БАЗЫ ДАННЫХ

Лекция 2

ПЛАН НА СЕГОДНЯ

- Общий подход к внешней памяти
- Linux internals 101 - в том, что важно для нас
- Схема устройства реляционной СУБД
- Путь чтения
- Алгоритмы кеширования

ОСОБЕННОСТИ ВНЕШНЕЙ ПАМЯТИ

- Есть разные типы носителей
- В первом приближении HDD и SSD
- В ноутбуках HDD встречаются все реже
- В облачных хранилищах - реальный вариант
- Почему так - см. цены на аренду

РАЗБЕРЕМСЯ С УСТРОЙСТВАМИ

РАЗБЕРЕМСЯ С УСТРОЙСТВАМИ

- В SSD физические принципы другие
- И все сильно лучше, чем в HDD
- Но принципиально проблемы сохраняются
- Две проблемы: долгое время операции и отдельные издержки на начало операции

РАЗБЕРЕМСЯ С УСТРОЙСТВАМИ

- Стандартное решение - кеширование
- Если бы проблема была только в долгой операции, кешировали бы единицу обмена ("запись")
- Задержка на операцию располагает к тому, чтобы кешировать больше, чем запись
- Вопрос: сколько ?

РАЗМЕР ЕДИНИЦЫ КЕШИРОВАНИЯ

РАЗМЕР ЕДИНИЦЫ КЕШИРОВАНИЯ

- Пусть у нас 16G памяти и 2 файла
- Тогда берем буферы по 8G - хуже не будет
- А если у нас 1G памяти и 10000 файлов
- Тогда дилемма: по 100K на файл и 10000-merge
- Или в два прохода

РАЗМЕР ЕДИНИЦЫ КЕШИРОВАНИЯ

- Сначала по 10M на файл и 100-merge
- И так 100 раз
- И потом еще один такой проход
- В два раза больше потоковых чтений, но меньше latency

ЧТО НАМ ТУТ ВАЖНО

- Нам важны не конкретные прикидки для конкретной задачи
- Важен общий ответ на вопрос "какие должны быть буферы по размеру"

подход

- Для потоковых задач можно поставить вопрос так: пусть задержка будет сопоставима с временем чтения
- Тогда издержки на latency будут похожи на разумную ассимптотическую константу

ПРИБРОСИМ

- На HDD задержка порядка 5-10ms, скорости от сотни мегабайт в секунду
- Время позиционирования - примерно время передачи 1 MB
- В SSD время короче, но соотношение примерно такое же
- Получается, что буфер от 1 mb - нормально, если меньше - то задержка значима

РАЗМЕР СТРАНИЦ В СУБД

- Но в традиционных СУБД размер страницы сильно меньше
- Где-в районе 4-8К
- Потому что обработка непотоковая
- И большие блоки будут тратить ресурс кеша
- Сложившаяся устойчивая практика - компромисс между крайностями

ОС И ФАЙЛОВЫЕ СИСТЕМЫ

- Чисто технически СУБД может работать напрямую с диском
- Но на практике это делается через операционную/файловую систему
- У СУБД часто есть свое кеширование
- У ОС - свое файловое кеширование

ОС И ФАЙЛОВЫЕ СИСТЕМЫ

- А зачем нужно свое кеширование ?
- А ничего, что и ОС, и СУБД кешируют ?
- И есть другие процессы, и mmap, и swap и память не резиновая

LINUX INTERNALS 101

- Хотим понять (уточнить/восстановить)
  - Как ОС кеширует дисковый обмен
  - Как настраивать размер файлового ОС кеша
  - Что такое swap

LINUX INTERNALS 101

- Хотим понять (уточнить/восстановить)
  - Как работают read/write в простейшем варианте
  - Что такое ext4 и xfs и зачем про это знать в контексте СУБД
  - Полезные в контексте СУБД инструменты ОС-уровня

НАЧНЕМ С RAM

- Зачем это нам сейчас
  - Кеширование требует памяти
  - Память - ограниченный ресурс
  - Надо понимать, с кем и как конкурируем или взаимодействуем

НАЧНЕМ С RAM

- В процессоре x86 сегментно-страничная организация памяти
- Сегментная часть нам здесь неинтересна
- Страничная предполагает преобразование адресных пространств
- При исполнении команд, работающих с памятью на шину подается адрес
- И просто при исполнении команд

АДРЕСНЫЕ ПРОСТРАНСТВА

- Физический адрес - номер ячейки
- Он не равен тому адресу, который формируется в инструкции
- И который мы можем распечатать через printf("%p\n", &v);
- Поймем, как они соотносятся

АДРЕСНЫЕ ПРОСТРАНСТВА

- Разберем искуственно упрощенный пример
- Можно разделить адрес на две части: последние 12 бит и все остальное
- Все остальное будем считать адресом в некой таблице
- Таблица - массив структур

ПРИМЕР

- Адрес 0x12345678
- Последние 12 бит - 0x678
- Все остальное - 0x12345
- Идем в некую таблицу по индексу 0x12345

ОПИСАНИЕ СТРУКТУРЫ

- Одно из полей - базовый физический адрес страницы
- Другие поля - флаги с правами доступа
- При обращении к памяти пойдем в таблицу
- Узнаем, есть ли у нас права на доступ

ОПИСАНИЕ СТРУКТУРЫ

- Если есть, возьмем базовый адрес страницы
- Прибавим смещение (последние 12 бит исходного адреса)
- Получим физический адрес
- И это все делается аппаратно
- И отдельно аппаратно кешируется

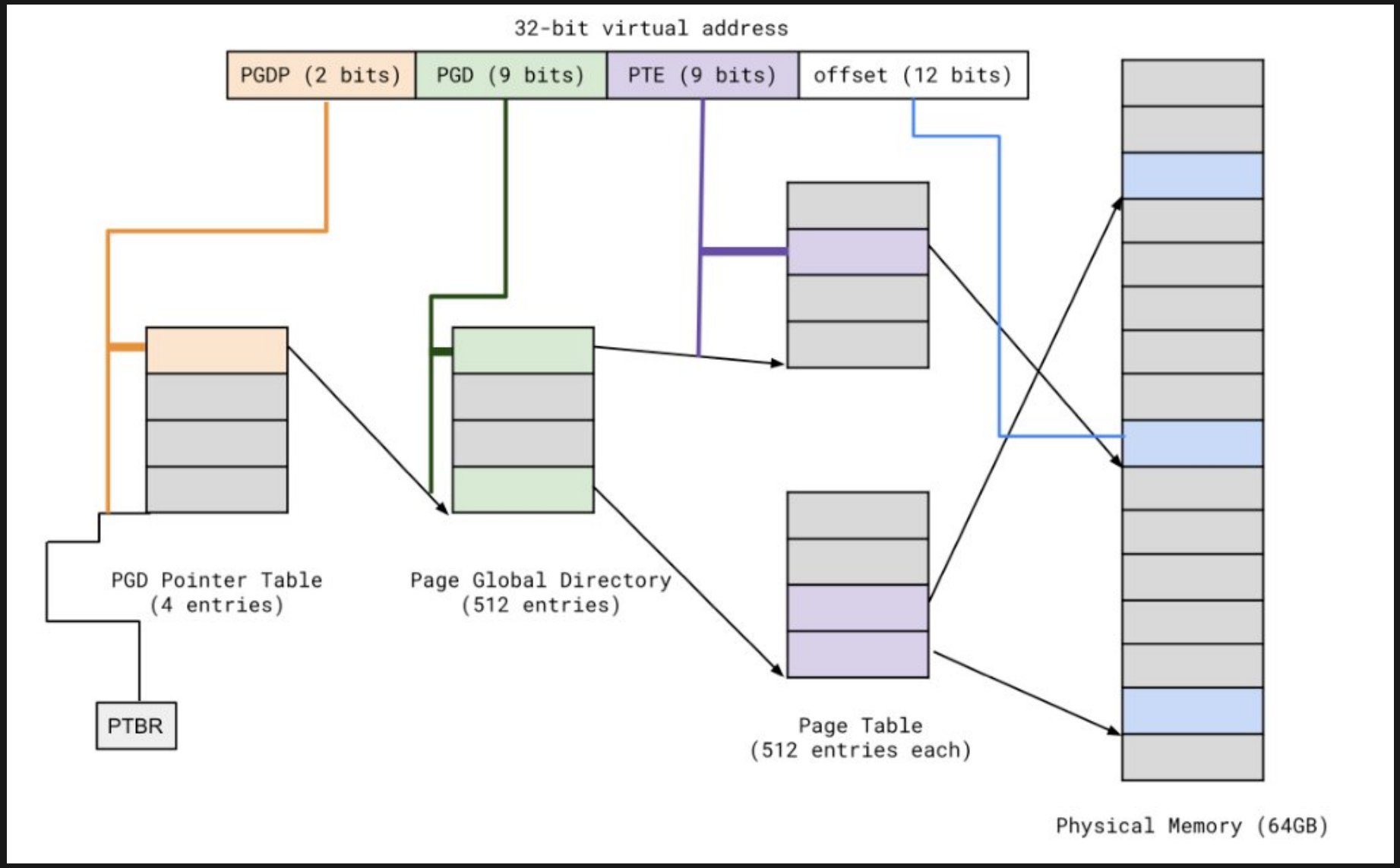
ПРОДОЛЖИМ ПРИМЕР

- В таблице по смещению 0x12345 лежит структура
- По ее данных есть права на чтение, но нет на запись
- Физический адрес - 0x54321
- При чтении будет физический адрес 0x54321678
- А при записи получим аппаратное исключение

БЛИЖЕ К ЖИЗНИ

- Можно сделать многоступенчатый процесс
- Возьмем кусочек адреса
- И он будет индексом в одной таблице
- Из нее узнаем адрес другой таблицы
- А позицией в ней будет другой кусочек исходного адреса

КАРТИНКА


32-bit virtual address
PGDP (2 bits)
PGD (9 bits)
PTE (9 bits)
offset (12 bits)
PGD Pointer Table
(4 entries)
Page Global Directory
(512 entries)
Page Table
(512 entries each)
PTBR
Physical Memory (64GB)

ЕСЛИ ХОЧЕТСЯ ПОДРОБНОСТЕЙ

- (Картинка отсюда) https://cs4118.github.io/www/2023-1/lect/18-x86-paging.html
- https://wiki.osdev.org/Paging
- Документация Intel

ПРИМЕНЕНИЕ В ОС

- Адрес первой таблицы хранится в специальнои регистре процессора
- Этот регистр - часть контекста процесса
- Задача ОС - формировать такие таблицы для процессов

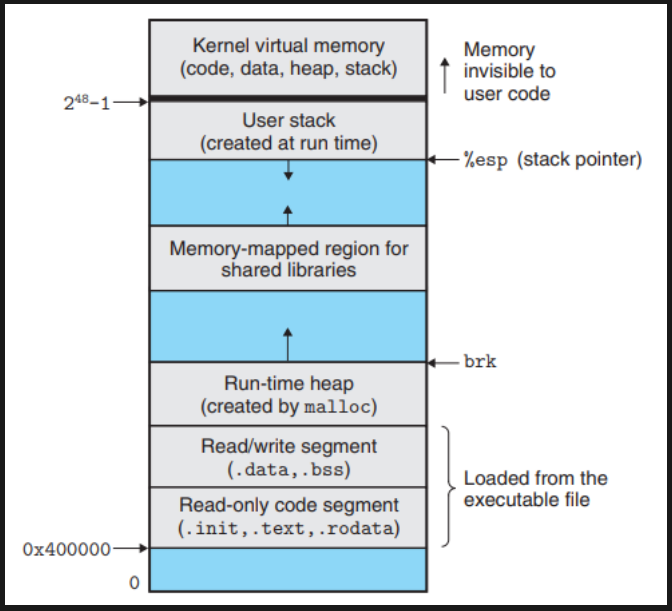
РАСПРЕДЕЛЕНИЕ ПАМЯТИ В LINUX

- Ядро выделяет какую-то память для своих структур
- При каждом старте процесса ядро знает, сколько памяти нужно на код и статику
- В правильных приложениях используются разделяемые библиотеки
- Два старта одного приложения могут использовать одну физическую копию собственного кода

НО ЕСТЬ ЕЩЕ СТЕК

- У каждого процесса свой стек
- Под это выделяются страницы физической памяти
- В своем адресном пространстве процесс видит сплошные области кода, данных, стека
- ОС соответсвенно настраивает таблицы страниц процесса

КАРТИНКА



Источник:

https://stackoverflow.com/questions/63770603/how-does-each-processs-private-address-space-gets-maped-to-physical-address

ПРО КУЧУ

- Стандартный malloc при первом запросе просит память у ОС (brk/sbrk)
- ОС пытается ее выделить и включает в адресное пространство процесса
- В дальнейшем malloc смотрит, нет в ли в куче процесса памяти
- И снова вызывает, когда памяти не хватает

ОСВОБОЖДЕНИЕ ПАМЯТИ

- Логично от free ожидать чего-то зеркального
- Возвращать в ОС запрошенную память
- Но есть проблема - sbrk работает по стековому принципу
- Если хоть байт не освобожен где-то "вверху", он блокирует возврат памяти в ОС

ОСВОБОЖДЕНИЕ ПАМЯТИ

- Есть другой механизм - "анонимный mmap"
- Он гибче выбирает место в адресном пространстве процесса
- Оборотная сторона - усложняется реализация malloc
- И при мелких malloc-ах сохраняется риск удержания памяти после free

КАК РАБОТАЕТ MALLOC

- Сочетает brk и mmap
- Ориентируется на размер
- Порог - где-то 128-256K

ПРАКТИЧЕСКИЙ СЦЕНАРИЙ

- Приложение ест много памяти
- Идем в приложение, что-то в нем закрываем
- А приложение все равно ест много памяти
- Вариант выхода - более крупные malloc-и
- Здесь и сейчас - убить приложение

КАК РАБОТАЕТ ФАЙЛОВЫЙ КЕШ В LINUX

- Файловые чтение кешируются блоками 4-8K
- Бывает prefetching
- Принцип Linux: незачем держать свободную память, если не отдать под файловый кеш
- Вся неиспользуемая процессами память используется под кеш

КАК РАБОТАЕТ ФАЙЛОВЫЙ КЕШ В LINUX

- Если запускается новый процесс, а память занята кешом - тогда что-то вытесним
- Нет конфиг-параметра "размер файлового кеша"
- Единственное влияние - экономить память и дать ему расти
- Есть вариант заказать прямое чтение

ЧТО ТАКОЕ SWAP

- У вас работает браузер со вкладками
- Какие-то вы оставили, чтобы глянуть попозже и забыли
- Память выделена, но не используется
- Или много кода, но не все фичи используются
- И когда память кончилась, почему обязательно из файлового кеша вытеснять страницу ?

ЧТО ТАКОЕ SWAP

- ОС может понять, что какая-то страничка процесса давно не использовалась
- Сохранить ее на диске
- А если это код, то и сохранять отдельно не обязательно
- Или она не менялась с последней загрузки
- А место в памяти передать нуждающимся

ЧТО ТАКОЕ SWAP

- А когда к ней обратились, ее подгрузить
- И кого-то другого вытеснить
- Файловые кеш и swap - про одну проблему с разных ракурсов
- И это один механизм

ОБЩИЙ МЕХАНИЗМ

- При подкачке из swap-а ищем, кого бы вытеснить
- Жертвой может быть и страничка файловаго кеша, а можем кого-то свопнуть
- Аналогично при кешировании
- Есть еще третий игрок - mmap

ФАЙЛОВЫЕ СИСТЕМЫ

- Организация данных на диске может быть разной
- С разными приоритетами
- Есть понятие "файловая система"
- В Linux есть абстракция VFS и механизм монтирования
- И есть dev-файлы - можно читать диск "как есть"

МОНТИРОВАНИЕ

- Каждая физическая файловая система имеет свое иерархическое устройство
- Иерархия загрузочного диска становится основой VFS
- Можно создать пустой каталог и "привязать" его к другому диску (смонтировать)
- Тогда каталог перестает быть пустым
- И в нем видим содержимое корня смонтированного диска

ОТКРЫТИЕ ФАЙЛА

- При открытии файла Linux опеределяет, на какой физической файловой системе он находится
- При последующих операциях он знает, какие функции вызывать
- read на разных файловых системах будет работать по-разному
- write - еще в большей степени по-разному

ЧТЕНИЕ ФАЙЛА

- Linux проверяет, нет ли нашего блока в кеше
- Если нет - вызывает callback, специфичный для файловой системы
- Файловая система определяет, какие метаданные надо прочитать
- И куда пойти, чтобы реально читать
- А кеш - общий

ЗАПИСЬ ФАЙЛА

- Запись - это часто обновление метаданных
- Возможны проблемы с выключением питания
- Разные файловые системы решают их по-разному
- Запись может быть отложенной

РАЗЛИЧИЯ ФАЙЛОВЫХ СИСТЕМ

- Кто-то ведет свое журналирование
- Кто-то ведет журналирует метаданные, кто-то - все
- Политика выделения блоков может отличаться
- Разные лимиты на размеры файлов, количество файлов в каталоге

ТИПИЧНЫЕ ФАЙЛОВЫЕ СИСТЕМЫ

- ext2 - старенький базовый вариант файловой системы для Linux
- ext4 - современный вариант, журналирование возможно
- fat16/fat32 - старая с Windows, не надо на тей ставить postgres

ТИПИЧНЫЕ ФАЙЛОВЫЕ СИСТЕМЫ

- ntfs - "хорошая" с Windows, хороший вариант, если вы целиком на Windows
- xfs - от RedHat с акцентом на большие объемы и много файлов

ЧТО ПОЛЕЗНО ДЛЯ НАБЛЮДЕНИЯ ЗА БД

- procfs/sysfs
- Модули ядра - в самых разных смыслах
- smem, free
  strace/ptrace
- ps, lsof
- /ust/share/man/man2

СХЕМА РАБОТЫ РЕЛЯЦИОННОЙ СУБД

- Первый вопрос
  - Через файловую систему или напрямую ?

НАПРЯМУЮ VS ЧЕРЕЗ ФАЙЛОВУЮ СИСТЕМУ

- Теоретически можно напрямую
- Реально - выигрыш не гарантирован
- Издержки большие

ЧТО С КЕШИРОВАНИЕМ ?

- Варианты
  - Делегируем файловой системе
  - Двойное кеширование
  - Только свое

ЗАБАВНЫЕ СЦЕНАРИИ

- Свое кеширование не гарантирует, что не пойдем на диск
- Даже если страница лежит в кеше
- Возможная причина: swap

ДЕЛЕГИРОВАНИЕ ОС

- Плюс: не умножаем сущности
- Минус: сильно зависимы от общей нагрузки
- И от ОС

СВОЙ КЕШ

- Точно знаем размер кеша, на который расчитываем
- По модулю swap-а
- При двойном можем получить "вторичное ускорение"

СВОЙ КЕШ

- За счет того, что в наш кеш не попало, а в файловый - попало
- Минус двойного: меньше памяти оставляем другим

ОБЩАЯ НАСТРАИВАЕМОСТЬ

- Вопросов много
- Не все ответы очевидны
- Возможный выход - гибкость настройки, модульность, плагины

СХЕМА РАБОТЫ РЕЛЯЦИОННОЙ СУБД

- Таблица организуется постранично
- Страница - 4K или 8K
- В ней живут метаданные - где какая запись начинается
- Технически запись может определяться смещением на странице

СХЕМА РАБОТЫ РЕЛЯЦИОННОЙ СУБД

- Удобно внутристраничную адресацию "спрятать"
- Тогда запись - номер страницы и номер записи внутри страницы
- Позволяет реорганизаваться внутри страницы и не обнулять индексы
- Postgres: select ctid, * from t;
- Есть аналог в MS SQL, в MySQL - нет

СХЕМА РАБОТЫ РЕЛЯЦИОННОЙ СУБД

- Нужны метаданные: данные о таблицах
- Для каждой таблицы знаем ее файл
- Количество записей, количество страниц
- Удобно хранить на общих основаниях

ПРОСТЕЙШИЙ ПУТЬ ЧТЕНИЯ

- Разумные по структуре таблицы
- И пока - влезающие в память по размеру
- Уже созданы и заполнены
- Делаем select на холодной базе
- По таблице определяем ее файл и читаем постранично

ПРОСТЕЙШИЙ ПУТЬ ЧТЕНИЯ

- Изначально кеш пустой
- Каждую страничку кладем в кеш
- В общем случае - проверяем, нет ли странички в кеше
- Если есть, берем из кеша

ВЫТЕСНЕНИЕ

- Если таблиц много, когда-то месте в кеше закончится
- Надо кого-то вытеснить
- Пока мы не думаем про запись
- И вытеснить - это просто переписать и пометить, что эта страница больше не в кеше

АЛГОРИТМ БЕЛАДИ

- Теоретический алгоритм
- Пусть у нас есть знание о последовательности предстоящих запросов в кеш
- Будем вытестнять страницу, которая позже всех понадобится в будущем
- Он используется для оценок других алгоритмов

ПРАКТИЧЕСКИЕ АЛГОРИТМЫ

- Требуется эффективность в смысле хорошего уровня попаданий
- И "дешевизна" вспомогательных структур
- Хороший вариант: LRU
- Дешево и эффективно

АЛГОРИТМ LRU

- Теоретический факт: на памяти в константу больше количество промахов не более, чем в константу больше
- И реализовать несложно
- Очень популярен
- Но есть к нему две претензии

АЛГОРИТМ LRU

- Претензия 1: не очень удобен для параллельной работы
- Обновление - неприятный hotspot
- Претензия 2: вымывание кеша на длинных SELECT-ах
- Нужна какая-то градация, повторяемость на малом интервале как-то учитывать
- И дешево в реализации

АЛЬТЕРНАТИВЫ LRU

- Иногда используют какие-то эвристики
- Развитый вариант эвристики - алгоритм LIRS
- Общая идея LIRS: на основе LRU среди "недавних" страниц вводится два класса
- LIR - более "заслуженные"
- HIR - кандидаты в LIR

АЛЬТЕРНАТИВЫ LRU

- Другая альтенатива - Clock
- Заводим счетчик для каждой страницы в кеше
- На каждое попадание увеличиваем
- Когда ищем жертву, перебираем счетчик

АЛЬТЕРНАТИВЫ LRU

- Уменьшаем счетчик на 1 и сравниваем с 0
- Если равен 0, это жертва
- Если больше 0, идем к следующему счетчику
- Следующий поиск начинаем там, где остановились - как часовая стрелка

ОСМЫСЛИМ CLOCK

- Возникает вопрос: а не будем бегать кругами ?
- Начнем с того, что это не очень реальный расклад
- Но даже если так, это даже прекрасно с точки зрения пропускной способности
- Каждая единичка в счетчике - это уже состоявшееся попадание в кеш
- И польза от него выше, чем стоимость декремента

ОСМЫСЛИМ CLOCK

- Для времени ответа это может стать проблемой
- Решается ограничением на счетчик
- Postgres использует Clock с ограничением на счетчик
- MySQL - в зависимости от engine LIRS или LRU со своими эвристиками
- sqlite - чистый LRU