Лекция 10. Многопоточное программирование.

Илья Макаров

ИТМО JB

9 ноября 2021

Санкт-Петербург

Многопоточное исполнение

Все современные программы многопоточные.
Какие проблемы решают разработчики?

- Deadlock'и.

Шаг	Процесс 1	Процесс 2
0	Хочет захватить A и B, начинает с A	Хочет захватить A и B, начинает с B
1	Захватывает ресурс A	Захватывает ресурс B
2	Ожидает освобождения ресурса B	Ожидает освобождения ресурса A

- Race condition'ы.

Многопоточное исполнение

Какие абстракции у нас есть?

- `std::thread`.
- `std::future` и `std::promise`.
- `std::atomic` и CAS.
- `co_await`, `co_yield` и `co_return` (C++20).

Какие библиотеки у нас есть?

- pthread.
- Boost.
- OpenMP и MPI.
- OpenCL и CUDA.
- …

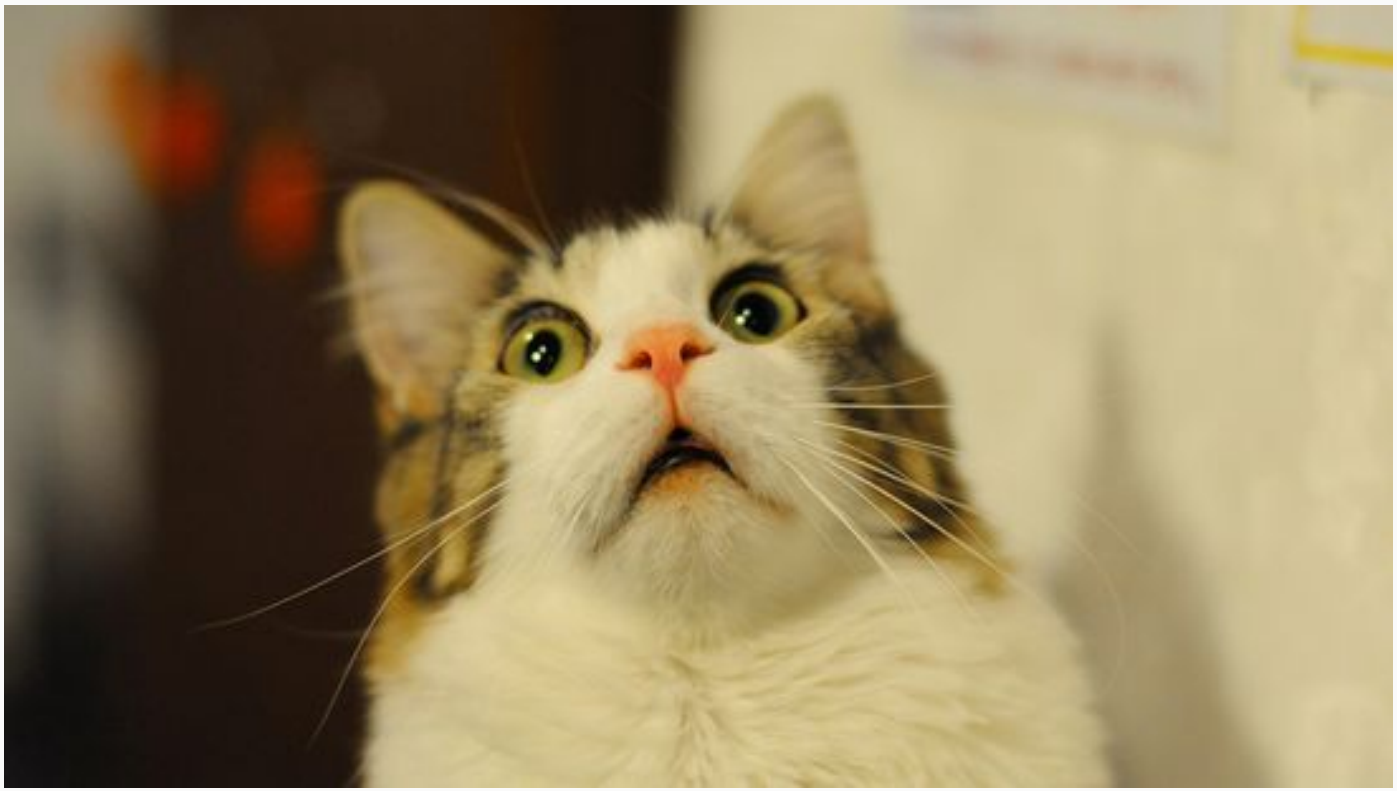
Синхронизация

Какие инструменты синхронизации есть?

- `std::mutex`, `std::recursive_mutex`, ...
- `std::lock_guard`, `std::scoped_lock`, `std::unique_lock`.
- `std::counting_semaphore`, `std::binary_semaphore` (C++20).
- `std::conditional_variable`, ...
- `std::memory_ordering`.

С помощью этих инструментов мы можем строить более сложные примитивы.

Просто кот



Асинхронное выполнение

Пусть мы хотим вычислить doAsyncWork асинхронно.

```
int doAsyncWork(); // many work here
```

В C++ есть несколько способов выполнения задач асинхронно:

- создать поток вручную `std::thread`,
- использование `std::async`,
- использовать корутины.

Асинхронное выполнение - вариант 1

```
int doAsyncWork(); // many work here
```

```
#include <thread>

int main()
{
    std::thread t(doAsyncWork);
    ...
    t.join();
}
```

std::thread

- Сразу же начинает вычислять переданную функцию.
- Игнорирует возвращаемое значение функции.

```
int main()
{
    int res = 0; // here we store the result
    std::thread t([&res](){res = doAsyncWork();});
    ...
    t.join();
    // res is ready here.
    ...
}
```

std::thread

- Метод `join()` позволяет заблокировать текущий поток, пока выполнение потока не завершится.
- Метод `detach()` позволяет отключить поток от объекта, т.е. разорвать связь между объектом и потоком.
- При вызове деструктора подключаемого потока программа завершается, т.е. необходимо вызвать `join` или `detach`.
- Исключения не могут покидать пределы потока.
- Метод `native_handle()` возвращает дескриптор потока.

Асинхронное выполнение - вариант 2

```
int doAsyncWork(); // many work here
```

```
#include <future>

int main()
{
    std::future<int> fut = std::async(doAsyncWork);
    ...
    int res = fut.get(); // or wait()
}
```

`std::async` может (зависит от планировщика) отложить выполнение задачи до вызова `get` или `wait`.

std::async

- Имеет две стратегии выполнения: асинхронное выполнение и отложенное (синхронное/ленивое) выполнение.
  1. `std::launch::async`
  2. `std::launch::deferred`
- По умолчанию имеет стратегию:
  `std::launch::async | std::launch::deferred`

```
int main()
{
    std::future<int> fut = std::async(
        std::launch::async,
        doAsyncWork);
    ...
    int res = fut.get();
}
```

std::async

- Отложенная задача может никогда не выполниться, если не будет вызвано `get` или `wait`.
- Возвращает `std::future<T>`, который позволяет получить возвращаемое значение.
- Позволяет обрабатывать исключения.

Асинхронное выполнение - вариант 3

Функция является корутиной, если выполнено одно из следующих условий:

- Использует оператор `co_await` для приостановки исполнения

```
task<> tcp_echo_server() {
  char data[1024];
  while (true) {
    size_t n = co_await
        socket.async_read_some(buffer(data));
    co_await async_write(socket, buffer(data, n));
  }
}
```

Асинхронное выполнение - вариант 3

Функция является корутиной, если выполнено одно из следующих условий:

- Использует ключевое слово `co_return` для завершения исполнения и возврата результата.

```
lazy<int> f() { co_return 7; }
```

- Использует ключевое слово `co_yield` для приостановки исполнения и возврата результата.

```
generator<int> iota(int n = 0) {
  while(true)
    co_yield n++;
}
```

Асинхронное выполнение - вариант 3

Пример с использованием библиотеки cppcoro:

```
#include <cppcoro/generator.hpp>
#include <iostream>

cppcoro::generator<int> iota(int n = 0) {
  while(true)
    co_yield n++;
}

void usage()
{
  for (auto i : iota(1000))
  {
    std::cout << i << std::endl;
  }
}
```

Синхронизация: mutex

```
double shared = 0; std::mutex mtx;

void compute(int begin, int end)  {
    for (int i = begin; i != end; ++i) {
        double current = someFunction(i);
        // critical section
        std::scoped_lock lock(mtx);
        shared += current;
    }
}

int main() {
  std::thread th1 (compute, 0,   100);
  std::thread th2 (compute, 100, 200);
  th1.join(); th2.join();
  std::cout << shared << std::endl;
}
```

Синхронизация: semaphore

```
std::binary_semaphore mainToThread(0), threadToMain(0);
void worker() {
        mainToThread.acquire(); // wait
        std::cout << "[thread] Got the signal\n";
        using namespace std::literals;
        std::this_thread::sleep_for(3s);
        std::cout << "[thread] Send the signal\n";
        threadToMain.release(); // send
}
int main() {
        std::jthread thrWorker(worker);
        std::cout << "[main] Send the signal\n";
        mainToThread.release(); // send
        threadToMain.acquire(); // wait
        std::cout << "[main] Got the signal\n";
}
```

Синхронизация: semaphore

```
[main] Send the signal
[thread] Got the signal
[thread] Send the signal
[main] Got the signal
```

std::atomic

- Шаблон `std::atomic` позволяет определить переменную, операции с которой будут атомарны.
- Определён только для целочисленных встроенных типов и указателей.

```
template<class T>
struct shared_ptr_data {
    void addref() {
        ++counter; // atomic increment
    }

    T * ptr;
    std::atomic<size_t> counter;
};
```

Синхронизация: lock-free

```
template<class T>
struct Node {
    T data;
    Node * next;
    Node(const T& data)
    : data(data)
    , next(nullptr) {}
};

template<class T>
class Stack {
    std::atomic<node<T>*> head;
public:
    void push(const T& data);
};
```

Синхронизация: lock-free

```
template<class T>
void Stack<T>::push(const T& data)
{
    Noe<T>* new_node = new node<T>(data);
    new_node->next = head.load(std::memory_order_relaxed)
    while(!std::atomic_compare_exchange_weak_explicit(
        &head,
        &new_node->next,
        new_node,
        std::memory_order_release,
        std::memory_order_relaxed))
    ; // the body of the loop is empty
}
```

Синхронизация: lock-free

```
int main() {
    Stack<int> s;
    s.push(1);
    s.push(2);
    s.push(3);
}
```

Общие советы и замечания

- При использовании `std::thread` следите за тем, чтобы исключения не покидали функцию потока.
- Предпочитайте `std::async` прямому созданию потоков*.
- Критические секции должны быть минимальными.
- Используйте `std::atomic` вместо мьютекса, когда синхронизация нужна только для одной целочисленной переменной.
- Помните про thread-санитайзер.
- `volatile` — это не про многопоточность.

***Полезные замечания:**
Сергей Видюк, Нестандартный future/promise