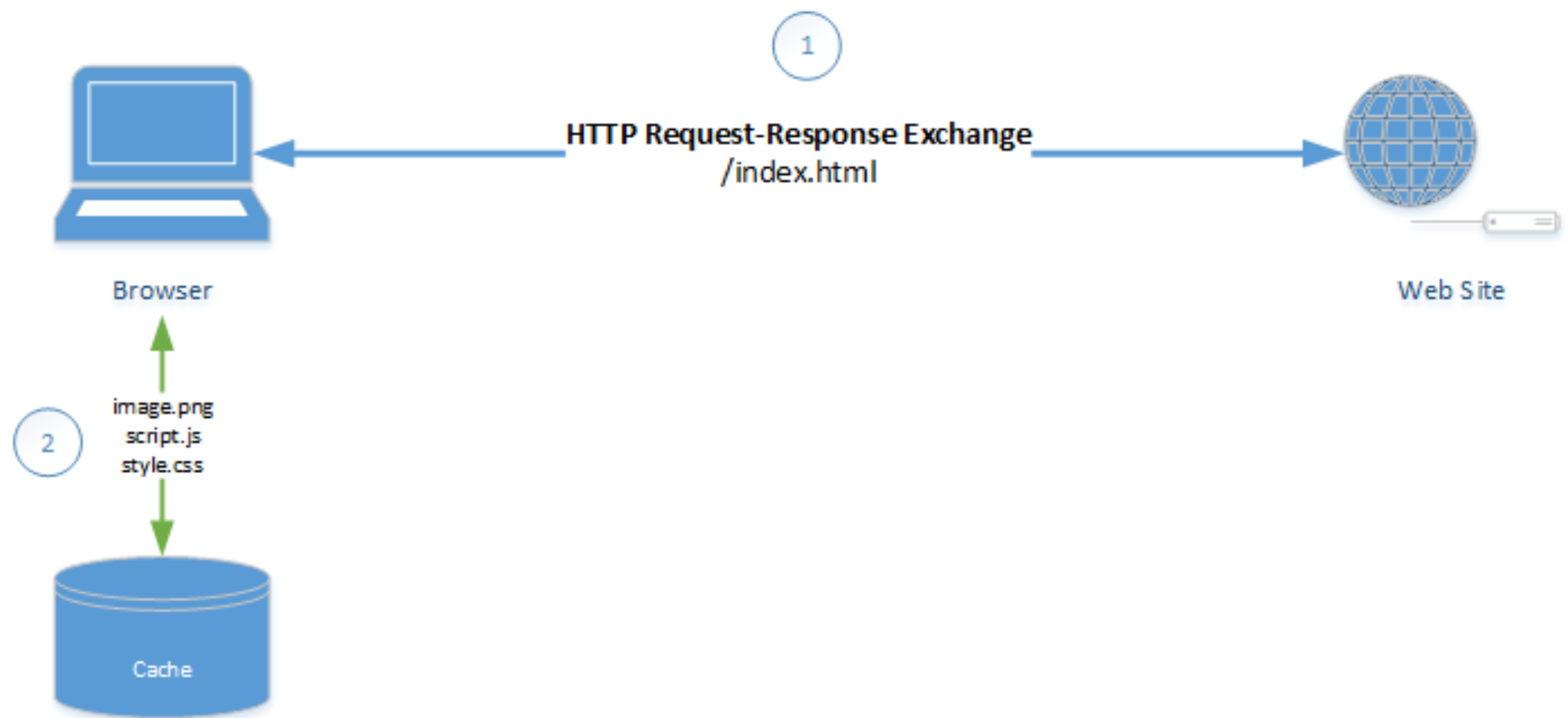
- single processor 80x86 - вариация CLOCK или second chance алгоритма.
- на многопроцессорных системах очистка reference bit инвалидирует TLB других процессоров, это дорогая операция, поэтому используется вариация FIFO.

Примеры использования кэшей:

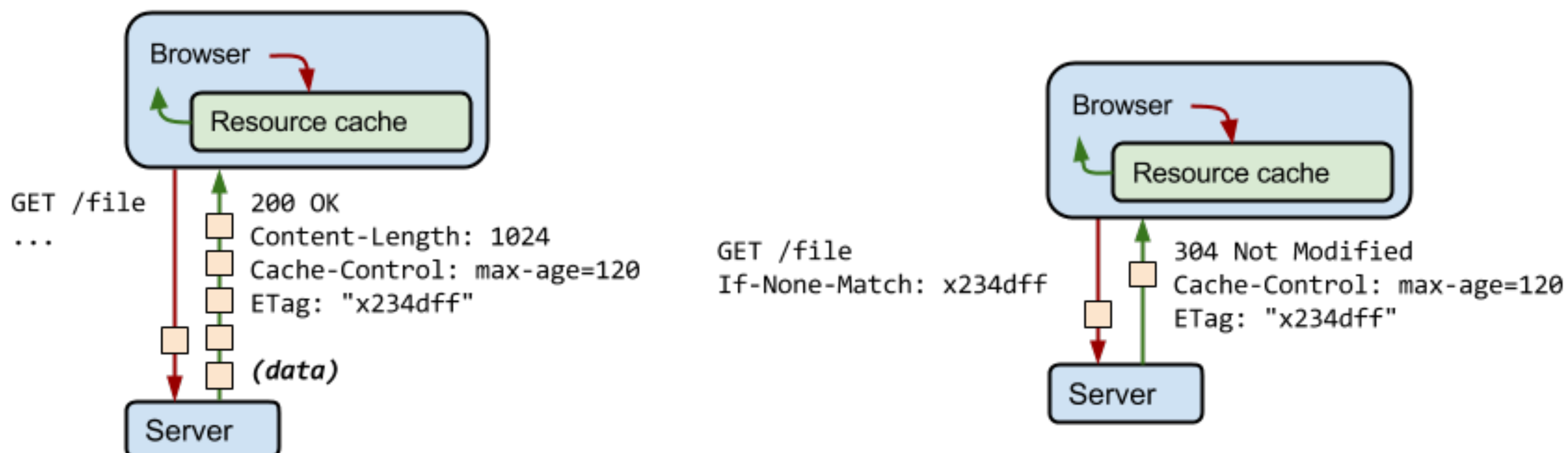
- кэширование веб-страниц;
- кэширование файлов;
- кэширование данных из БД;
- кэширование промежуточных результатов при сборке;
- процессорные кэши;
- кэширование результата выполнения функций.

Пример - кэширование сайта

Кэширование сайта — при посещении веб-страницы, браузер может сохранить данные локально на компьютере, чтобы потом их не скачивать второй раз.



- пользователь посылает запрос на веб-сервер
- в ответе веб-сервер возвращает HTTP-заголовки: content-type, length, опции кэширования и т.д.
- в данном случае сервер возвращает 1024-байтный ответ, сообщает клиенту, что его нужно закэшировать максимум на 120 секунд, и выдает validation token ETag ("x234dff"), который может использоваться для проверки, изменился ли ресурс по истечении срока экспирации.
- если ресурс не изменился, никакие данные не пересылаются.



Предположим, что по истечении 120 секунд с момента первоначального запроса браузер начал новый запрос этого ресурса

- сначала браузер проверяет локальный кэш и находит предыдущий ответ;
- поскольку прошло время экспирации, браузер не может его использовать;
- теперь браузер вынужден послать новый запрос, но если она все же не изменилась, то в этом нет необходимости
- используется хэш от содержимого файла, посылается как ETag;
- браузер посылает назад этот ETag при следующем запросе (в хэдере If-None-Match);
- если файл не изменился, возвращается ответ 304 Not Modified и тот же ETag возвращается назад;
- иначе пересылается новый файл и новый ETag

"max-age"

- указывает максимальное время в секундах, в течение которого можно переиспользовать полученный ответ с момента запроса. Например, "max-age=60" указывает, что мождно закэшировать ответ и переиспользовать его в течение ближайших 60 секунд.

"no-cache" и "no-store"

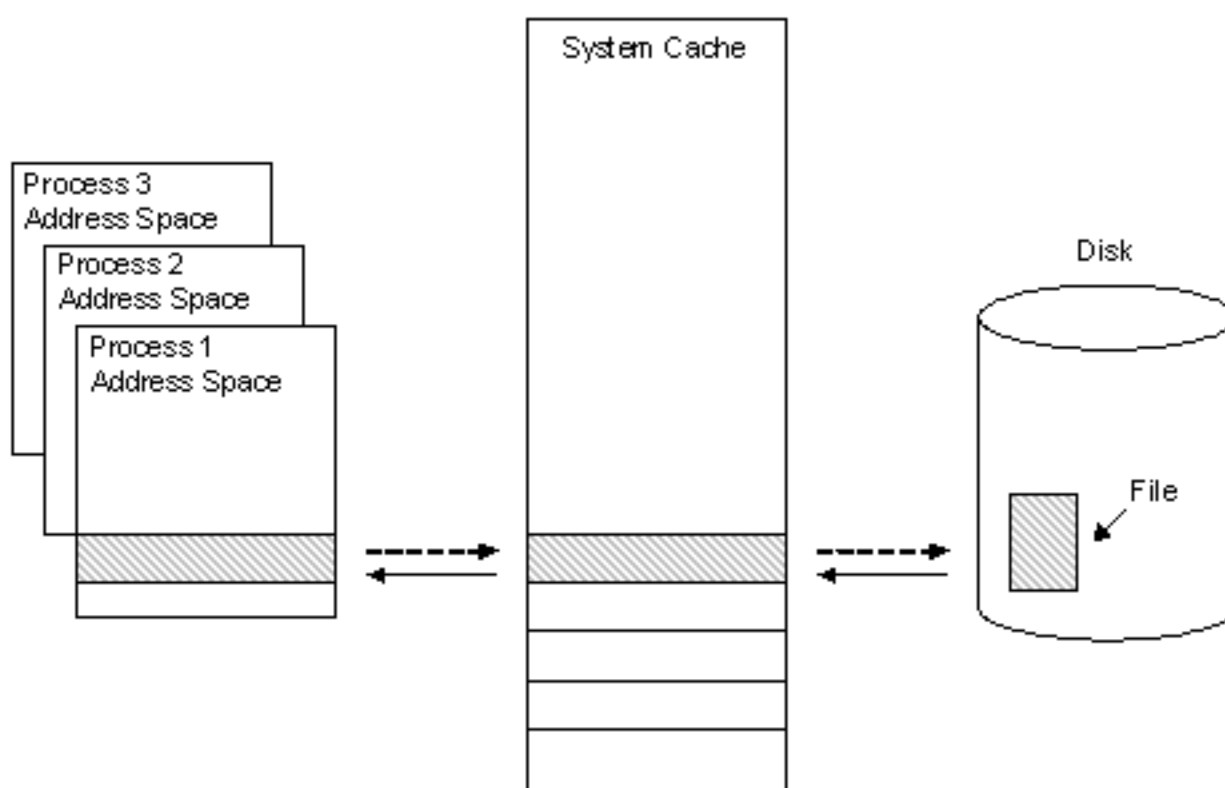
- "no-cache" указывает, что возвращенный ответ нельзя использовать для последующих ответов на тот же запрос без проверки на сервере, что ответ не изменился. Если присутствует ETag, произойдет обращение на сервер для валидации, но не будет скачки файла если ресурс не изменился.
- "no-store" запрещает хранить в кэше данный ответ — например, если он содержит чувствительные данные. Каждый раз при запросе этого ресурса даунлоадится полный ответ.

Пример - кэширование файла

Кэширование файла операционной системой — операции чтения по факту читают файл из системной памяти (system file cache), а не с физического диска.

```
- операции записи пишут в системный кэш, а не на диск.
- данные из системного кэша сбрасываются на диск с регулярными интервалам
и (cache flushing).
```

Такой тип кэша называется ***write-back cache***.



Пример - кэширование данных из БД

Можно кэшировать результат запросов из БД, но есть проблемы с инвалидацией кэша при масштабировании. Несколько бэкэндов. Несколько разных копий одних и тех же данных существуют одновременно в разных кэшах - проблема когерентности кэшей.

Что можно сделать - вопрос к аудитории?

Нужен протокол когерентности

- write update

Когда производится запись в один из кэшей, новые данные также записываются во все остальные кэши.

- write invalidate

Когда производится запись в один из кэшей, все остальные инвалидируются.

MySQL Query Cache

MySQL Query Cache: кэширует результаты SQL-запросов

- ищет входящие запросы, которые начинаются с "SEL", в хэш-таблице, и, если есть совпадение, возвращает результат от предыдущего исполнения того же запроса.

Deprecated с версии MySQL 8.0 Почему? Из-за многочисленных ограничений.

- запрос должен соответствовать байт в байт тому, что в ключе кэша (чтобы избежать парсинга запросов в кэше);
- использование недетерминированных фич приводит к тому, что запрос не кэшируется (включая временные таблицы, пользовательские переменные, RAND(), NOW() и User-defined functions);
- любые модификации любых таблиц, участвующих в каком-либо запросе, приводят к инвалидации ВСЕХ записей с участием этих таблиц в кэше;
- не масштабируется на многоядерные машины.

Лучше применять специальные СУБД для этого, например, Redis (речь ниже).

Caching servers

Memcached

- in-memory распределенная система для кэширования объектов;
- работает с парами "ключ-значение";
- может использоваться, например, для кэширования небольших объектов из СУБД.

In [ ]:

```
# Пример
import memcache
mc = memcache.Client(['localhost:11211'], debug=0)
obj = mc.get(key)
if not obj:
    obj = fetch_obj_from_database(...) # have to pickle into string
    mc.set(key, obj)
return obj
```

Redis

- in-memory хранилище данных;
- можно использовать как базу данных, кэш и message broker;
- в отличие от memcached поддерживает структуры данных: strings, hashes, lists, sets и т.д.

In [ ]:

```
# Пример
import redis

r = redis.StrictRedis(host='localhost', port=6379, db=0)
obj = r.get(key)
if not obj:
    obj = fetch_obj_from_database(...)
    r.set(key, obj)
return obj
```

Кэширование промежуточных результатов при сборке

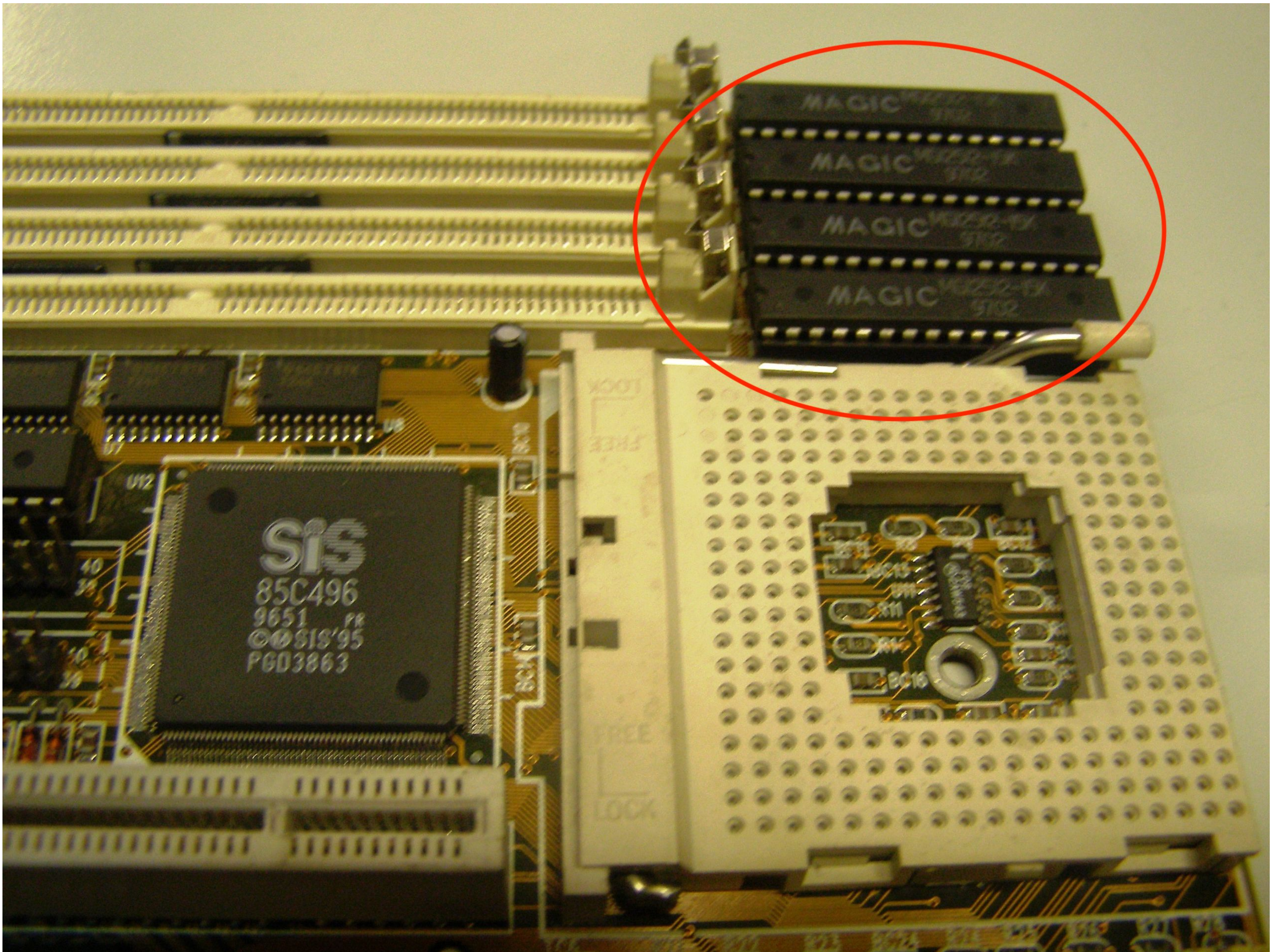
- precompiled headers

precompiled header - это C или C++ хэдер, компилируемый в промежуточную форму, которую компилятор обрабатывает быстрее. Использование precompiled headers может существенно сократить время компиляции, особенно если применяется к большим хэдерам, которые включают в себя множество других, или хэдеров, которые включаются во многие единицы трансляции.

- ccache

ccache на основе компилируемого файла, опций компилятора и выводимых им предупреждений/ошибок вычисляет хэш (по алгоритму md4) и сохраняет его, результат компиляции и все сообщения компилятора в папке кэша (~/.ccache). Если при вызове ccache оказывается, что этот файл уже компилировался (производится сравнение хэша), вместо реального вызова компилятора ccache выдает откомпилированный файл из своего кэша и при необходимости генерирует те же сообщения, которые в нормальных условиях выдал бы компилятор. Разумеется, проверка и выдача из кэша результата гораздо быстрее обычной компиляции.

Пример - процессорные кэши



Четыре микросхемы кэша второго уровня на материнской плате для процессоров семейства i486. Располагаются в буквальном смысле между ЦП и ОЗУ.

Аппаратное кеширование

- на заре компьютерных технологий доступ в память был лишь немного медленнее доступа к процессорным регистрам.
- со времени разрыв в производительности между процессорами и памятью стал нарастать. Память становилась узким местом при достижении полной производительности от системы.
- было решено использовать небольшую, но быструю кэш-память, для смягчения разрыва в производительности.

Общая иерархия памяти (всё является кешом для чего-то другого)

		Access time	Capacity	Managed By
On the datapath	Registers	1 cycle	1 KB	Software/Compiler
On chip	Level 1 Cache	2-4 cycles	32 KB	Hardware
On chip	Level 2 Cache	10 cycles	256 KB	Hardware
On chip	Level 3 Cache	40 cycles	10 MB	Hardware
Other chips	Main Memory	200 cycles	10 GB	Software/OS
Other chips	Flash Drive	10-100us	100 GB	Software/OS
Mechanical devices	Hard Disk	10ms	1 TB	Software/OS

Процессорные кэши

Так называемая "сверхоперативная память"

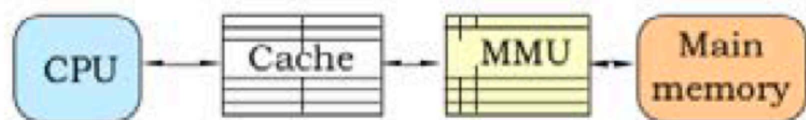
L1, L2 - внутри CPU, L3 - на материнской плате, либо тоже внутри CPU

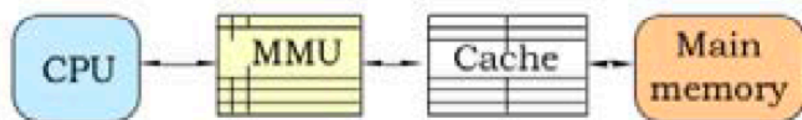
Как они работают?

- процессор выставляет адрес, который он хочет прочитать/записать в кэш;
- два варианта:
  - cache hit - адрес есть в кэше, возвращаются соответствующие данные;
  - cache miss - адеса нет в кэше, делаем запрос в основную память, возвращаем результат процессору; кроме того, обновляем кэш (возможно, вытесняя какие-то другие данные);
- процессор должен понимать, что время выполнения его запроса может варьировааться.

Куда поместить кэш-память? До MMU или за MMU?

До MMU - кешируем виртуальные адреса, за MMU - кешируем физические адреса.





Ваши варианты?

До MMU:

Плюсы:

- отвечаем быстрее, потому что время MMU не учитывается;

Минусы:

- надо сбрасывать кэш при переключении контекста, потому что кешируем виртуальные адреса;

За MMU:

Плюсы:

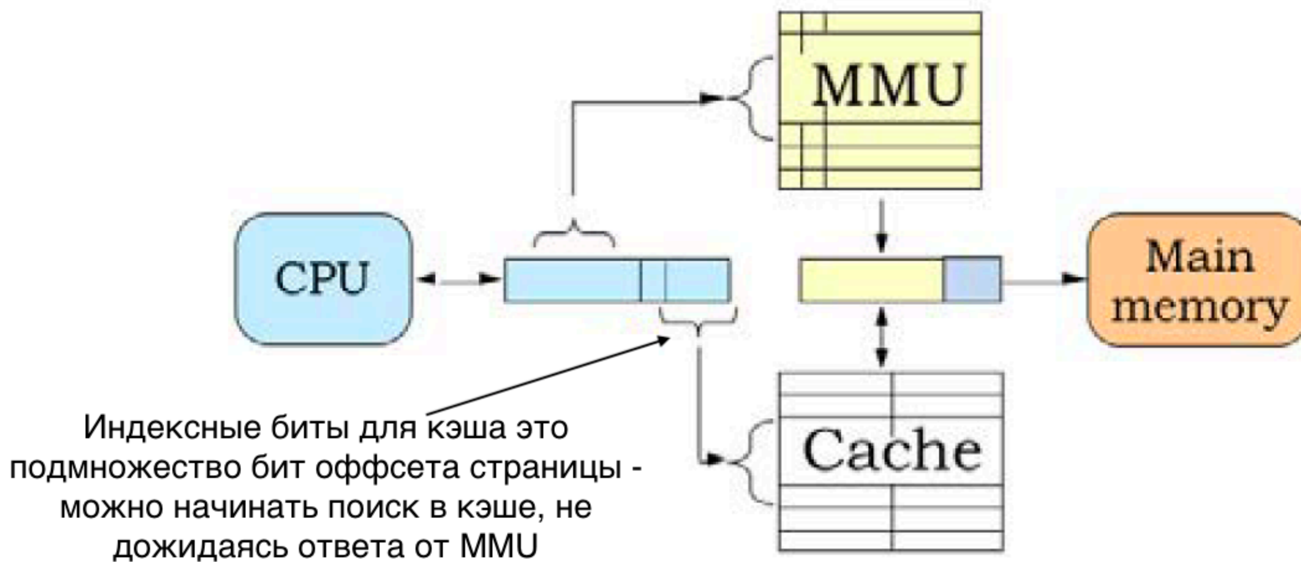
- не надо сбрасывать кэш при переключении контекста, избегаем неактуальных данных;

Минусы:

- медленнее, потому что есть время MMU.

Кэши: оптимальный вариант, берем лучшее из двух подходов

Комбинация двух вариантов: перекрывающаяся адресация



Каждый виртуальный адрес (оффсет страницы) делится на 3 части:

- собственно номер виртуальной страницы;
- индекс массива в кэше;
- индекс в кэш-линии.

Не ждём окончания работы MMU, сразу начинаем запрашивать данные из кеша. Кэш индексируется виртуальными адресами, а тэгируется физическими. В то время, как MMU использует номер виртуальной страницы, чтобы получить номер физической страницы, кэш использует индексную часть, чтобы найти в кэше адрес. Идет параллельный lookup из кэша и трансляция виртуального адреса в физический через MMU. Затем сравнивается физический адрес от MMU физический тэг от кэша. Если они совпадают, был cache hit. Если нет, то cache miss. Тогда возвращается результат из памяти и записывается в кэш.

Достоинства:

- работает быстро, потому что lookup из кэша идет сразу же, и его не надо сбрасывать при переключении контекстов.

Недостаток:

- чтобы увеличить размер кэша, надо увеличивать к-во бит в в виртуальном адресе.

Стратегии выталкивания

- часто применяются варианты LRU

Стратегии записи в кэш

Write-through

- основная память апдейтится каждый раз, когда CPU пишет в кэш

Преимущество:

- основная память всегда содержит то же, что и кэш, поэтому при наличии нескольких CPU локальные кэши других CPU могут обновляться сразу, следя за трафиком основной памяти

Недостатки:

- замедляет запись
- большой трафик

Write-back

- изначально пишем только в кэш
- обновленные участки кэш-памяти помечаются битом обновления и время от времени сбрасываются в основную память
- если блок надо заменить, удалив из кэша, пишем его только если установлен бит обновления

Недостатки:

- кэши других CPU out of sync
- устройства I/O должны обращаться через кэш к главной памяти
- фактически 15% обращений к памяти - это записи

Наиболее часто встречаемая политика записи - это write-back.

Размер линии кэша

- Получаем не только необходимое слово, но и соседние слова
- если увеличить размер блока, увеличится hit ratio - принцип локальности
- если еще увеличить, hit ratio уменьшится - вероятность использования новой информации становится меньше, чем вероятность переиспользования замененной Большие блоки:
- уменьшают к-во блоков, которые размещаются в кэше;
- данные перезаписываются вскоре после того, как были получены из кэша;
- каждое добавочное слово менее локально, то есть вероятность, что оно нужно, уменьшается Не найдено оптимальное значения блока Разумно - от 8 до 64 байт

Многоуровневые кэши

- нужны для разгрузки системной шины
- L1 в CPU, L2 - может быть вне CPU в SRAM
- L1, L2 в CPU, L3 - вне или в CPU

Эволюция кэшей в процессорах Intel

Проблема	Решение	Процессор, в котором фича впервые применена
Внешняя память медленнее, чем системная шина	Добавили внешний кэш с использованием более быстрой технологии памяти	i386
Увеличилась скорость CPU, внешняя шина стала узким горлышком для доступа к кэшу	Внешний кэш перевели в CPU, теперь он работает с частотой процессора	i486
Внутренний кэш становится слишком мал ввиду ограниченного места в чипе CPU	Добавился внешний кэш L2, более быстрый, чнм основная память	i486
Instruction Prefetcher и Execution Unit конкурируют за доступ в кэш - один ждет, пока отработает другой	Сделали отдельные кэши для инструкций и для данных	Pentium
Снова увеличилась скорость CPU, так что теперь L2 стал узким местом	L2 перенесен в CPU	Pentium II
Некоторые приложения работают с массивными базами данных и должны иметь быстрый доступ к большим объемам данных. Внутренние кэши слишком маленькие	Добавили внешний кэш L3	Pentium III
-	Перенесли L3 на кристалл	Pentium 4

Cache-oblivious algorithm

Кэш-oblivious (буквально "забывающий о кэше") или cache-transcendent ("работающий поверх кэша") алгоритм - это алгоритм, учитывающий особенности работы CPU-кэша и пользующийся его преимуществами. Оптимальный cache-oblivious алгоритм это cache-oblivious algorithm, который использует процессорный кэш оптимально.

In [1]:

```
>xeus-cling-cpp14
#include <iostream>
#include <chrono>
#include <random>
#include <vector>

struct TimeLogger
{
    const char* blockName ;
```

```
    const std::chrono::time_point<std::chrono::high_resolution_clock> start_;

    TimeLogger(const char* blockName)
        : blockName_(blockName)
        , start_(std::chrono::high_resolution_clock::now())
    {
    }

    ~TimeLogger()
    {
        auto finish = std::chrono::high_resolution_clock::now();
        auto diff = finish - start_;
        std::cout << blockName_ << ": " << std::chrono::duration_cast<std::chrono
                << " ms passed" << std::endl;
    }
};

const size_t SIZE = 1024 * 1024 * 64;

auto test = [](const char* name, const std::vector<int64_t>& data, const std::vec
    TimeLogger log(name);

    uint64_t sum = 0;
    for (size_t index : indexes)
        sum += data[index];

    return sum;
};

std::random_device rd;
std::default_random_engine e(rd());

auto intData = std::vector<int64_t>(SIZE, 0);
std::generate(intData.begin(), intData.end(), e);

// [0, 1, 2, ... N]
auto idx = std::vector<size_t>(SIZE, 0);
size_t i = 0;
std::generate(idx.begin(), idx.end(), [&i]() { return i++; });

// [N, N-1, ... 0]
auto reverseIdx = std::vector<size_t>(idx.rbegin(), idx.rend());

// random permutation of [0, 1, ... N]
auto randIdx = idx;
std::shuffle(randIdx.begin(), randIdx.end(), e);

size_t dataSize  = intData.size() * sizeof(int64_t);
size_t indexSize = idx.size() * sizeof(int64_t);
std::cout << "vectors filled, data (MB): " << dataSize / 1024 / 1024.0 << "; inde
    << "; total (MB): " << (dataSize + indexSize) / 1024 / 1024.0 << std::endl;

uint64_t sum1 = test("regular access", intData, idx);
uint64_t sum2 = test("reverse access", intData, reverseIdx);
uint64_t sum3 = test("random access", intData, randIdx);

std::cout << sum1 << ", " << sum2 << ", " << sum3 << std::endl;
```

```
vectors filled, data (MB): 512; index (MB): 512; total (MB): 1024
regular access: 62443 ms passed
reverse access: 62469 ms passed
random access: 77731 ms passed
72060347100154493, 72060347100154493, 72060347100154493
```

Выводы

- если данных много, лучше читать из памяти линейно
- если линейно читать не удается, лучше иметь мало данных, чтобы рабочий working set влезал хотя бы в кэш L3 (лучше L2 или L1).

Пример: кэширование результатов функций

- кэшировать можно только "чистые" функции

"Чистая функция" - это функция, которая:

- является детерминированной;
- не обладает побочными эффектами.

Детерминированная функция

Ф-ция является детерминированной, если для одного и того же набора входных значений она возвращает одинаковый результат. В вычислении не участвуют глобальные переменные.

Недетерминированность функции — возможность возвращения функцией разных значений несмотря на то, что ей передаются на вход одинаковые значения входных аргументов.

- невозможно построить однозначную таблицу значений функции;

Побочные эффекты

- модификация значений глобальных переменных;
- осуществление операций ввода-вывода;
- выбрасывать исключения.

Частичная защита в C++ - модификатор const у ф-ций-членов классов. Если constexpr - результат ф-ции детерминирован уже на этапе компиляции. При подаче в нее constexpr выражения, известного на этапе компиляции, получаем constexpr-выражение, также известное на этапе компиляции.

In [ ]:

```
// Примеры "чистых" функций
floor
max
sin
```

In [ ]:

```
// Примеры "нечистых" функций
// возврат глобальной переменной
int f() {
  return x;
}

// возврат мутабельного аргумента
int f(int* x) {
  return *x;
}

// модификация статической переменной
void f() {
  static int x = 0;
  ++x;
}

// модификация мутабельного аргумента
void f(int* x) {
  ++*x;
}

// вывод
void f() {
  std::cout << "Hello, world!" << std::endl;
}
```

Практика

Кэширующие библиотеки для python-a

python - functools.lru_cache, cachetools, cacheout

pip install cacheout

pip install cachetools

In [4]:

```
import time

def timeit(method):

    def timed(*args, **kw):
        ts = time.time()
        result = method(*args, **kw)
        te = time.time()

        print ('%r (%r, %r) %2.2f sec' % \
              (method.__name__, args, kw, te-ts))
        return result

    return timed
```

In [2]:

```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

@timeit
def fibo():
    print ([fib(n) for n in range(35)])

fibo()
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1
597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 19
6418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887]
'fibo' ((), {}) 4.57 sec
```

In [4]:

```
# Decorator to wrap a function with a memoizing callable that saves up to the max
# If typed is set to true, function arguments of different types will be cached s
# @functools.lru_cache(maxsize=128, typed=False)
import functools

@functools.lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

@timeit
def fibo():
    print ([fib(n) for n in range(35)])

fibo()

print (fib.cache_info())
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1
597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 19
6418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887]
'fibo' ((), {}) 0.00 sec
CacheInfo(hits=66, misses=35, maxsize=None, currsize=35)
```

In [5]:

```
import time
def request_rest_api(query):
    time.sleep(1)
    return 1 if query == 'good' else 2

@timeit
def work():
    for i in range(0,5):
        print(request_rest_api('good'))
        print(request_rest_api('bad'))
work()
```

```
1
2
1
2
1
2
1
2
1
2
'work' ((), {}) 10.04 sec
```

In [6]:

```python
import functools

@functools.lru_cache(maxsize=128, typed=False)
def request_rest_api(query):
    time.sleep(1)
    return 1 if query == 'good' else 2

@timeit
def work():
    for i in range(0,5):
        print(request_rest_api('good'))
        print(request_rest_api('bad'))
work()
request_rest_api.cache_info()
```

```
1
2
1
2
1
2
1
2
1
2
'work' ((), {}) 2.01 sec
```

Out[6]:

```
CacheInfo(hits=8, misses=2, maxsize=128, currsize=2)
```

In [ ]:

```python
## cachetools
# cachetools.Cache(maxsize, getsizeof=None)
# cachetools.LFUCache(maxsize, getsizeof=None)
# cachetools.LRUCache(maxsize, getsizeof=None)
# cachetools.RRCache(maxsize, choice=random.choice, getsizeof=None)
# cachetools.TTLCache(maxsize, ttl, timer=time.monotonic, getsizeof=None)
```

In [7]:

```
from cachetools import cached, TTLCache # импортируем декоратор и объект TTLCache
cache = TTLCache(maxsize=2, ttl=300)  # создаем объект кэша, maxsize - максимальн
@cached(cache)  # декорируем метод, чтобы использовать кэш
def request_rest_api(query):
    time.sleep(1)
    return 1 if query == 'good' else 2

@timeit
def work():
    for i in range(0,5):
        print(request_rest_api('good'))
        print(request_rest_api('bad'))
work()
```

```
1
2
1
2
1
2
1
2
1
2
'work' ((), {}) 2.01 sec
```

In [2]:

```
# Сделаем свою реализацию LRUCache
class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity # емкость кэша
        self.tm = 0              # счетчик к-ва чтений и записей в кэш ("таймстемп
        self.cache = {}          # сам кэш
        self.lru = {}            # использованные ключи

    def get(self, key):
        if key in self.cache:
            self.lru[key] = self.tm
            self.tm += 1
            return self.cache[key]
        return -1

    def set(self, key, value):
        if len(self.cache) >= self.capacity:
            # find the LRU entry
            old_key = min(self.lru.keys(), key=lambda k:self.lru[k])
            self.cache.pop(old_key) # выталкивается самый давно используемый ключ
            self.lru.pop(old_key)
        self.cache[key] = value
        self.lru[key] = self.tm
        self.tm += 1
```

Вопрос: что не так с этой реализацией? Как ее исправить?

In [1]:

```
import collections

class LRUCacheFixed:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = collections.OrderedDict() # запоминает порядок, в котором бы

    def get(self, key):
        try:
            value = self.cache.pop(key)
            self.cache[key] = value
            return value
        except KeyError:
            return -1

    def set(self, key, value):
        try:
            self.cache.pop(key) # убрать ключ
        except KeyError:
            if len(self.cache) >= self.capacity:
                self.cache.popitem(last=False) # вернуть и удалить пару "ключ-зна
        self.cache[key] = value # перевыставить ключ-значение
```

In [13]:

```
def testcache(Cache):
    @timeit
    def verify(cache):
        cache.set('test', 1)
        assert cache.get('test')

        for j in range(20):
            for i in range(10000):
                cache.set(i, i)

        for i in range(10000):
            cache.get(i)

    cache = Cache(100)
    verify(cache)

testcache(LRUCache)
testcache(LRUCacheFixed)
```

```
'verify' ((<__main__.LRUCache object at 0x107f3b9e8>,), {}) 2.35 sec
'verify' ((<__main__.LRUCacheFixed object at 0x107b407f0>,), {}) 0.1
5 sec
```

Кэширующие библиотеки для Java

- org.apache.commons.collections.map.LRUMap;
- Spring Framework - кэширование методов с помощью аннотаций;
- Google Guava.

In [ ]:

```
>java
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Component;

@Component
public class MathService {

    @Cacheable("piDecimals")
    public int computePiDecimal(int i) {
        // ...
    }

}
```

Кэширование в многопоточном окружении

Кэши и подходы, которые мы рассмотрели, как правило не предназначены для многопоточного использования и требуют отдельных усилий для обеспечения многотопочной работы.

- можно использовать мьютексы (производительно ухудшится);
- можно использовать lock-free подход: single writer и multiple readers.

Домашнее задание - написать свою реализацию скользящего timed-кэша на вашем языке программирования (C/C++/Java/Python)

Цели:

- лучше понять, как устроен один из видов кэша - скользящий timed cache.

Задача:

- сделать свой скользящий timed cache, реализовав функции get, set.
- кэш должен параметризоваться размером и таймаутом - временем, прошедшим с момента последнего доступа к объекту, по истечении которого объект вычищается из кэша.

Требования к реализации:

- в кэш приходят случайные объекты, в любом объеме и с любой частотой;
- стоит уделить время производительности кэша, чтобы он отвечал на запросы так быстро, как только возможно;
- должны присутствовать тесты, показывающие правильность работы кэша;
- (опционально) написать тесты производительности, меряющие производительность вашего кэша и сравнивающие его с этой же (библиотечной) и/или иными реализациями кэша в вашем ЯП (если есть).

Заполните опросник в конце занятия:
https://docs.google.com/forms/d/e/1FAIpQLSfGj4BY8YxblJ9SXVmqBOvpmkN1BZGb2LfldaV3y8ZLUK4BFQ
(https://docs.google.com/forms/d/e/1FAIpQLSfGj4BY8YxblJ9SXVmqBOvpmkN1BZGb2LfldaV3y8ZLUK4BFQ

Полезные ссылки и литература

Математика

- Some mathematical facts about optimal cache replacement. Pierre Michaud. https://hal.inria.fr/hal-01411156v2/document (https://hal.inria.fr/hal-01411156v2/document)

Архитектура компьютерных систем

- The Cache Memory Book (The Morgan Kaufmann Series in Computer Architecture and Design).

О кэшах процессора

- О кэшах процессора: https://lwn.net/Articles/252125/ (https://lwn.net/Articles/252125/) - "What every programmer should know about memory". Ulrich Drepper. Part 2. CPU Caches. http://rus-linux.net/MyLDP/hard/memory/memory-03-01.html (http://rus-linux.net/MyLDP/hard/memory/memory-03-01.html) - русский перевод.
- Написание cache-oblivious приложений - https://lwn.net/Articles/255364/ (https://lwn.net/Articles/255364/) - "What every programmer should know about memory". Ulrich Drepper. Memory part 5: What programmers can do. http://rus-linux.net/lib.php?name=/MyLDP/hard/memory/memory-6-1.html (http://rus-linux.net/lib.php?name=/MyLDP/hard/memory/memory-6-1.html) - Что могут делать программисты - оптимизация кэша - русский перевод.
- О кэшах процессора в контексте оптимизации программ - доклад Андрея Аксенова "Низкоуровневая оптимизация C/C++" на HighLoad 2011: http://profyclub.ru/docs/146 (http://profyclub.ru/docs/146)

memcached, redis

- Redis in Action. Josiah L Carlson.

OTUS

Спасибо
за внимание!

OTUS

ОНЛАЙН-ОБРАЗОВАНИЕ

OTUS

Кэширование

Михаил Горшков
разработчик

ВКЛЮЧИТЬ ЗАПИСЬ!!!

Заполните опросник перед началом занятия: https://docs.google.com/forms/d/e/1FAIpQLSe_X0-ueYIVqPEnTOPcbVGkrteLmCjpDBFfsgiQj7JSvEBKvA/viewform (https://docs.google.com/forms/d/e/1FAIpQLSe_X0-ueYIVqPEnTOPcbVGkrteLmCjpDBFfsgiQj7JSvEBKvA/viewform)

Общая информация про кэширование

Вопрос к аудитории: приведите примеры кэшей

Кэш - от англ. *cache* (не "наличные деньги" - cash, а от фр. *cacher* — "прятать");

Что такое кеширование, зачем применяется, что дает.

Примерное объяснение - вы обращаетесь к более медленному ресурсу, а результат обращения храните на более быстром ресурсе для того, чтобы впоследствии воспользоваться этим результатом быстрее. Для пользователя этот более быстрый ресурс "спрятан", он его не замечает.

Размещение данных в специально отведенном месте для ускоренного доступа к ним в будущем.

Как работает кэш

- когда пользователь системы с кэшированием запрашивает данные, сначала проверяется кэш;
- если он содержит требуемые данные, то используются они без обращения к основному хранилищу данных;
- если данных в кэше нет, они запрашиваются в хранилище и добавляются в кэш;
- когда кэш заполняется, данные из него каким-то образом очищаются, чтобы освободить место для новых данных.

Вопросы к аудитории:

- Что такое 'cache hit'?
- Что такое 'cache miss' (промах)?
- Что такое 'hit rate'?
- Что такое 'miss rate'?
- Что такое 'hit time'?
- Что такое 'вымывание' или 'загрязнение' кэша (cache pollution)?
- Что такое 'cache coherence'?
- Что такое 'cache oblivious'?

Термины

- cache hit - объект найден в кэше;
- cache miss - объект не найден в кэше;
- hit rate - total number of hits / (total number of hits + total number of misses) * 100%; Если 100% - все нашли в кэше, если 0% - ничего не нашли, если 50% - нашли половину
- miss rate - total number of misses / (total number of hits + total number of misses) * 100%;
- hit time - время хита;
- cache pollution - вымывание/загрязнение кэша - быстрое удаление элементов из него ввиду постоянной записи новых данных, недостаточного размера кэша, неподходящей стратегии его реализации или иных причин, ввиду чего кэш дает почти всегда 100% miss rate;
- cache coherence - поддержание одинаковости разных локальных кэшей, кэширующих одно и то же - например, кэш процессора в многопроцессорной системе: в одном кэше изменились данные => надо обновить остальные;
- cache oblivious алгоритм - алгоритм, учитывающий и использующий особенности кэша;

Когда может использоваться кэширование?

- вы производите "дорогие" вычисления (например, сложные запросы к БД или ресурсоемкие вычисления) - тогда можете закэшировать результат;
- когда можно переиспользовать ранее вычисленное значение и это целесообразно;
- MRU кэш для пользовательских запросов (последних открытых имен файлов в редакторах).

Когда не нужно писать свой кэш?

- не нужно дублировать уже существующий кэш, например, файловый;
- бывают ситуации, когда кэшировать нецелесообразно, например, вычислить что-либо стоит столько же, сколько взять из кэша.

Вопрос к аудитории: можно ли кэшировать результат функции, использующей для своих вычислений текущее время?

Нельзя кэшировать!!

- результат вычисления функций с сайд-эффектами, функций, которые создают какие-либо объекты на каждом вызове;
- результат функций, предназначенных для генерирования недетерминированных данных, например time() или random();
- поговорим о кэшировании функций далее.

Стратегии и алгоритмы общецелевого кэширования (процессорный, дисковый, БД, веб)

Стратегии "выталкивания" элементов из кэша

Когда требуемых данных нет в кэше, это cache miss. Над принять решение: хранить ли данные, полученные из основного хранилища, в кэше, и, если да, то какие данные убрать из кэша, чтобы дать место новым? Это т.н. политика замены, replacement policy. Наиболее эффективная стратегия - это всегда убирать объект, который не будет нужен в будущем. В 1966 Belady разработал replacement policy, минимизирующую число промахов. Доказано, что этот алг-м оптимальный (см ссылку в конце). На практике этот оптимальный алгоритм невозможно применить, можно только сравнить эффективность разных алгоритмов кэширования в данных условиях.

- FIFO - первым добавлен, первым вытолкнется
- LIFO - последним добавлен, первым вытолкнется
- MRU - выталкивается последний используемый
- LRU - выталкивается наиболее давно используемый
- LFU - выталкивается наименее часто используемый
- LRU2 - выталкивается наименее используемый 2 раза
- 2Q - 2 очереди
- Time-based expiration - выталкиваются давно размещенные

FIFO

(First In First Out):

- объекты добавляются в кэш по мере доступа к ним, располагаются в очереди (буфере) и не меняют своего положения в буфере;
- когда кэш заполняется, объекты удаляются в порядке, в котором они были добавлены;
- время доступа к кэшу - константное независимо от размера кэша (можно реализовать через массив с прямым поиском).

Преимущества:

- простой и быстрый;

Недостаток:

- не очень "интеллектуальный"; нет функционала для хранения часто используемых объектов.

LIFO

(Last In First Out): похож на FIFO, но при заполнении кэша объекты выталкиваются в обратном порядке.

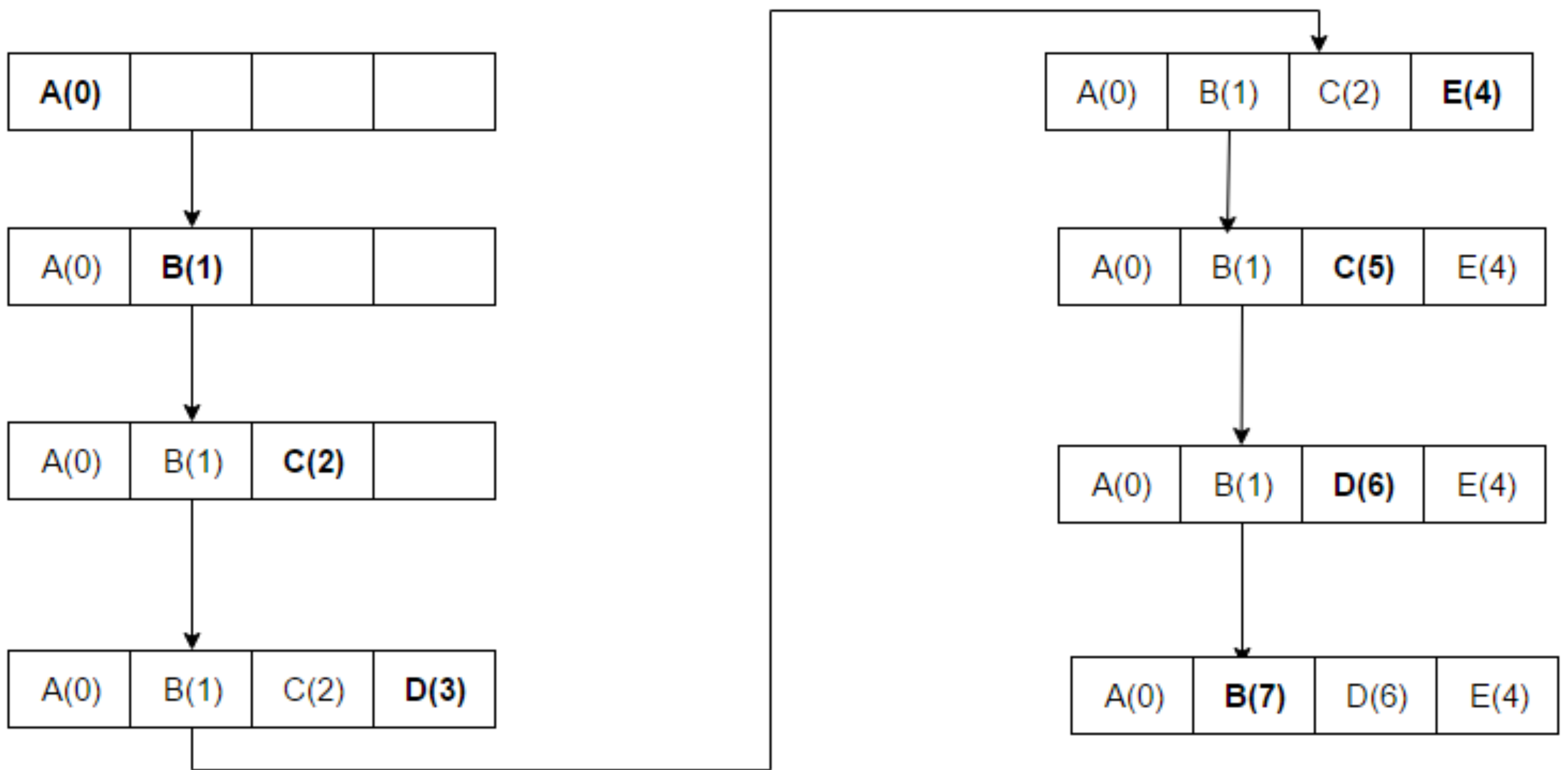
- для хранения используется стек;

MRU

Выталкивается последний использованный

- подходит для сканов - один раз прошли по объекту, больше не вернемся к нему, можно выталкивать;
- используется в ситуациях, когда чем старее объект, тем больше вероятность, что он будет затребован.

На диаграмме показана обработка последовательности A B C D E C D B



LRU

(Least Recently Used)

Наиболее часто используемый алгоритм кэширования.

- объекты добавляются в кэш по мере доступа к ним;
- когда кэш заполняется, выталкивается объект, который читался (использовался) максимально давно.
- обычно реализуется как связный список, так что объект кэша, который достается, перемещается в голову; объекты удаляются с конца списка;

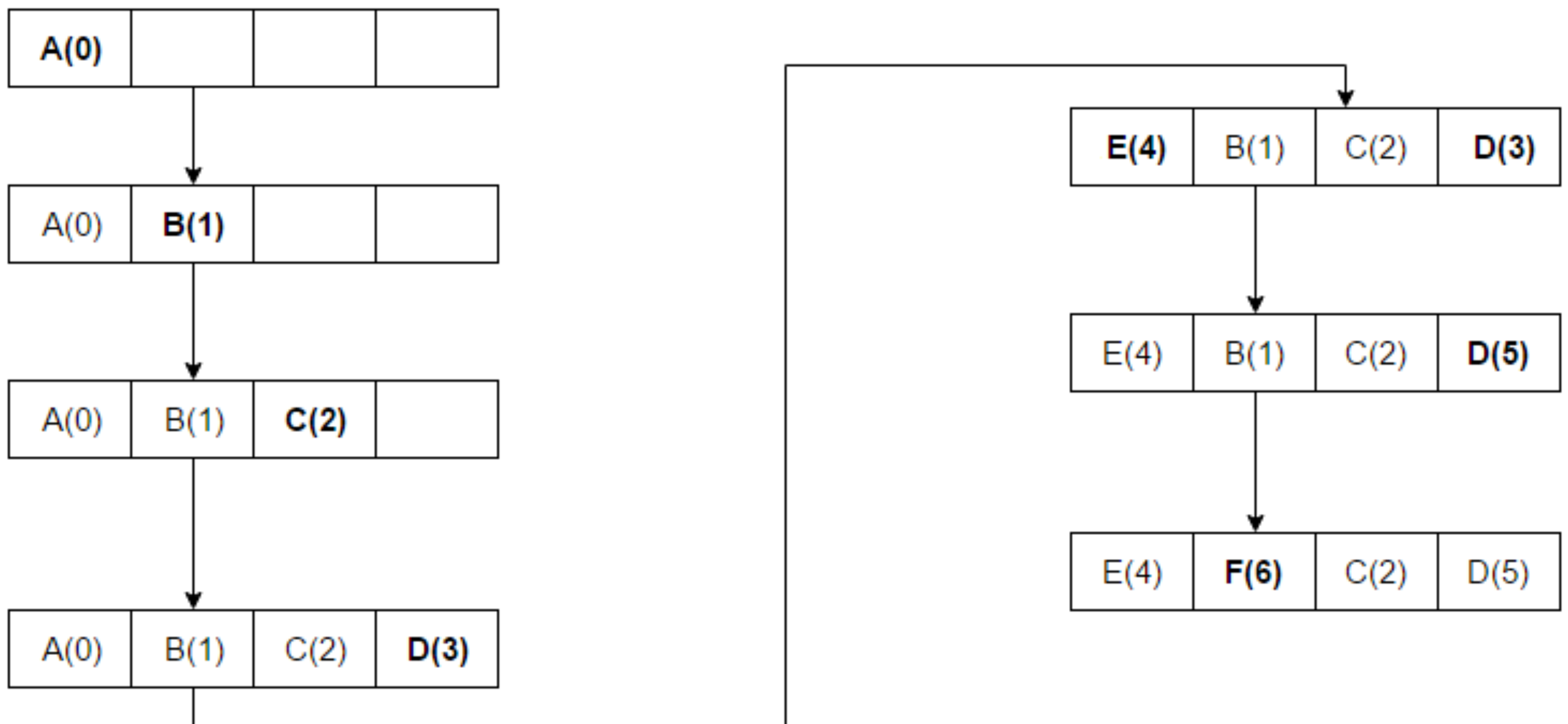
Достоинства:

- простой, константное время доступа
- преимущество по сравнению с FIFO в подстройке под режим доступа к данным; часто используемые объекты скорее всего не вытолкнутся из кэша. Например, подходит для кэширования часто используемых "передовиц" новостных статей.

Недостатки:

- в отличие от MRU не подходит для сканов: может заполниться объектами, которые могут и не понадобиться в ближайшем будущем, например, если делать скан по бОльшему числу объектов, чем помещается в кэш. Тогда выталкиваем не объекты, которые только что обработали, а объекты, которые обработали давно, что не имеет смысла в данном случае.

На диаграмме показана обработка последовательности A B C D E D F.



LFU

LFU - (Least Frequently Used)

- для всех объектов хранится частота их использования;
- в кэше остаются наиболее часто используемые объекты.

Недостатки:

- оверхэд на хранение доп информации растет с размером кэша; также надо отслежить все объекты, независимо от того, в кэше они или нет;
- инертность при смене паттерна использования;
- небыстрый.

Преимущество:

- долгоживущие объекты хорошо покрываются этим сценарием, сканы обрабатываются также хорошо.

LRU2

(Least Recently Used Twice)

- объекты добавляются в кэш только на 2-й раз
- когда кэш заполнен, удаляется 2-й по счету самый недавний объект.
- надо хранить 2 самых последних времени доступа для каждого объекта. Надо хранить инфо по объектам, которых еще нет в кэше.

Преимущества:

- адаптируется к смене паттернов доступа как LRU
- не заполняется при сканировании так как объекты не остаются в кэше после 1-го доступа.

Недостаток:

- не очень быстрый

2Q

(Две очереди) Объекты помещаются в LRU-кэш, когда за ними приходят. Если еще раз приходят, они помещаются во второй, бОльший, LRU-кэш. Объекты обычно чистятся так что размер первого кэша около 1/3 размера второго. Этот алгоритм пытается воспользоваться преимуществами LRU2 при константном оверхэде, без увеличения размера кэша.

Time based expiration

Простой

- Данные в кэше инвалидируются через фиксированный промежуток времени. Объекты добавляются в кэш и остаются там фиксированное время.

Достоинства:

- быстрый

Недостатки:

- не адаптивный
- не защищает от сканов

Расширенный

- Данные инвалидируются через относительные промежутки времени. Например, каждые 5 мин.

Достоинства:

- быстрый

Недостатки:

- не адаптивный
- не защищает от сканов

Скользящий

- Данные в кэше инвалидируются через заданное время, прошедшее после последнего доступа к ним.

Достоинства:

- быстрый

Недостатки:

- адаптивный
- не защищает от сканов

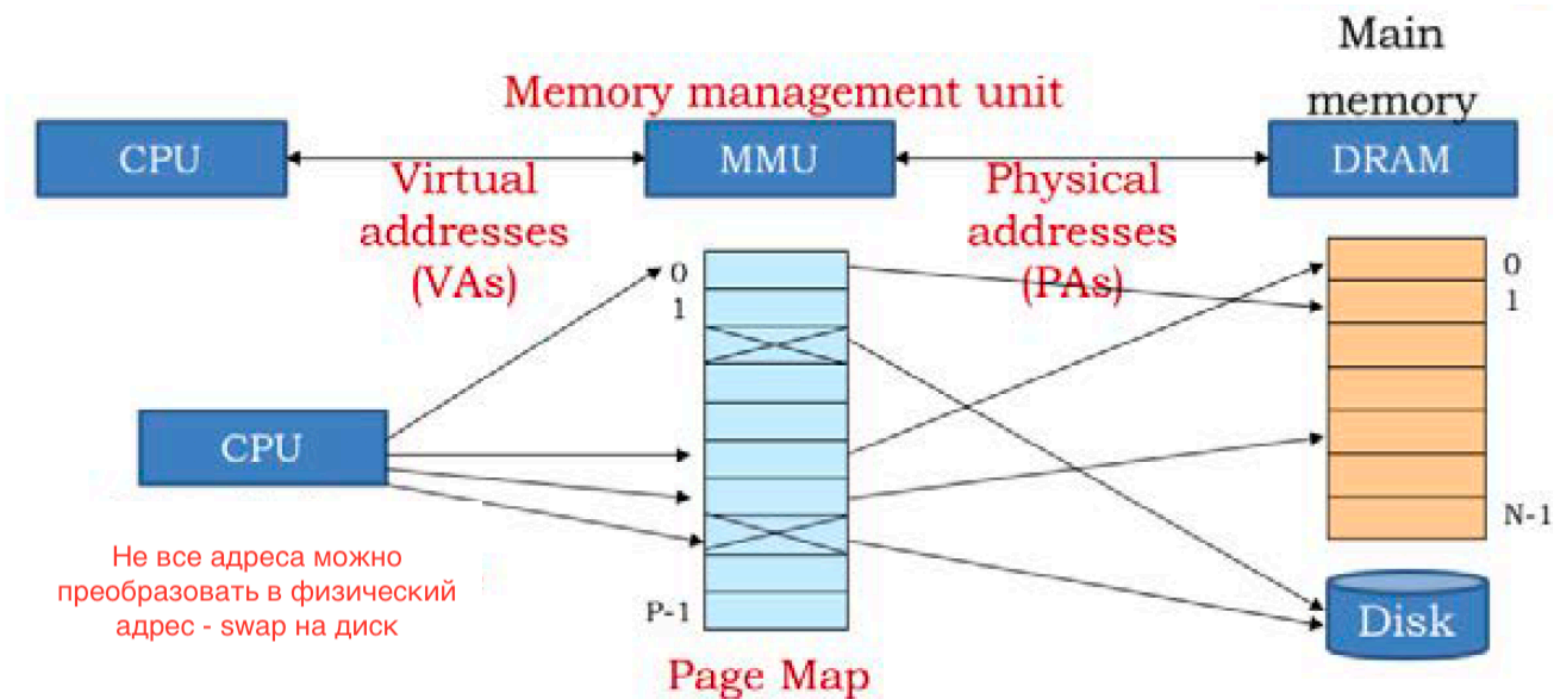
Стратегии записи в кэш

Подробнее рассмотрим позже на примере процессорных кэшей

- write through - запись в кэш сразу приводит к записи в backed storage;
- write back - запись в кэш не сразу приводит к записи в backed storage.

Алгоритмы вытеснения страниц

Как работает менеджер виртуальной памяти в OS.



- менеджеру виртуальной памяти приходит запрос на аллокацию страницы;
- если запрошенная страница отсутствует в памяти, и нет свободной страницы, происходит вытеснение одной из страниц;
- при вытеснении страница виртуальной памяти, "закэшированная" системой, сбрасываются на диск, происходит swap out (page out);
- если на вытесненную страницу происходит обращение, надо сделать page in (чтение с диска);
- чем быстрее будут происходить page-ins, тем лучше.

Алгоритмы вытеснения страниц:

- FIFO - вытесняется самая долгоиспользуемая страница;
- Second Chance - страница вытесняется только на 2-й раз;
- CLOCK - с итератором на страницы;
- Random - вытесняется случайная страница;

FIFO

- работает аналогично FIFO для обычных кэшей;
- хранится очередь из страниц в памяти;
- вытесняется страница в начале очереди, т.е. фрейм, который был в памяти наиболее долгое время, вытесняется.

Достоинства:

- простой;

Недостаток:

- плохой performance.

Применяется в системах VAX/VMS

Second chance

- работает по аналогии с FIFO;
- в отличие от FIFO, вытесняет страницу, только если у нее не установлен reference bit, то есть никто ее не использует;
- если установлен, бит очищается и страница перекидывается в начало очереди;
- очередь работает как circular queue;
- если все страницы имеют установленный reference bit, на второй раз страница убирается, так как у нее уже нет reference bit;
- если у всех страниц нет reference bit, то алгоритм вырождается в обычный FIFO;
- дает странице "второй шанс": если старая страница, возможно, используется, не убирать ее в swap сразу.

CLOCK

- более эффективная версия FIFO, чем Second-chance: страницы не нужно постоянно пушить в конец очереди;
- содержит кольцевой список страниц в памяти, с итератором, указывающим на последнюю проверенную страницу;
- когда нужно вытеснить страницу, проверяется R-бит в позиции итератора;
- если он 0, вытесняется текущая страница;
- иначе бит очищается, итератор инкрементируется и процесс повторяется, пока какая-то страница не вытеснится.

Random

- вытесняется случайная страница;
- работает лучше, чем FIFO.

Что используется в реальности в операционных системах?

Linux

- Linux использует частичную имплементацию CLOCK-PRO;

Windows

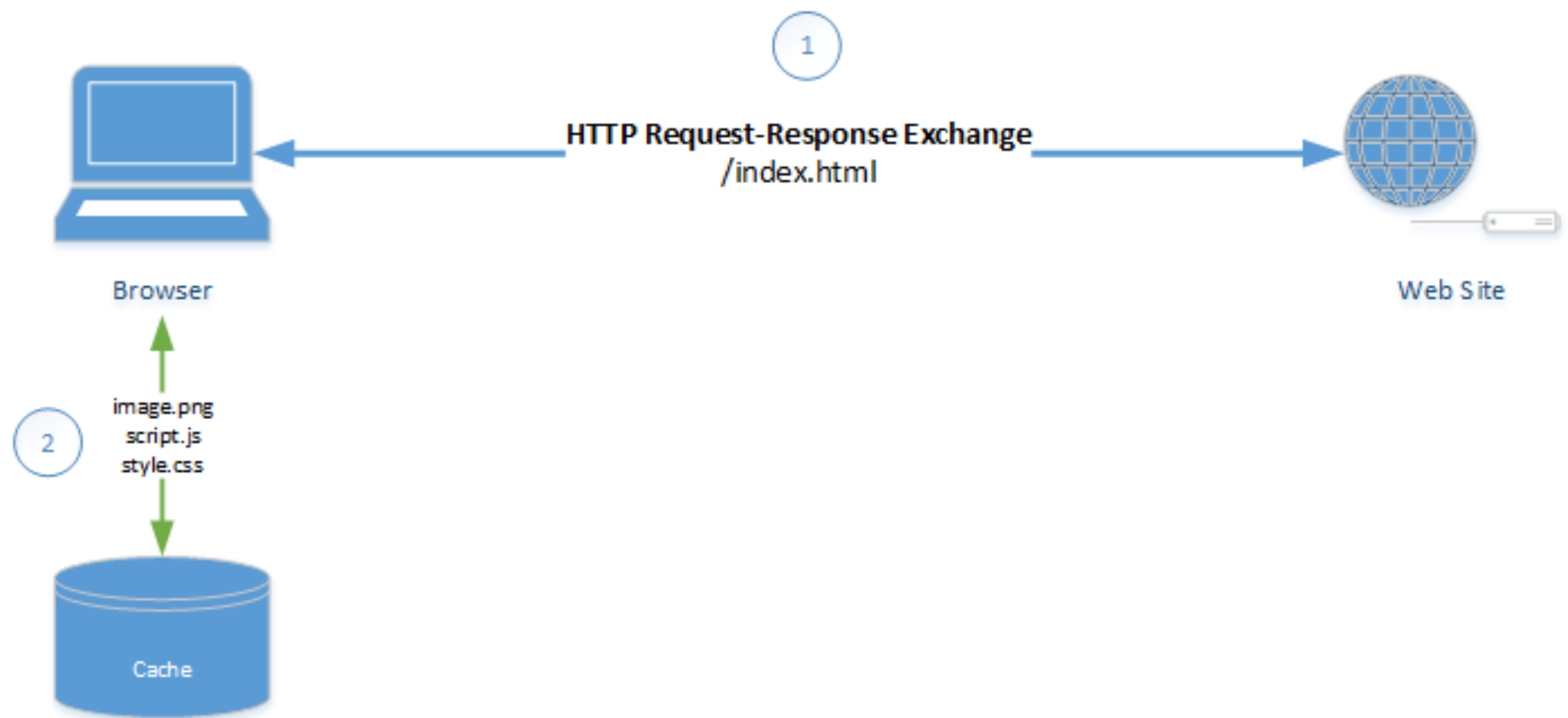
- single processor 80x86 - вариация CLOCK или second chance алгоритма.
- на многопроцессорных системах очистка reference bit инвалидирует TLB других процессоров, это дорогая операция, поэтому используется вариация FIFO.

Примеры использования кэшей:

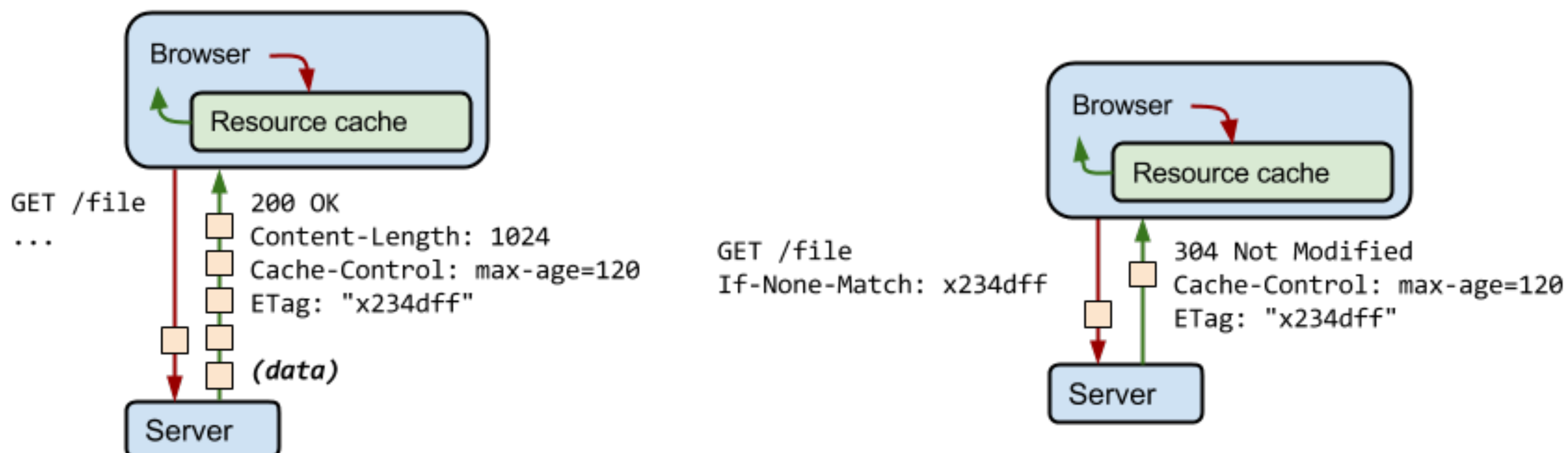
- кэширование веб-страниц;
- кэширование файлов;
- кэширование данных из БД;
- кэширование промежуточных результатов при сборке;
- процессорные кэши;
- кэширование результата выполнения функций.

Пример - кэширование сайта

Кэширование сайта — при посещении веб-страницы, браузер может сохранить данные локально на компьютере, чтобы потом их не скачивать второй раз.



- пользователь посылает запрос на веб-сервер
- в ответе веб-сервер возвращает HTTP-заголовки: content-type, length, опции кэширования и т.д.
- в данном случае сервер возвращает 1024-байтный ответ, сообщает клиенту, что его нужно закэшировать максимум на 120 секунд, и выдает validation token ETag ("x234dff"), который может использоваться для проверки, изменился ли ресурс по истечении срока экспирации.
- если ресурс не изменился, никакие данные не пересылаются.



Предположим, что по истечении 120 секунд с момента первоначального запроса браузер начал новый запрос этого ресурса

- сначала браузер проверяет локальный кэш и находит предыдущий ответ;
- поскольку прошло время экспирации, браузер не может его использовать;
- теперь браузер вынужден послать новый запрос, но если она все же не изменилась, то в этом нет необходимости
- используется хэш от содержимого файла, посылается как ETag;
- браузер посылает назад этот ETag при следующем запросе (в хэдере If-None-Match);
- если файл не изменился, возвращается ответ 304 Not Modified и тот же ETag возвращается назад;
- иначе пересылается новый файл и новый ETag

"max-age"

- указывает максимальное время в секундах, в течение которого можно переиспользовать полученный ответ с момента запроса. Например, "max-age=60" указывает, что можжно закэшировать ответ и переиспользовать его в течение ближайших 60 секунд.

"no-cache" и "no-store"

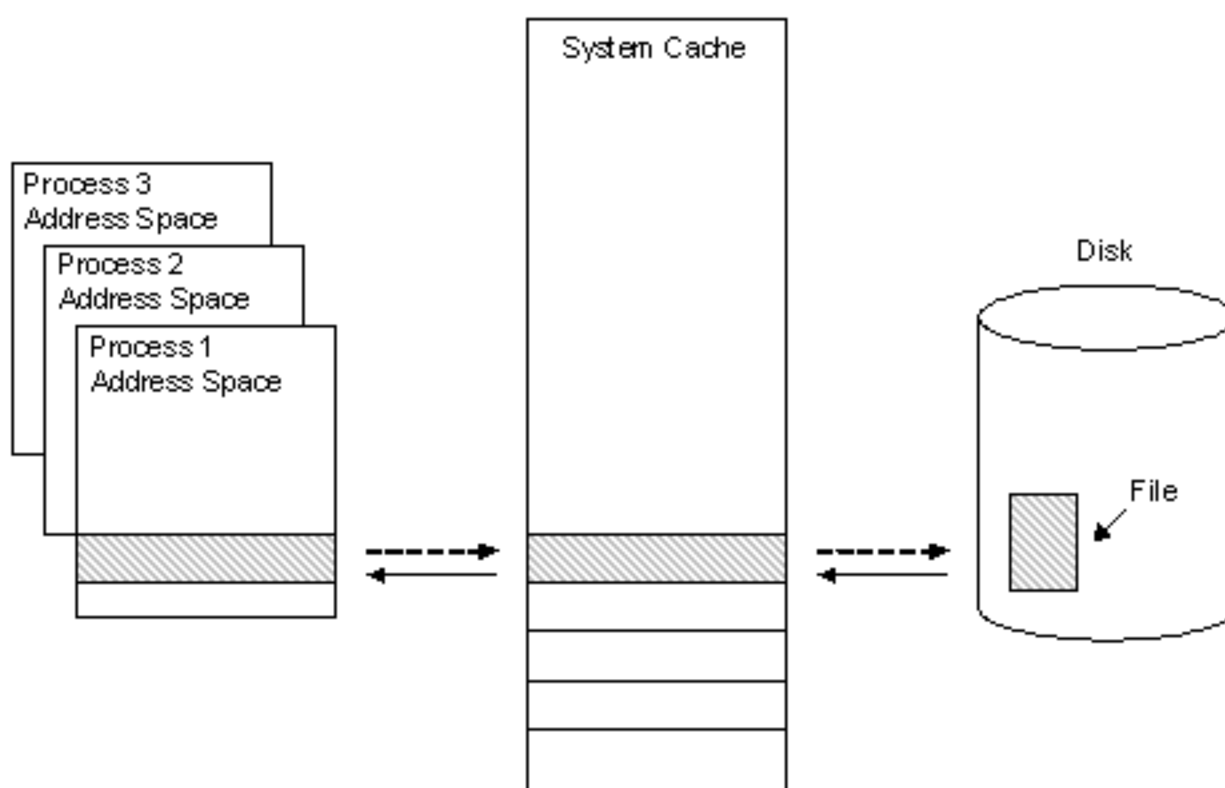
- "no-cache" указывает, что возвращенный ответ нельзя использовать для последующих ответов на тот же запрос без проверки на сервере, что ответ не изменился. Если присутствует ETag, произойдет обращение на сервер для валидации, но не будет скачки файла если ресурс не изменился.
- "no-store" запрещает хранить в кэше данный ответ — например, если он содержит чувствительные данные. Каждый раз при запросе этого ресурса даунлоадится полный ответ.

Пример - кэширование файла

Кэширование файла операционной системой — операции чтения по факту читают файл из системной памяти (system file cache), а не с физического диска.

```
- операции записи пишут в системный кэш, а не на диск.
- данные из системного кэша сбрасываются на диск с регулярными интервалам
и (cache flushing).
```

Такой тип кэша называется ***write-back cache***.



Пример - кэширование данных из БД

Можно кэшировать результат запросов из БД, но есть проблемы с инвалидацией кэша при масштабировании. Несколько бэкэндов. Несколько разных копий одних и тех же данных существуют одновременно в разных кэшах - проблема когерентности кэшей.

Что можно сделать - вопрос к аудитории?

Нужен протокол когерентности

- write update

Когда производится запись в один из кэшей, новые данные также записываются во все остальные кэши.

- write invalidate

Когда производится запись в один из кэшей, все остальные инвалидируются.

MySQL Query Cache

MySQL Query Cache: кэширует результаты SQL-запросов

- ищет входящие запросы, которые начинаются с "SEL", в хэш-таблице, и, если есть совпадение, возвращает результат от предыдущего исполнения того же запроса.

Deprecated с версии MySQL 8.0 Почему? Из-за многочисленных ограничений.

- запрос должен соответствовать байт в байт тому, что в ключе кэша (чтобы избежать парсинга запросов в кэше);
- использование недетерминированных фич приводит к тому, что запрос не кэшируется (включая временные таблицы, пользовательские переменные, RAND(), NOW() и User-defined functions);
- любые модификации любых таблиц, участвующих в каком-либо запросе, приводят к инвалидации ВСЕХ записей с участием этих таблиц в кэше;
- не масштабируется на многоядерные машины.

Лучше применять специальные СУБД для этого, например, Redis (речь ниже).

Caching servers

Memcached

- in-memory распределенная система для кэширования объектов;
- работает с парами "ключ-значение";
- может использоваться, например, для кэширования небольших объектов из СУБД.

In [ ]:

```

```

Redis

- in-memory хранилище данных;
- можно использовать как базу данных, кэш и message broker;
- в отличие от memcached поддерживает структуры данных: strings, hashes, lists, sets и т.д.

In [ ]:

```

```

Кэширование промежуточных результатов при сборке

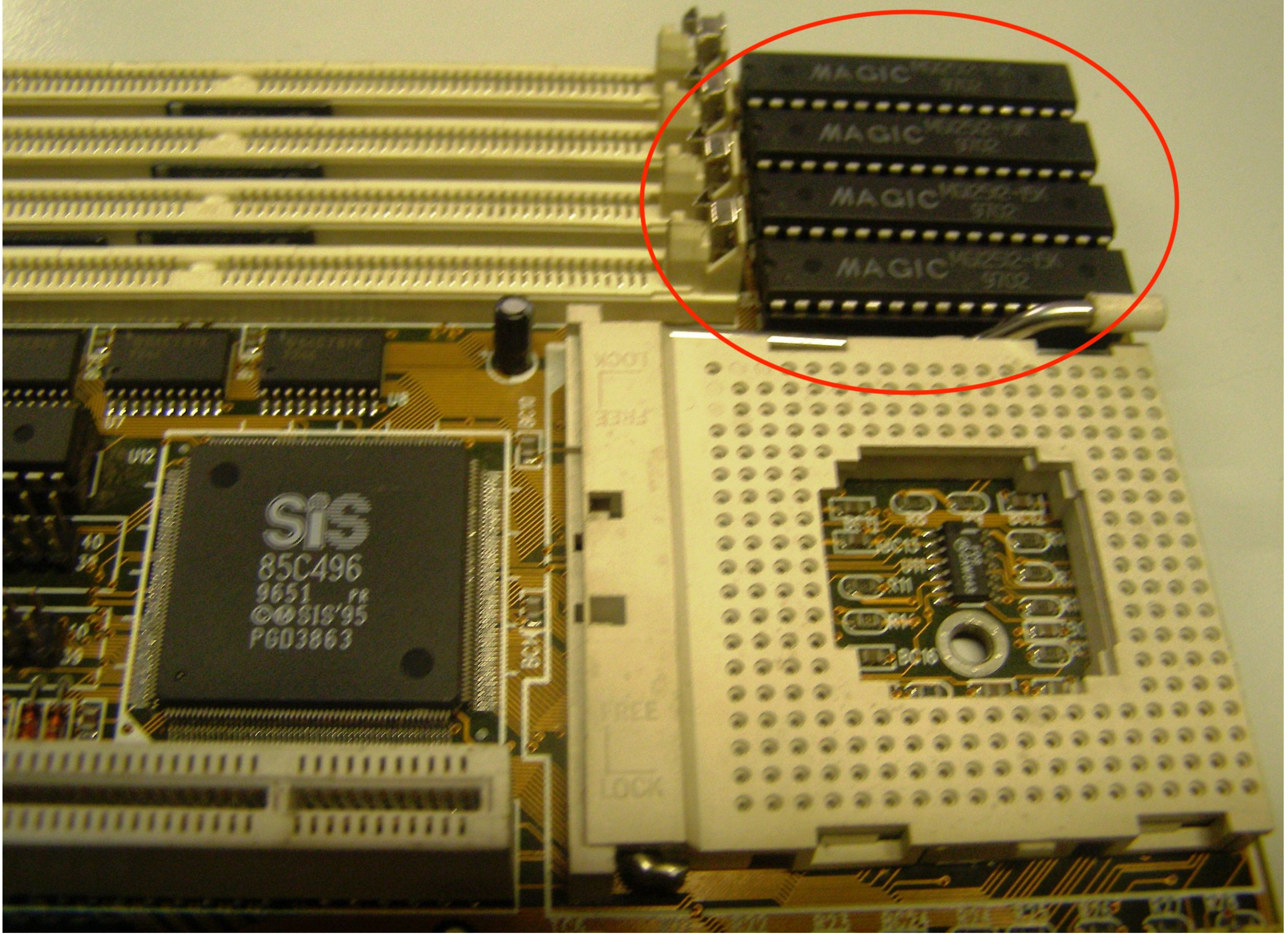
- precompiled headers

precompiled header - это C или C++ хэдер, компилируемый в промежуточную форму, которую компилятор обрабатывает быстрее. Использование precompiled headers может существенно сократить время компиляции, особенно если применяется к большим хэдерам, которые включают в себя множество других, или хэдеров, которые включаются во многие единицы трансляции.

- ccache

ccache на основе компилируемого файла, опций компилятора и выводимых им предупреждений/ошибок вычисляет хэш (по алгоритму md4) и сохраняет его, результат компиляции и все сообщения компилятора в папке кэша (~/.ccache). Если при вызове ccache оказывается, что этот файл уже компилировался (производится сравнение хэша), вместо реального вызова компилятора ccache выдает откомпилированный файл из своего кэша и при необходимости генерирует те же сообщения, которые в нормальных условиях выдал бы компилятор. Разумеется, проверка и выдача из кэша результата гораздо быстрее обычной компиляции.

Пример - процессорные кэши



Четыре микросхемы кэша второго уровня на материнской плате для процессоров семейства i486. Располагаются в буквальном смысле между ЦП и ОЗУ.

Аппаратное кеширование

- на заре компьютерных технологий доступ в память был лишь немного медленнее доступа к процессорным регистрам.
- со времени разрыв в производительности между процессорами и памятью стал нарастать. Память становилась узким местом при достижении полной производительности от системы.
- было решено использовать небольшую, но быструю кэш-память, для смягчения разрыва в производительности.

Общая иерархия памяти (всё является кешом для чего-то другого)

		Access time	Capacity	Managed By
On the datapath	Registers	1 cycle	1 KB	Software/Compiler
On chip	Level 1 Cache	2-4 cycles	32 KB	Hardware
	Level 2 Cache	10 cycles	256 KB	Hardware
	Level 3 Cache	40 cycles	10 MB	Hardware
Other chips	Main Memory	200 cycles	10 GB	Software/OS
	Flash Drive	10-100us	100 GB	Software/OS
Mechanical devices	Hard Disk	10ms	1 TB	Software/OS

Процессорные кэши

Так называемая "сверхоперативная память"

L1, L2 - внутри CPU, L3 - на материнской плате, либо тоже внутри CPU

Как они работают?

- процессор выставляет адрес, который он хочет прочитать/записать в кэш;
- два варианта:
  - cache hit - адрес есть в кэше, возвращаются соответствующие данные;
  - cache miss - адеса нет в кэше, делаем запрос в основную память, возвращаем результат процессору; кроме того, обновляем кэш (возможно, вытесняя какие-то другие данные);
- процессор должен понимать, что время выполнения его запроса может варьироваться.

Куда поместить кэш-память? До MMU или за MMU?

До MMU - кешируем виртуальные адреса, за MMU - кешируем физические адреса.

Ваши варианты?

До MMU:

Плюсы:

- отвечаем быстрее, потому что время MMU не учитывается;

Минусы:

- надо сбрасывать кэш при переключении контекста, потому что кешируем виртуальные адреса;

За MMU:

Плюсы:

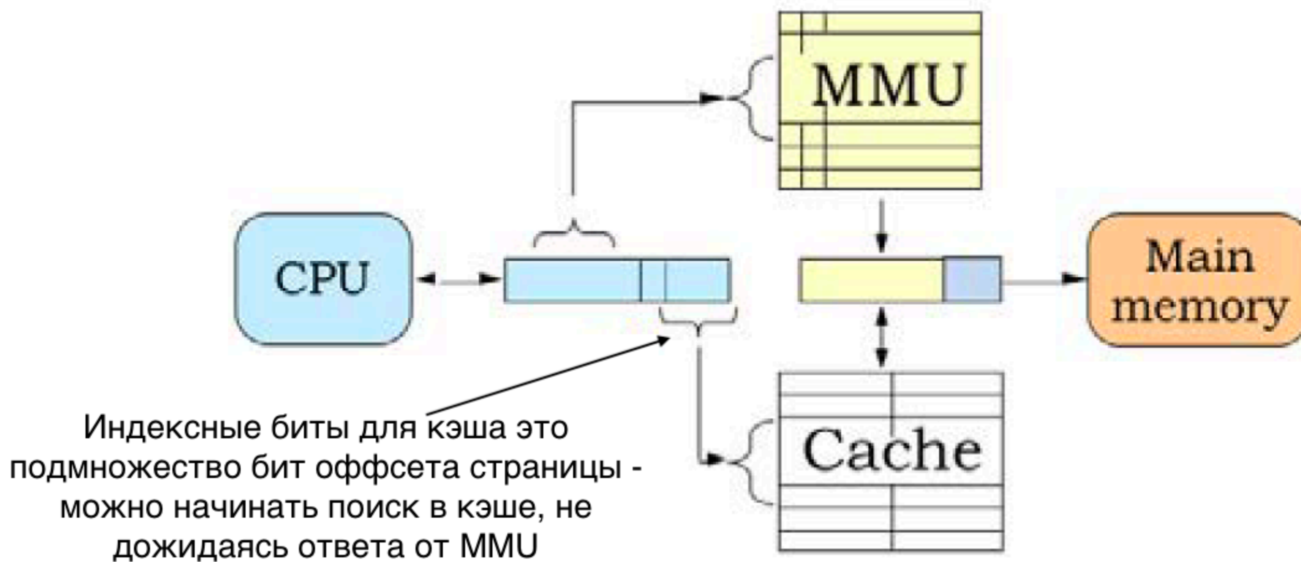
- не надо сбрасывать кэш при переключении контекста, избегаем неактуальных данных;

Минусы:

- медленнее, потому что есть время MMU.

Кэши: оптимальный вариант, берем лучшее из двух подходов

Комбинация двух вариантов: перекрывающаяся адресация



Каждый виртуальный адрес (оффсет страницы) делится на 3 части:

- собственно номер виртуальной страницы;
- индекс массива в кэше;
- индекс в кэш-линии.

Не ждём окончания работы MMU, сразу начинаем запрашивать данные из кеша. Кэш индексируется виртуальными адресами, а тэгируется физическими. В то время, как MMU использует номер виртуальной страницы, чтобы получить номер физической страницы, кэш использует индексную часть, чтобы найти в кэше адрес. Идет параллельный lookup из кэша и трансляция виртуального адреса в физический через MMU. Затем сравнивается физический адрес от MMU физический тэг от кэша. Если они совпадают, был cache hit. Если нет, то cache miss. Тогда возвращается результат из памяти и записывается в кэш.

Достоинства:

- работает быстро, потому что lookup из кэша идет сразу же, и его не надо сбрасывать при переключении контекстов.

Недостаток:

- чтобы увеличить размер кэша, надо увеличивать к-во бит в в виртуальном адресе.

Стратегии выталкивания

- часто применяются варианты LRU

Стратегии записи в кэш

Write-through

- основная память апдейтится каждый раз, когда CPU пишет в кэш

Преимущество:

- основная память всегда содержит то же, что и кэш, поэтому при наличии нескольких CPU локальные кэши других CPU могут обновляться сразу, следя за трафиком основной памяти

Недостатки:

- замедляет запись
- большой трафик

Write-back

- изначально пишем только в кэш
- обновленные участки кэш-памяти помечаются битом обновления и время от времени сбрасываются в основную память
- если блок надо заменить, удалив из кэша, пишем его только если установлен бит обновления

Недостатки:

- кэши других CPU out of sync
- устройства I/O должны обращаться через кэш к главной памяти
- фактически 15% обращений к памяти - это записи

Наиболее часто встречаемая политика записи - это write-back.

Размер линии кэша

- Получаем не только необходимое слово, но и соседние слова
- если увеличить размер блока, увеличится hit ratio - принцип локальности
- если еще увеличить, hit ratio уменьшится - вероятность использования новой информации становится меньше, чем вероятность переиспользования замененной Большие блоки:
- уменьшают к-во блоков, которые размещаются в кэше;
- данные перезаписываются вскоре после того, как были получены из кэша;
- каждое добавочное слово менее локально, то есть вероятность, что оно нужно, уменьшается Не найдено оптимальное значения блока Разумно - от 8 до 64 байт

Многоуровневые кэши

- нужны для разгрузки системной шины
- L1 в CPU, L2 - может быть вне CPU в SRAM
- L1, L2 в CPU, L3 - вне или в CPU

Эволюция кэшей в процессорах Intel

Проблема	Решение	Процессор, в котором фича впервые применена
Внешняя память медленнее, чем системная шина	Добавили внешний кэш с использованием более быстрой технологии памяти	i386
Увеличилась скорость CPU, внешняя шина стала узким горлышком для доступа к кэшу	Внешний кэш перевели в CPU, теперь он работает с частотой процессора	i486
Внутренний кэш становится слишком мал ввиду ограниченного места в чипе CPU	Добавился внешний кэш L2, более быстрый, чнм основная память	i486
Instruction Prefetcher и Execution Unit конкурируют за доступ в кэш - один ждет, пока отработает другой	Сделали отдельные кэши для инструкций и для данных	Pentium
Снова увеличилась скорость CPU, так что теперь L2 стал узким местом	L2 перенесен в CPU	Pentium II
Некоторые приложения работают с массивными базами данных и должны иметь быстрый доступ к большим объемам данных. Внутренние кэши слишком маленькие	Добавили внешний кэш L3	Pentium III
-	Перенесли L3 на кристалл	Pentium 4

Cache-oblivious algorithm

Кэш-oblivious (буквально "забывающий о кэше") или cache-transcendent ("работающий поверх кэша") алгоритм - это алгоритм, учитывающий особенности работы CPU-кэша и пользующийся его преимуществами. Оптимальный cache-oblivious алгоритм это cache-oblivious algorithm, который использует процессорный кэш оптимально.

In [1]:

```
vectors filled, data (MB): 512; index (MB): 512; total (MB): 1024
regular access: 62443 ms passed
reverse access: 62469 ms passed
random access: 77731 ms passed
72060347100154493, 72060347100154493, 72060347100154493
```

Выводы

- если данных много, лучше читать из памяти линейно
- если линейно читать не удается, лучше иметь мало данных, чтобы рабочий working set влезал хотя бы в кэш L3 (лучше L2 или L1).

Пример: кэширование результатов функций

- кэшировать можно только "чистые" функции

"Чистая функция" - это функция, которая:

- является детерминированной;
- не обладает побочными эффектами.

Детерминированная функция

Ф-ция является детерминированной, если для одного и того же набора входных значений она возвращает одинаковый результат. В вычислении не участвуют глобальные переменные.

Недетерминированность функции — возможность возвращения функцией разных значений несмотря на то, что ей передаются на вход одинаковые значения входных аргументов.

- невозможно построить однозначную таблицу значений функции;

Побочные эффекты

- модификация значений глобальных переменных;
- осуществление операций ввода-вывода;
- выбрасывать исключения.

Частичная защита в C++ - модификатор const у ф-ций-членов классов. Если constexpr - результат ф-ции детерминирован уже на этапе компиляции. При подаче в нее constexpr выражения, известного на этапе компиляции, получаем constexpr-выражение, также известное на этапе компиляции.

In [ ]:

```

```

In [ ]:

```

```

Практика

Кэширующие библиотеки для python-а

python - functools.lru_cache, cachetools, cacheout

pip install cacheout

pip install cachetools

In [4]:

In [2]:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1
597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 19
6418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887]
'fibo' ((), {}) 4.57 sec
```

In [4]:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1
597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 19
6418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887]
'fibo' ((), {}) 0.00 sec
CacheInfo(hits=66, misses=35, maxsize=None, currsize=35)
```

In [5]:

```
1
2
1
2
1
2
1
2
1
2
'work' ((), {}) 10.04 sec
```

In [6]:

```
1
2
1
2
1
2
1
2
1
2
'work' ((), {}) 2.01 sec
```

Out[6]:

```
CacheInfo(hits=8, misses=2, maxsize=128, currsize=2)
```

In [ ]:

In [7]:

```
1
2
1
2
1
2
1
2
1
2
'work' ((), {}) 2.01 sec
```

In [2]:

Вопрос: что не так с этой реализацией? Как ее исправить?

In [1]:

```

```

In [13]:

```

```

```
'verify' ((<__main__.LRUCache object at 0x107f3b9e8>,), {}) 2.35 sec
'verify' ((<__main__.LRUCacheFixed object at 0x107b407f0>,), {}) 0.1
5 sec
```

Кэширующие библиотеки для Java

- org.apache.commons.collections.map.LRUMap;
- Spring Framework - кэширование методов с помощью аннотаций;
- Google Guava.

In [ ]:

```

```

Кэширование в многопоточном окружении

Кэши и подходы, которые мы рассмотрели, как правило не предназначены для многопоточного использования и требуют отдельных усилий для обеспечения многотопочной работы.

- можно использовать мьютексы (производительно ухудшится);
- можно использовать lock-free подход: single writer и multiple readers.

Домашнее задание - написать свою реализацию скользящего timed-кэша на вашем языке программирования (C/C++/Java/Python)

Цели:

- лучше понять, как устроен один из видов кэша - скользящий timed cache.

Задача:

- сделать свой скользящий timed cache, реализовав функции get, set.
- кэш должен параметризоваться размером и таймаутом - временем, прошедшим с момента последнего доступа к объекту, по истечении которого объект вычищается из кэша.

Требования к реализации:

- в кэш приходят случайные объекты, в любом объеме и с любой частотой;
- стоит уделить время производительности кэша, чтобы он отвечал на запросы так быстро, как только возможно;
- должны присутствовать тесты, показывающие правильность работы кэша;
- (опционально) написать тесты производительности, меряющие производительность вашего кэша и сравнивающие его с этой же (библиотечной) и/или иными реализациями кэша в вашем ЯП (если есть).

Заполните опросник в конце занятия:
https://docs.google.com/forms/d/e/1FAIpQLSfGj4BY8YxbIJ9SXVmqBOvpmkN1BZGb2LfldaV3y8ZLUK4BFQ
(https://docs.google.com/forms/d/e/1FAIpQLSfGj4BY8YxbIJ9SXVmqBOvpmkN1BZGb2LfldaV3y8ZLUK4BFQ

Полезные ссылки и литература

Математика

- Some mathematical facts about optimal cache replacement. Pierre Michaud. https://hal.inria.fr/hal-01411156v2/document (https://hal.inria.fr/hal-01411156v2/document)

Архитектура компьютерных систем

- The Cache Memory Book (The Morgan Kaufmann Series in Computer Architecture and Design).

О кэшах процессора

- О кэшах процессора: https://lwn.net/Articles/252125/ (https://lwn.net/Articles/252125/) - "What every programmer should know about memory". Ulrich Drepper. Part 2. CPU Caches. http://rus-linux.net/MyLDP/hard/memory/memory-03-01.html (http://rus-linux.net/MyLDP/hard/memory/memory-03-01.html) - русский перевод.
- Написание cache-oblivious приложений - https://lwn.net/Articles/255364/ (https://lwn.net/Articles/255364/) - "What every programmer should know about memory". Ulrich Drepper. Memory part 5: What programmers can do. http://rus-linux.net/lib.php?name=/MyLDP/hard/memory/memory-6-1.html (http://rus-linux.net/lib.php?name=/MyLDP/hard/memory/memory-6-1.html) - Что могут делать программисты - оптимизация кэша - русский перевод.
- О кэшах процессора в контексте оптимизации программ - доклад Андрея Аксенова "Низкоуровневая оптимизация C/C++" на HighLoad 2011: http://profyclub.ru/docs/146 (http://profyclub.ru/docs/146)

memcached, redis

- Redis in Action. Josiah L Carlson.

OTUS

Спасибо
за внимание!