Лекция 4

Управление потоками

Created by drewxa@

Содержание

- параллелизм в ПО
- переключение контекста потоков
- `std::thread`
- `std::async`
- `std::feature`

Параллелизм

Говоря о параллелизме, мы имеем ввиду, что несколько задач выполняются одновременно.

Аппаратным параллелизмом

Оборудование с одноядерными процессорами не способно выполнять одновременно несколько задач. Но они могут создавать иллюзию этого.

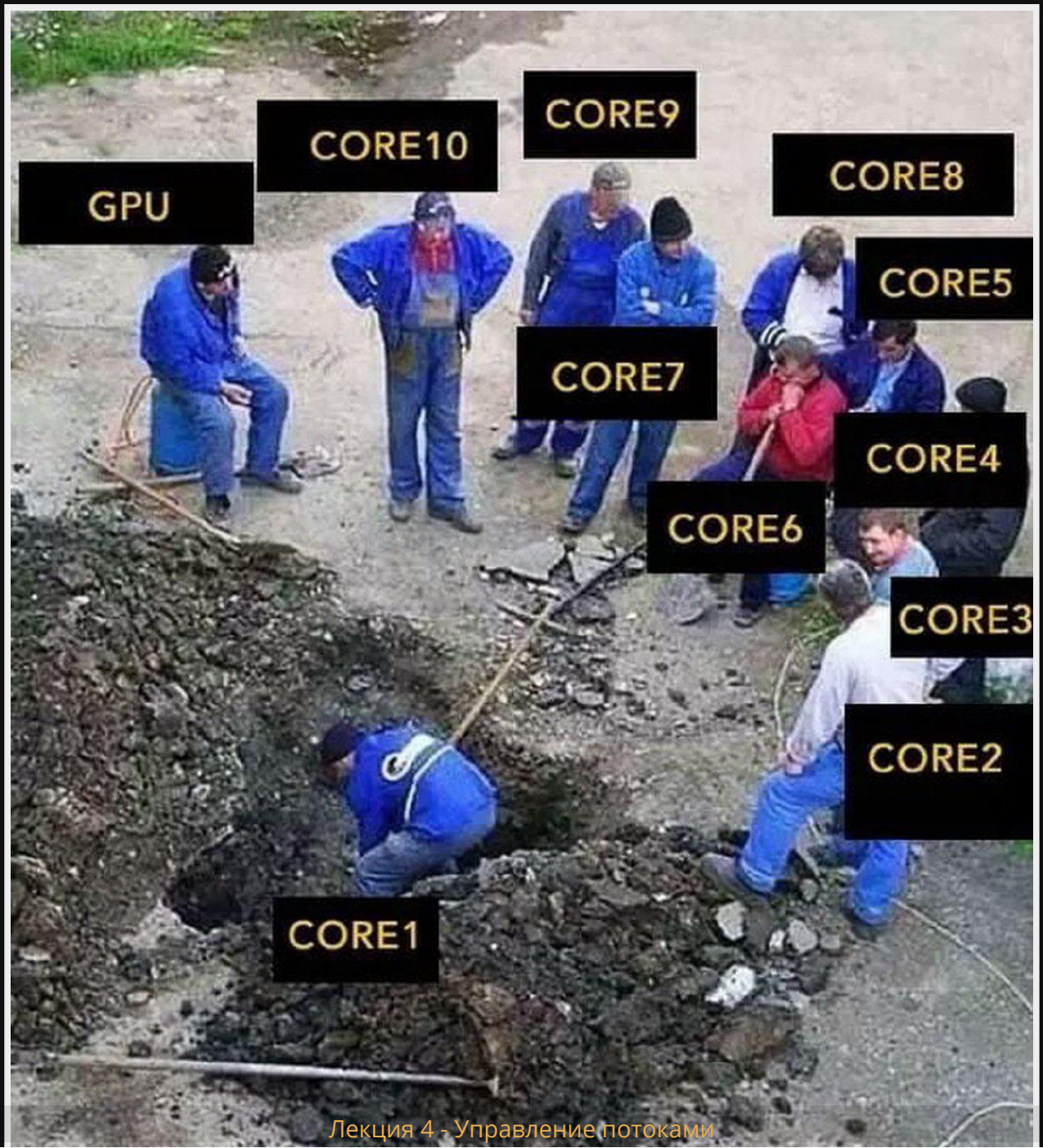
Оборудование с несколькими процессорами (или несколькими ядрами на одном процессоре) может выполнять несколько задач одновременно. Это называется аппаратным параллелизмом.

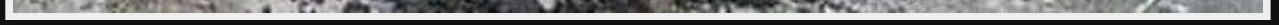
Причины использовать параллелизм

Первая причина для использования параллелизма - это разделение обязанностей. Например, пользовательский интерфейс зачастую выполняется в отдельном потоке, в то время как основная логика ПО выполняется в иных потоках.

Причины использовать параллелизм

Другая причина использования параллелизма - повышение производительности. Сейчас существуют персональные компьютеры с 16 и более ядрами (не говоря уже о серверном оборудовании). При этом использование только одного потока для выполнения всех задач является ошибкой.


GPU
CORE10
CORE9
CORE8
CORE5
CORE7
CORE4
CORE6
CORE3
CORE2
CORE1



Разделение обязанностей

Самые распространненые применения параллелизма:

- отзывчивый интерфейс приложения
- запуск длительнных операций в отдельном потоке (например, сетевые запросы, чтение и запись на диск)

Повышение производительности

Параллелизм ради повышения производительности приложения встречается не так часто, как выполнение асинхронное выполнение IO операций. Однако, использование параллелизма позволяет значительно повысить производительность приложения.

Контекст потоков

Чтобы операционная система поддерживала многозадачность, каждый выполняемый поток должен обладать своим контекстом исполнения.

Контекст потоков

Этот контекст используется для хранения данных о текущем состоянии потока (значения регистров процессора, указателя на стек данных, указатель на текущую выполняемую команду).

Многозадачность в ОС

В системе количество потоков может превышать (почти всегда превышает) число ядер, тогда будет применяться механизм переключения между выполнением задач.

Многозадачность на одном ядре

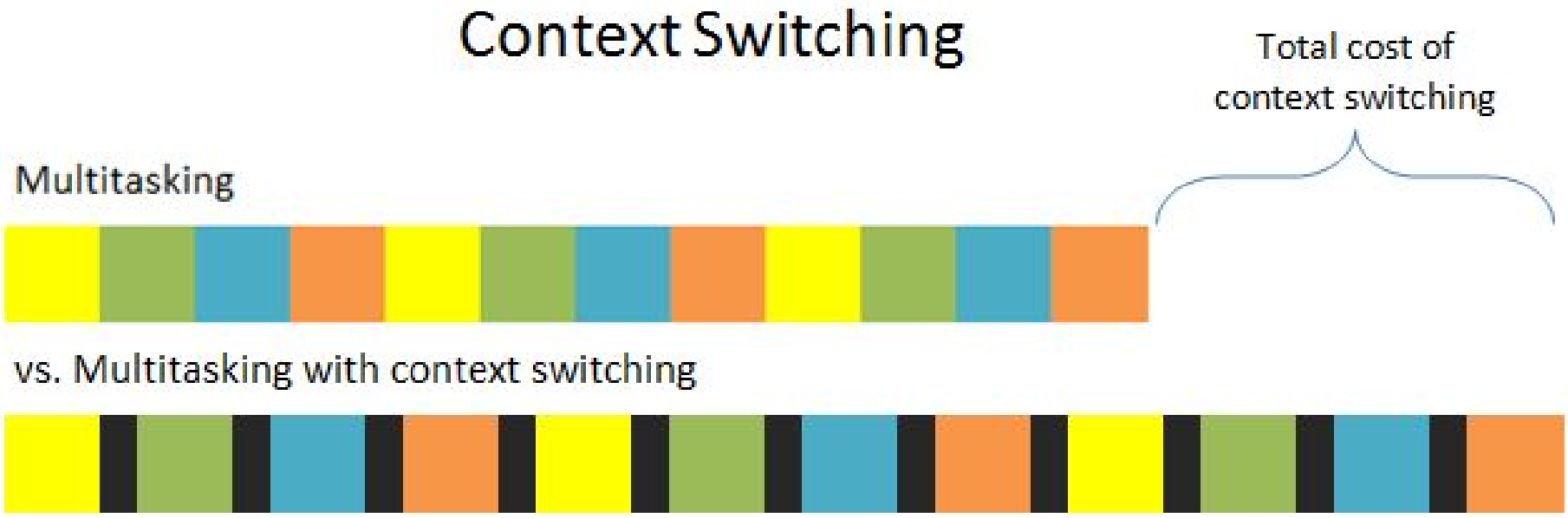
ОС передает поток на исполнение ядру процессора. Этот поток исполняется в течение некоторого временного интервала. После завершения этого интервала контекст ОС переключается на другой поток.

Переключение контекста

При переключении исполнения потоков происходит следующее:

- обновляется контекст текущего потока;
- из имеющихся потоков в ОС выбирается один, который будет исполняться на процессоре;
- загружается контекст выбранного потока.

Переключение контекста



C++11

В стандарте C++11 появились классы для управления потоками, синхронизации операций между потоками и низкоуровневыми атомартными операциями.

В качестве основы для библиотек по работе с многопоточностью в стандарте были взяты аналоги из библиотеки Boost.

Эффективность библиотеки многопоточности

Использование любых высокоуровневых механизмов вместо низкоуровневых средств влечет за собой некоторые издержки. Их называют платой за абстрагирование.

Одной из целей, которую преследовали при проектировании библиотеки многопоточности, была минимизация платы за абстрагирование.

Потоки

```cpp
#include <iostream>
#include <thread>

void hello() {
  std::cout << "Hello, World!";
}

int main() {
  std::thread th(hello);
  th.join();
}
```

std::thread

Создание объекта типа `std::thread` запускает новый поток.

До вызова деструктора у объекта необходимо вызвать или метод `join` или метод `detach`.

`std::thread::join`

Вызов метода `join` приведет к ожиданию завершения потока

Это значит, что до тех пор пока поток не завершит своё выполнение, основной поток не будет выполнять код находящийся после вызова метода `join`.

`std::thread::join`

Этот метод необходимо использовать, если основному потоку необходим и важен результат выполнения дочернего потока. Например, загрузка файла из сети и дальнейшая его обработка.

`std::thread::detach`

Вызов функции detach оставляет поток работать в **фоновом** режиме.

Это значит, что код находящийся после вызова метода `detach` может выполняться пока выполняется запущенный поток.

`std::thread::detach`

Этот метод необходимо использовать, если основному потоку не важен результат выполнения дочернего потока. Например, отправка пользовательской статистики.

Передача владения потоком

Класс `std::thread` является перемещающимся типом со всеми вытекающими последствиями:

- возможность передавать владение потоком “из рук в руки”
- позволяет хранить объекты типа `std::thread` контейнерах

std::async

Своеобразными “конкурентами” классу `std::thread` являются функция `std::async` и класс `future<T>`.

std::async

```cpp
#include <iostream>
#include <thread>

std::string hello() {
  return "Hello, World!";
}

int main() {
  std::future<std::string> res =
      std::async(hello);

  std::cout << res.get();
}
```

std::future<T>

Функция std::async возвращет объект класса future<T>, который предоставляет доступ к результату выполнения потока: возвращаемому значению или исключению.

`std::future<T>::get()`

При вызове функции `std::future<T>::get()` может произойти одно из трех событий:

1. Если выполнение асинхронной функции было начато функцией `async` в отдельном потоке и уже закончилось, то результат получится немедленно.

std::future<T>::get()

2. Если выполнение асинхронной функции было начато функцией async в отдельном потоке, но еще не закончилось, то функция get() блокирует основной поток до получения результата.

`std::future<T>::get()`

3. Если выполнение целевой функции еще не начиналось, то она начнет выполняться как обычная синхронная функция.

Исключения в потоках

Если выполнение фоновой задачи было завершено из-за исключения, которое не было обработано в потоке, это исключение сгенерируется снова при попытке получить результат выполения потока.

Исключения в потоках

```
auto f = std::async([](){
  throw 42;
});

try {
  f.get();
} catch(int i) {
  std::cout << i;
}
```

std::shared_future

Класс `std::future` позволяет обрабатывать результат параллельных вычислений. Однако этот результат можно обрабатывать только один раз. Второй вызов функции `std::future::get` приводит к неопределенному поведению.

`std::shared_future`

Иногда приходится обрабатывать результат вычислений несколько раз, особенно если его обрабатывают несколько потоков. Для этой цели существует `std::shared_future`

std::shared_future

Объекты типа std::shared_future допускают несколько вызовов метода get, возвращает один и тот же результат или генерирует одно и то же исключение.

Домашнее задание

- `thread_local`