**Лекция 18**

Измерим время работы такой программы:

#include <iostream>  
  
const int N = 64 \* 1024 \* 1024;  
int a[N];  
  
int main() {  
 for (int k = 0; k < 10; ++k) {  
 for (int i = 0; i < N; ++i) {  
 a[i] \*= 3;  
 }  
 }  
}

**time ./a.out** — с помощью такой команды

real 0m0,634s

user 0m0,526s

sys 0m0,109s

Теперь вместо ++i будем делать i += 3:

real 0m0,389s

user 0m0,284s

sys 0m0,106s

Стало примерно в 2 раза быстрее. Если теперь сделать i += 10:

real 0m0,301s

user 0m0,191s

sys 0m0,110s

i += 32:

real 0m0,245s

user 0m0,139s

sys 0m0,106s

Время работы меняется нелинейно. Время работы зависит не только от количества операций, но и от локальности этих операций. Если обращаемся к каждой ячейке памяти подряд, это происходит не сильно медленнее, чем если бы мы обращались к каждой второй ячейке памяти. Обращение к каждой 10 ячейке по времени примерно равно обращению к каждой 15 ячейке. Все потому, что основное время тратится не на выполнение операций, а на загрузку новых областей памяти в кэш процессора.

<https://igoro.com/archive/gallery-of-processor-cache-effects/> - продемонстрированная на лекции статья

**Кэш локальность** — свойство алгоритмов обращаться преимущественно к данным,

расположенным в памяти последовательно и рядом.

То есть если рандомно брать элементы из разных кусков массива, проходя его не подряд, работа замедляется. Процессору гораздо удобнее работать, если обращение к памяти происходит подряд, а не по одной ячейке за раз из разных частей массива.

MergeSort — кэш локальная сортировка, а HeapSort — нет.

У процессора помимо регистров есть также **кэш память**.

У CPU есть MMU с TLB cache. Но также есть кэш не только для страниц, но и для обычной памяти.

Оперативная память огромна, до нее ходить далеко. Те переменные, с которыми он непосредственно работает, лежат в регистрах. Помимо них, рядом с процессором есть кэши разных уровней:

* **L1 cache** ~ 32 kB - 128 kB меньше и быстрее
* **L2 cache** ~ 1 MB чуть побольше чуть помедленнее
* **L3 cache** ~ 8 MB еще больше и еще медленнее

Если в компьютере несколько ядер процессора, то обычно первый и второй уровни кэша приватные для каждого ядра, а третий кэш общий для всех ядер, и он самый большой.

TLB cache находится перед этими кэшами.

Когда обращаемся к памяти, мы сначала идем в TLB cache, смотрим, к какому реальному адресу памяти нам нужно обратиться, далее проверяем, нет ли этого адреса в кэше. Последовательно проверяем L1, L2, L3 — если нигде его нет, идем в оперативную память. У каждого кэша есть своя аппаратная мапка, по которой он быстро проверяет, есть страница в кэше или нет.

Как только считали какую-то переменную из оперативной памяти, она автоматом проходит через весь конвейер кэшей и подгружается во все три кэша.

Помимо страниц, память делится на кусочки по 64 байта, которые называются кэш линии (**cache lines**). Кэши оперируют кэш линиями. Именно поэтому между шагом в 1 и в 16 в цикле нет особой разницы, поскольку все равно приходится подгружать примерно столько же кэш линий. 16 — пороговый шаг, после которого мы начинаем пропускать некоторые кэш линии и загружать не все из них.

Всякий раз, когда обращаемся к числу, соответствующая ему кэш линия либо находится в одном из кэшей и, после того как достали ее, подгружается во все предыдущие, либо находится в оперативной памяти и все равно после обращения подгружается во все кэши.

Поэтому если проходимся по массиву подряд, это работает гораздо быстрее, чем если бы мы обрабатывали его ячейки в рандомном порядке, потому что это вынуждает процессор забывать старые кэш линии и подгружать новые. Такая работа с рядом лежащими значениями и есть кэш локальность.

L1 кэш делится на 2 кэша:

* **L1 data cache** — тот который обсуждали выше, с переменными
* **L1 instruction cache** — код, подгружаемый в память. Если есть какая-то функция, которая часто исполняется из разных мест, то ее код будет лежать близко к процессору в кэше, и ее код не придется заново подгружать.

Это разделение сделано, чтобы кэши не вытесняли друг друга.

Когда загружаем функцию, которую никогда до этого не исполняли, сама загрузка ее бинарного кода из RAM занимает больше времени, чем переход в функцию, которую уже недавно исполняли.

Есть забавные приемы сделать так, чтобы код выполнялся быстрее. Если, допустим, есть какая-то функция «отправить заказ», довольно легковесная, тогда, чтобы сделать работу быстрее, выполняем постоянно эту функцию. Но отправки заказа при этом не происходит: ее сдерживает один бит в каком-нибудь регистре, отвечающий за то, надо ли заказ действительно отправить. Такой трюк обеспечивает то, что код функции постоянно находится в instruction cache, и его не приходится долго откуда-то доставать.

**Beer hierarchy**

Представим, что захотелось попить пива (зачем представлять…).

Бывают разные ситуации, как долго приходится доставать пиво (ну или переменную в нашем случае).

1. Переменная в регистре — пиво уже в руке, достаточно просто взять и выпить.
2. L1 cache — сижу за столом, а рядом со мной на столе на другом краю пиво стоит. Доступ недолгий — возьми пиво и выпей.
3. L2 cache — стою где-то недалеко на кухне, а на столе все так же стоит пиво. Нужно дойти до стола, взять пиво и выпить.
4. L3 cache — где-нибудь на диване в комнате лежу, а пиво на кухне в холодильнике. Нужно дойти до кухни и взять пиво из холодильника.
5. Обращение к RAM — сходить в магазин за пивом.
6. Обращение к диску — в регионе пиво не продают, надо лететь за пивом в другой город… Если SSD — на элке скататься в соседний город, HDD — слетать на самолете в другую страну))
7. Получить значение по сети — самому пиво сделать…

Еще один известный пример: обход матрицы. Что быстрее, обойти матрицу по строкам или по столбцам? Конечно, по строкам, потому что, проходя матрицу по строкам, мы обходим элементы подряд, а используя проход по столбцам, пришлось бы заново подгружать каждый раз кэш линии, что дорого.

**Instruction-level parallelism**

Есть два цикла.

int steps = 256 \* 1024 \* 1024;

int[] a = new int[2];

// Loop 1

for (int i=0; i<steps; i++) { a[0]++; a[0]++; }

// Loop 2

for (int i=0; i<steps; i++) { a[0]++; a[1]++; }

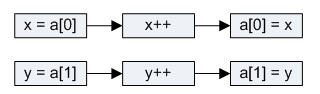
Оказывается, что второй цикл работает почти в два раза быстрее первого. Потому что процессор обладает instruction-level пареллелизмом: он видит, что мы делаем операции над ячейками, которые друг от друга не зависят, тогда за один проход он может сразу инкрементировать обе ячейки — и по сути это займет одно действие.

Причем в ассемблере количество инструкций будет одинаковым, но сам процессор оптимизирует исполняемый код.

Наглядно: первый цикл

****

Второй цикл:



**Branch prediction**

#include <algorithm>#include <ctime>#include <iostream>int main(){ // Generate data const unsigned arraySize = 32768; int data[arraySize]; for (unsigned c = 0; c < arraySize; ++c) data[c] = std::rand() % 256; // !!! With this, the next loop runs faster. std::sort(data, data + arraySize); // Test clock\_t start = clock(); long long sum = 0; for (unsigned i = 0; i < 100000; ++i) { for (unsigned c = 0; c < arraySize; ++c) { // Primary loop. if (data[c] >= 128) sum += data[c]; } } double elapsedTime = static\_cast<double>(clock()-start) / CLOCKS\_PER\_SEC; std::cout << elapsedTime << '\n'; std::cout << "sum = " << sum << '\n';}

Рассмотрим этот код. Завели массив на много элементов, заполнили рандомными числами. Затем выполняем цикл, который повторяется много раз и считает сумму элементов, если выполняется условие.

Вопрос: почему на отсортированном массиве это работает гораздо быстрее?

|  |  |
| --- | --- |
| Без сортировки:  time = 10.8894  sum = 314931600000 | С сортировкой:  time = 3.22172  sum = 314931600000 |

Мы жертвы branch prediction fail!!! Вопрос отсюда взят:

[https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array/11227902#11227902](https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array/11227902" \l "11227902)

Что такое branch prediciton? Представим железнодорожный разъезд. Мы едем на поезде и не знаем, куда указывает стрелка. Перед этим разъездом нужно остановиться, проверить, что стрелка ведет куда надо и, если нужно, перевести ее, и ехать дальше.

Но что, если мы знаем, что один путь — главный, а второй почти никогда не используется? Разумнее было бы ехать, не останавливаясь, а если вдруг поняли, что заехали не туда, отъехать назад, перевести стрелку и поехать уже в нужном направлении.

Если одна из ветвей условия выбирается почти всегда, то можно «не останавливаться перед стрелкой», а проехать как всегда, понадеявшись, что выполняется то условие, которое обычно. А если обнаружим, что условие оказалось неверно, откатываемся и после этого уже делаем то, что нужно.

То есть для процессора if — это стрелка.

Процессор старается заранее подгрузить наперед те инструкции, которые он собирается выполнять. Когда встречается на пути if, процессор не знает заранее, что подгружать. А так как подгрузка выполняется быстрее, чем вычисление операции сравнения, то сравнить числа еще не успели, а подгрузка инструкций уже произошла. Тогда процессор пытается угадать (из каких-то своих соображений), по какой ветке ifа мы пойдем следующим шагом (до того как if выполнился!).

Если обе ветки if примерно равновероятны (как в случае с неотсортированным массивом), то в половине случаев процессор ошибается. Поэтому приходится много откатываться и подгружать другие инструкции, и этот случай работает дольше. Если же массив отсортирован, то сначала выполняется много раз одна ветка if, затем много раз другая, то после нескольких итераций процессор запоминает, по какой ветке постоянно идем и подгружает в дальнейшем именно ее на исполнение.

Однако, начиная с C++20 есть атрибуты [[likely]] и [[unlikely]], которые подсказывают компилятору, какая из веток if более вероятна. Тогда компилятор попытается сгенерировать код так, чтобы процессору было сразу понятно, какую инструкцию правильнее подгружать.

Теперь представим, что мы делаем бинпоиск в большом массиве, и постоянно происходит так, что приходится искать маленькие числа (что-то близкое к началу массива). Тогда такой бинпоиск будет работать быстрее, чем тот, который равновероятно обращается к разным концам массива.

Как именно это работает — хз)

**Ассемблерные вставки**

Иногда в код на C очень хочется вставить кусочек ассемблера. Но как вставить кусок ассемблера, если мы заранее не знаем, в какие регистры какие переменные положил компилятор? И не знаем, из каких регистров брать значения переменных.

**Синтаксис:**

asm asm-qualifiers ( AssemblerTemplate

: OutputOperands

[ : InputOperands

[ : Clobbers ] ])

AssemblerTemplate — шаблонные инструкции того, что хотим сделать

Далее в определенном формате указываем выходные и входные операнды.

**Пример:**

#include <iostream>int main() { int src = 1; int dst; asm ("mov %1, %0\n\t" "add $1, %0" : "=r" (dst) : "r" (src)); printf("%d\n", dst);}

Этот код копирует src в dst и добавляет 1 к dst. Результат: 2.

Под %0 и %1 подставляются какие-то регистры. %0 — нулевой по счету из указанных имен переменных среди параметров, %1 — первый по счету.

**volatile** — квалификатор, который отключает компиляторные оптимизации. Часто используется с ассемблерными вставками, чтобы компилятор делал именно то, что написано (пишем *asm volatile*).

**Для чего они могут быть нужны?** Иногда мы хотим прочитать значение с какого-то регистра или записать в него что-то. Пример такого кода:

#include <stdio.h>unsigned long long rdtsc(void) { unsigned long long tsc; asm volatile("rdtsc" : "=A" (tsc)); return tsc;}int main(void) { unsigned long long start, end, ticks; start = rdtsc(); int x = 0; for (int i = 0; i < 100; ++i) { x += 5; } end = rdtsc(); ticks = end - start; printf("Ticks: %llu\n", ticks);}

В процессоре есть много разных регистров, в которых хранится полезная информация, появляющаяся по ходу выполнения программы. Например, регистры, в которых записаны адреса, по которым процессор пойдет обрабатывать прерывания, если они случатся. Процессор помнит, по какому адресу пойти исполнять код, если случается pagefault, например; помнит, где искать таблицы страниц. И есть также регистры, в которых ведется счетчик тактов — tsc. В данном коде цель: замерить количество процессорных инструкций между двумя участками кода. Вызов стандартных функций стоит дорого, поэтому на помощь приходит ассемблерная вставка.

Здесь **rdtsc** — ассемблерная инструкция, которая берет значение из регистра tsc и записывает его в переменную tsc.

**Как работает ОС то по итогу?**

У всех современных процессоров есть разные режимы работы — привилегированный и непривилегированный (обычный). В первом режиме доступно больше инструкций. ОС исполняется в привилегированном режиме.

**Примеры привилегированных инструкций:**

**HLT** — приостановить процессор, пока не случится следующее прерывание.

**INVLPG** — инвалидировать одну из записей в TLB cache.

**LIDT** — загрузить таблицу прерываний (перед запуском программы процессору сообщается адрес, по которому находится код ядра ОС, куда надо идти в случае прерывания, прихода какого-то сигнала).

**MOVE CR** — загрузить или взять что-то из control registers.

sudo не дает привилегированного режима на уровне процессора, это привилегированный пользователь на уровне ОС. У него больше прав в системе, но нет прав исполнять привилегированные инструкции.

Все пользовательские процессы работают в обычном режиме процессора. Процессор же всегда помнит, в каком он режиме сейчас работает. У него есть специальные регистры: **control registers**. В одном из таких регистров хранится информация о режиме, в котором исполняется процессор. Также есть, к примеру, регистры:

**CR0** — включена ли виртуальная адресация. Когда переходим в привилегированный режим, процессор выключает виртуальную адресацию (идем по адресам напрямую, защита выключается).

**CR2** — в нем хранится адрес памяти, куда прыгать в случае pagefault (обращение к незагруженной странице). Адрес туда заранее ОС положила.

**CR3** — адрес, указывающий на начало таблицы страниц для текущего исполнения.

**Что на самом деле происходит, когда вызываем сискол?**

Все сисколы из 2 раздела — на самом деле обертки над сисколом.

Настоящий сискол это ассемблерная инструкция — **syscall**.

Проблема в том, что перед тем, как сделать сискол, нужно, согласно спецификации линукса, в правильные регистры положить аргументы и численный номер этого сискола, чтобы когда пришла ОС, она, глядя на определенный регистр, поняла по номеру, какой сискол ей нужно выполнить.

syscall отличается от call тем, что при ее исполнении процессор переключается в привилегированный режим, открывает новый стек (вроде сисколы исполняются на другом стеке), и вызывает код ядра ОС по адресу, который заранее записан в одном из control registers. Код ядра ОС первым делом смотрит на номер сискола и кто его попросил исполнить (понимает он это тоже смотря на определенные регистры). После выполнения сискола ОС выполняет инструкцию sysret — выход из сискола. Благодаря этой инструкции мы обратно переходим в обычный режим, все регистры возвращаем в изначальное состояние (какими они были до вызова сискола), результат сискола тоже записывается в определенный регистр.