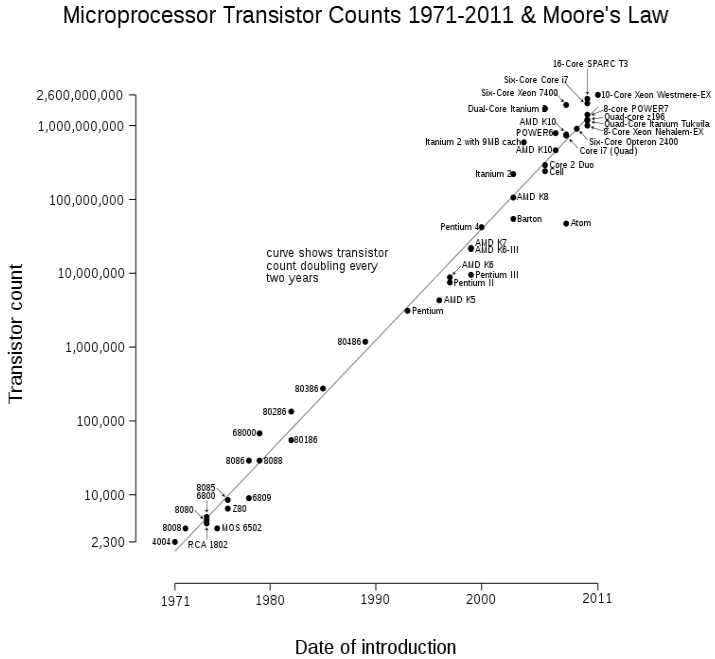
Многопоточное программирование на C++

О курсе, синхронизации и задачах

О параллелизме, многопоточности и лени

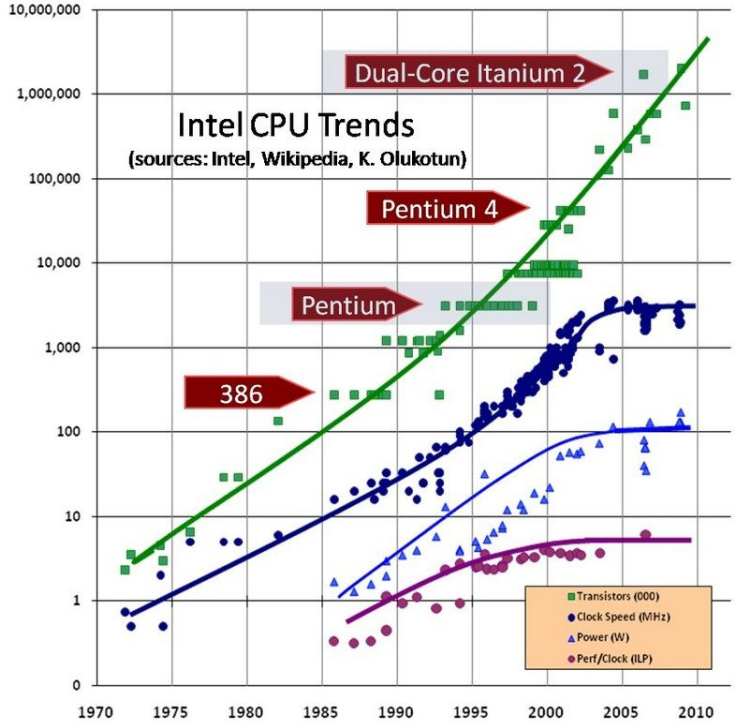
- Закон Мура
  - Примерно с середины десятилетия 2000-х годов по разным причинам производители процессоров предпочитают многоядерные архитектуры

- Закон Амдала
  - В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента»



О неизбежности многопоточности

- Количество задач растет
  - Количество данных растет
  - А так же число методов обработки
- Вычислительная мощность?



Чего не будет

- Задач о филосовах
- Рассказов о API (почти)
- Распределенных систем

Чем будем заниматься

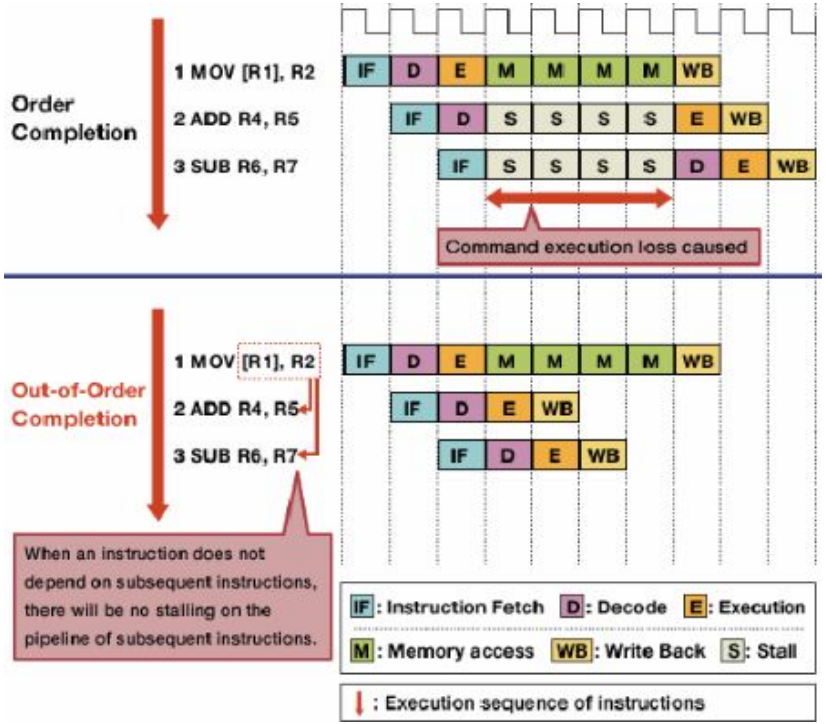
- Посмотрим какие бывают практические задачи
- Рассмотрим разные варианты их решения
- Попробуем понять недостатки и достоинства каждого из методов

Условия игры

- Linux: CentOS 7, ArchLinux
- Количество домашек:
  - Обязательных: 4
  - За остальные даем баллы
    - На 4 нужно: обязательные + все кроме 3
    - На 5 нужно: обязательные + все кроме 1
- Сроки сдачи:
  - Каждая домашка, если не оговорено отдельно, на 2 недели
  - Каждую неделю - новая домашка
  - Задача, не сданная в срок, стоит половину номинала
  - Если задача появилась в репозитории в срок, выглядит более-менее прилично, но есть замечания, то задача все равно сдана в срок

Железки

Порядок выполнения инструкций



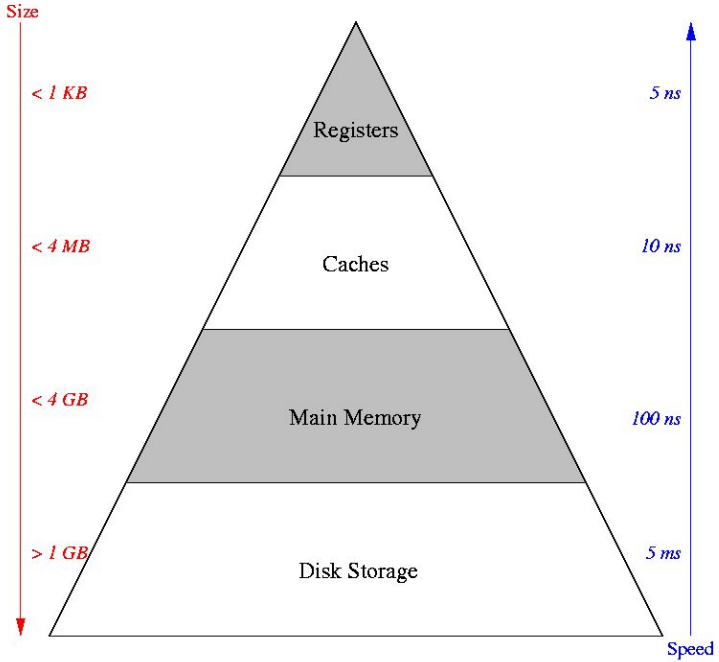
Последовательный:

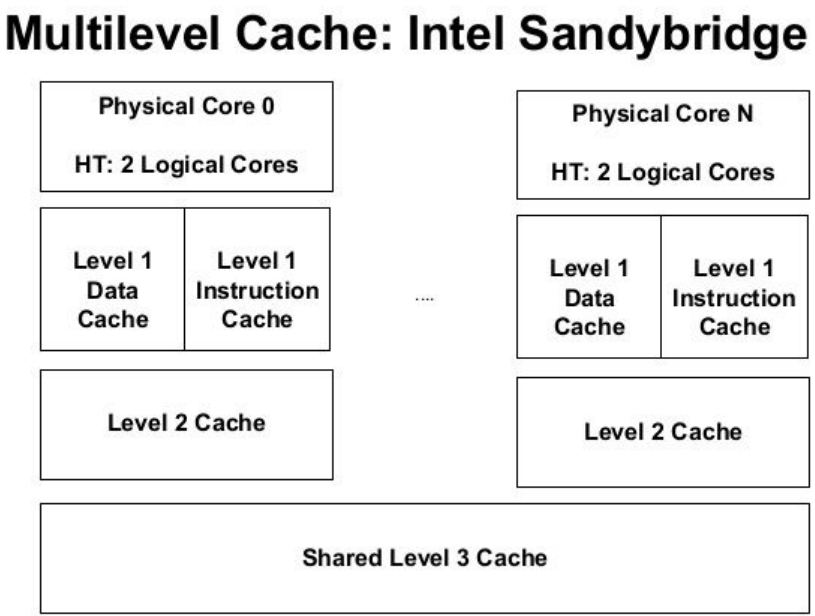
1. Загрузка инструкции
2. Если операнды доступны (к примеру уже в регистрах), инструкция отправляется на вычисление.Если хотя бы один операнд сейчас не доступен (обычно его нужно загрузить из памяти), процессор останавливается пока идет загрузка
3. Инструкция исполняется на соответствующем логическом модуле
4. Результат записывается в выходной регистр

Внеочередной:

1. Загрузка инструкции
2. Инструкция помещается в очередь
3. Ожидание в очереди, пока все операнды не станут доступны. При этом возможно что инструкция пришедшая позже покинет очередь раньше других
4. Инструкция исполняется на каком-то логическом модуле
5. Результат отправляется в очередь
6. После того, как все предшествующие инструкции записали результат в выходные регистры,текущая инструкция может записать свой. (graduation или retire stage)

Доступ к памяти





Компилятор

Порядок инструкций

```
int A, B;

void foo() {
    A = B + 1;
    B = 0;
}
```

Компилятор может (*и будет!)* добавлять, *переставлять*, менять и даже удалять инструкции, если может доказать, что наблюдаемый результат такой же, как и в изначальном варианте

```
$ gcc -S -masm=intel foo.c
$ cat foo.s
        ...
        mov     eax, DWORD PTR _B
        add     eax, 1
        mov     DWORD PTR _A, eax
        mov     DWORD PTR _B, 0
        ...
```

```
$ gcc -O2 -S -masm=intel foo.c
$ cat foo.s
        ...
        mov     eax, DWORD PTR B
        mov     DWORD PTR B, 0
        add     eax, 1
        mov     DWORD PTR A, eax
        ...
```

Софт

mutex

Бинарный семафор, позволяет сделать только lock/unlock

- std::mutex
  Простой mutex, нельзя захватывать рекурсивно

- std::timed_mutex
  Можно захватить на какое-то время

- std::recursive_mutex
  Можно захватывать рекурсивно

- std::recursive_timed_mutex
  Композиция всех типов

lock

Обертка для безопасного захвата и освобождения mutex’а

- std::lock_guard
  Просто делает lock/unlock в текущем блоке

- std::unique_lock
  Захватывает/отпускает mutex, но поддерживает все возможные опции, такие как захват на время, не захватывать лок, но безопастно его отпустить, и.т.д

condition_variable

Позволяет дождаться определенного события, случившегося в другом потоке

- std::condition_variable
  - wait
  - wait_for
  - wait_until
  - notify_one
  - notify_all