

## 2.6

# Многоядерные/многопроцессорные системы. Одновременная многопоточность. SMT/HT.

Даниэль Ползик

8 января 2020 г.

## 1 Ядро vs Процессор.

### Многоядерность vs Много(мульти)процессорность. В чем разница?

#### 1.1 Ядро - часть процессора

Процессор - это всё, что находится на кристалле. Включая всякие кэши, системы предсказаний и т.п.

Ядро - это вычислитель. Простым языком на ядрах выполняются все команды. Обычно у каждого ядра есть 2 уровня кэшей: L1 (L1i и L1d), L2. И еще у всех ядер процессора есть общий уровень кэша - L3 (подробнее в билете про кэши)

#### 1.2 Немного историчеких справок

##### 1.2.1 Про многоядерные

Уже с 2001 года разные компании такие, как **IBM**, **Intel**, **AMD**, очень хотели создать многоядерные процессоры и даже создавали что-то наподобии.

Это были действительно процессоры с 2-мя ядрами на кристалле, но вот в чем штука. По факту, это были двухпроцессорные системы записанные в один кристалл и они не сильно взаимодействовали и параллельность была слабовата. По скорости эти "двухядерные процессоры" не были сильно быстрее.

Но к 2005 году начали появляться уже **реальные** без кавычек двухядерные процессоры с измененной под параллельность архитектурой. А в 2006 Intel выпустил для персональных компьютеров: *Core 2 Duo*

Дальнейшая история не сильно поможет в сдаче экзамена, но если очень хочется - википедия вам в помощь. Если захотелось, то советую ознакомиться с разборками Intel и AMD. Легендарная сага!

### 1.2.2 Про многопроцессорные

История многопроцессорных систем еще менее полезная, но просто чтобы вы понимали, насколько это было раньше многоядерных процессоров:

Первая система была создана в 1970-х, а в 1980-х появились уже вполне себе рабочие системы.

### 1.2.3 Пояснение простым языком к историческим справкам

Как вы знаете из прошлых билетов мы создали кучу всяких фишек для увеличения скорости: кэши, предсказатели и даже superscalar/VLIW. Но все хотят играть в Доту (которой еще нет), а для этого нужно ускорять работу.

Для этого мы понимаем, что можно добавить на комп еще один процессор, который будет работать с той же оперативной памятью и сможет делать что-то параллельное. **Важно!** Это не тоже самое, что поставить два компа рядом. Оперативка-то общая.

Как вы наверно догадываетесь, два процессора занимают место и один из них находится сильно дальше, чем другие от оперативки и нужных вещей, потому либо один процессор работает быстро, а остальные сильно медленнее (тогда зачем они нужны), либо все процессоры равноудалены и тогда их всех близко не расположить и значит все они работают медленнее. К тому же еще фиг знает, как распределять работу между процессорами. В те времена это скинули на программистов. Но программы, которые бы хорошо параллелились сильно больше не стало и в итоге почти всегда работал только один процессор из всех.

Короче говоря, фишка с несколькими процессорами конечно ускоряла, но не так, как хочется. А хочется, чтобы ты удвоил кол-во процессоров и скорость увеличилась в 2 раза. Но это было далеко не так. Еще и не всегда ускоряет. Короче, сплошное фиаско, еще и стоит дороже.

P.S. В те далекие времена процессор состоял только из вычислителя и поэтому тогда считалось: *процессор = ядро* Дальше мы уже говорим о процессорах помоднее, на которые напиханы предсказатели, кэш, ви-

деокарточка, вычислители и уже оставшееся место занимает исполнитель(ядро). То есть теперь: *процессор*  $\neq$  *ядро*

Наука развивается и компании хотят это использовать. И что же они делают? Идейно: они хотят записать на один кристалл несколько ядер (для начала хотя бы два), так как наука стала позволять это делать. Но для этого нужно переделать всю архитектуру процессора, а компании про которые мы говорим уже имеют надежные архитектуры, но для одного ядра. И что же вы думаете они делают? Да, они берут пихают на один кристалл грубо говоря два процессора, которые друг с другом почти не взаимодействуют.

Вроде бы хорошо, но скорости для доты всё еще не хватает, так как остаются те же проблемы, что и у многопроцессорных систем. Тогда разработчикам архитектур приходится оторвать свои пятые точки и начать создавать нормальные архитектуры для двух ядер. Что Intel успешно и сделал в 2006 году. Но по факту, проблемы с тем, как распараллелить остались, но благодаря новой архитектуре стало возможным добавление фич, которые действительно параллелят. И к тому же все стало ближе друг к другу, а это не хухры мухры.

## 2 Но как то же параллелят. Как?

### 2.1 Иллюзия параллельности. Прерывания

**Прерывание** (*interrupt*) - сигнал процессору о событии, которое требует его внимания. Процессор отвечает на сигнал прерывания, останавливая выполнение текущей задачи, сохраняя её состояние и вызывает **обработчик прерываний** (*interrupt handler*).

*Бывает два вида прерываний:*

**Аппаратное прерывание** используется устройствами чтобы сообщить о том, что им требуется внимание со стороны операционной системы. Например, нажатие клавиши на клавиатуре вызывает прерывание, после чего процессор считывает, какая клавиша была нажата, а затем продолжает выполнение.

**Программное прерывание** вызывается программным исключением (например, делением на ноль) или специальной инструкцией, которая вызывает прерывание при исполнении.

ОС с каким-то промежутком переключается между задачами, прерывая одну и запуская другую. Так как это происходит быстро, то создается

иллюзия многозадачности(параллельности).

## 2.2 Иллюзия параллельности. SMT/HT

### 2.2.1 Процессы. Треды

**Процесс** - экземпляр компьютерной программы, запущенный на выполнение. У каждого процесса своя память и свои права.

**Тред** - наименьшая последовательность инструкций, которая может независимо управляться планировщиком. У каждого тредя свои регистры (IP и общего назначения), свой выделенный стек в общей для процесса оперативной памяти (треды не пересекаются).

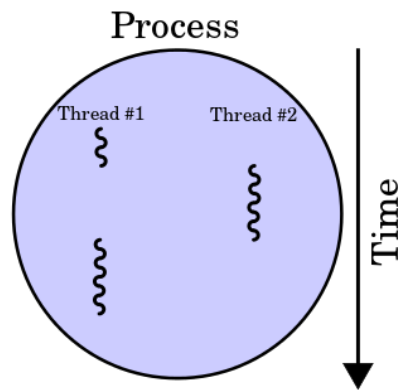


Рис. 1: Процесс с двумя тредями, запущенными на одном процессоре

Теперь распределение памяти происходит на уровне процессов, а распределение вычислительной мощности - на уровне тредов. Но чтобы вся эта прелесть работала и давала ускорение, нужно, во-первых, чтобы задачу вообще можно было распараллелить, а во-вторых, программисту нужно правильно написать.

### 2.2.2 SMT/HT

**SMT** - *Simultaneous multithreading* - техника, улучшающая эффективность работы суперскалярной архитектуры. Один физический CPU притворяется двумя ненастоящими(логическими). Треды независимы, поэтому планировщику проще найти независимые команды, следовательно конвейеры эффективнее используются.

**HT** - *Hyperthreading* - SMT у Intel. Ядро с SMT должно иметь больше регистров (отдельные для каждого тредя).

В лучшем случае мы можем получить выигрыш в два раза. В худшем тредам может не хватить кэша и тогда бо-бо. И мы очень сильно проигрываем.

Выигрыш от SMT получается, если:

- Тредов больше, чем физических ядер
- Каждый тред "написан не очень хорошо т.е. не по максимуму использует конвейеры.

А проигрываем так же, как и в НТ при нехватке кэша на треды.

## 3 Видеокарточки

### 3.1 Поток сознания

В начале времен видеокарточка была просто контроллером памяти: переводила данные из видеопамяти в аналоговый сигнал для ЭЛТ-монитора.

А потом стали популярны окошечные интерфейсы типа Windows 3. В чем проблема? Нужно заливать цветом большие области. Программно так делать можно было, но лагало (т.к. процессор занимался рисованием окошечек вместо полезных дел). Чтобы не лагало, появились первые 2D-ускорители.

2D-ускоритель - специальный блок на видеокарточке, который умел выполнять операции вида "залить блок таким-то цветом".

С другой стороны, появился разный САПР (система автоматизированного проектирования) софт, который позволял смотреть на чертежи в 3D. Но этим САПРам требуется много чего считать с плавающей точкой.

3D-ускорители аппаратно делали необходимые вычисления, но стоили дико дорого и встречались в специализированном железе. 3dfx - выпустила на рынок дешевый аппаратный 3D-ускоритель Voodoo.

Как выглядела Voodoo: отдельное устройство, один провод от видеокарточки, другой в монитор. Эта штука умела только в 3D, причем только в полноэкранный.

3dfx - появилась технология SLI: возможность использовать две карточки, одна будет считать четные строки, другая нечетные.

### 3.2 OpenGL & DirectX

OpenGL - 3D API для расчетов всякого разного 3D.

DirectX - 3D API от Microsoft, смысл которого - захватить мир.

### 3.3 Продолжение потока

Через некоторое время стало понятно, что аппаратно реализовать все извращения, которые хочется игрописателям, не представляется возможным, т.к. их слишком много. Очевидное решение: сделать часть процесса 3D-проектирования программным. Программные кусочки получили название **шейдеры**.

Исходно было два вида шейдеров: пиксельные и вершинные.

Вершинные (вертексные) отвечали за геометрию, пиксельные (фрагментные) - за закраску геометрии (текстурами, например).

Шейдеры позволили делать более хитрое освещение и т.п.

Исходно в видеокарточке половина блоков считала вершинные шейдеры, половина - пиксельные. Но была проблема в том, что игры неравномерно использовали шейдеры. (Есть два способа нарисовать кружочек: сделать много треугольничков(нагрузка на вершинные) или сделать два треугольничка, но сложную текстуру с прозрачностью(нагрузка на пиксельные)).

Со временем сделали универсальные вычислительные блоки.

### 3.4 GPGPU

**В видеокарточках нет одной из важных возможностей процессора - аппаратного разграничения доступа.**

По уровню вычислительной гибкости карточки почти дошли до уровня процессоров. В результате их стали пытаться использовать в качестве GPGPU.

**GPGPU** (*General Purpose computing on GPU* - вычисления общего вида на видеокарточках, не обязательно графика).

Сначала это работало так: рисовался прямоугольничек, создавались текстуры, в которые заливались текстуры входных данных), потом с помощью пиксельных шейдеров эти текстуры накладывались так, чтобы всё считалось как надо. Выходные данные тоже записывались в текстуру.

Далее стали появляться различные API, которые стали позволять считать не-графику на видеокартах с меньшим количеством извращений.

**CUDA** - платформа и API от NVIDIA, которая позволяет вставить в код конструкции со смыслом "а вот это посчитать на видеокарточке". Потом этот код разбивается на две части: одна компилируется вашим компилятором для исполнения CPU, другая компилируется под видеокарточку компилятором NVIDIA. В итоге получается код, зависимый от библиотек NVIDIA.

**OpenCL** - открытый стандарт, который проспонсировали Apple.

### 3.5 CPU vs GPU

В у нас есть много однотипных вычислений над разными точками. И эти вычисления выглядят как треды, выполняющие один и тот же код, но над разными данными (SIMT). Вместо того, чтобы создать цикл по всем точкам изображения, создаём много тредов, каждый из которых считает одну точку.

Следовательно, у нас есть много независимых вычислений. Кроме того, нас не интересует время выполнения одного треда, а только *выполнение всех тредов в целом*.

Если много тредов, то имеет смысл сделать много простых ядер. Т.к. нас не интересует время выполнения одного треда, то мы можем сэкономить на некоторых штуках, которые нам это время оптимизировали. Во-первых, можно выкинуть кэш (т.к. latency решается по-другому). Во-вторых, можно выкинуть планировщик. В-третьих, можно выкинуть всю невычислительную логику, например, предсказатели переходов.

Как решить проблему с долгим откликом памяти? На каждом ядре запустить много-много тредов. Как это работает: ядро начинает исполнять первый тред, исполняет, пока не случается обращение к памяти, ядро посылает команду обращения к памяти и переходит к следующему треду и т.д.

Ключевая разница с SMT: логически SMT исполняет два треда и физически исполняет два треда, а в потоковом ядре логически исполняется 16 тредов, а физически только один.

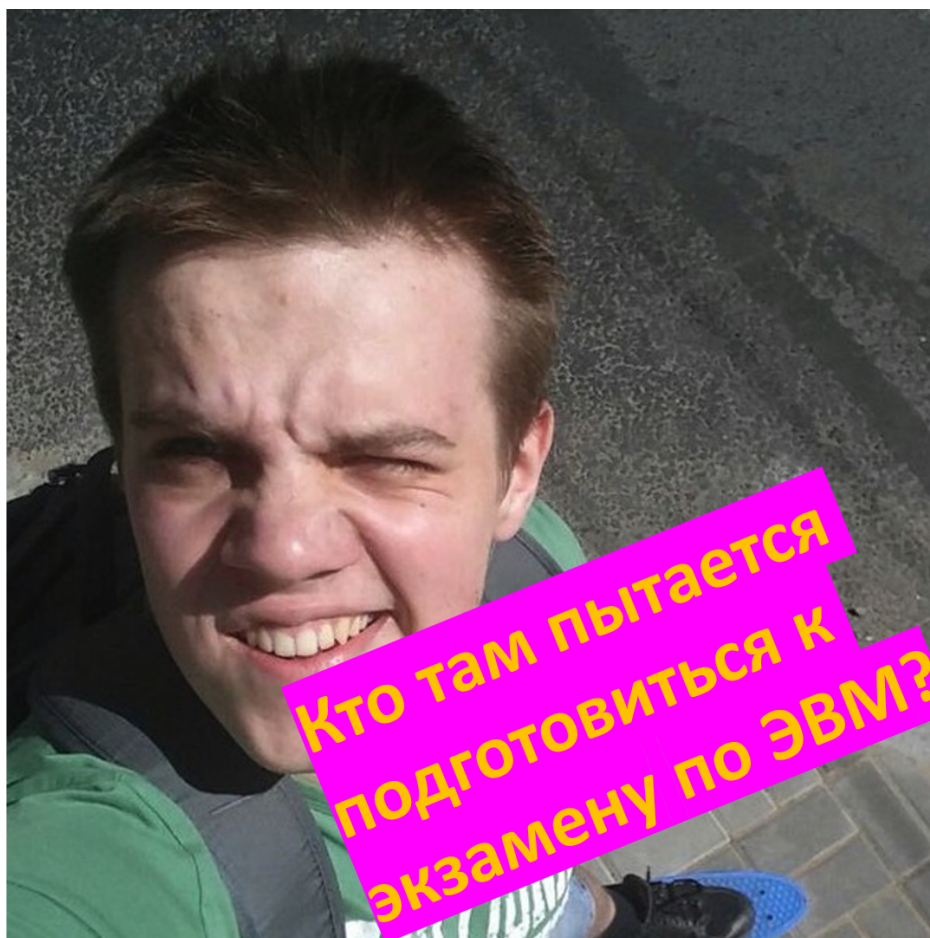


Рис. 2: Удачи!