Многопоточное программирование на C++

Александр Смаль

CS центр
17 апреля 2018
Санкт-Петербург

Асинхронное выполнение

Пусть мы хотим вычислить doAsyncWork асинхронно.

```
int doAsyncWork();
```

В C++ есть два способа выполнения задач асинхронно:

- создать поток вручную `std::thread`,

```
#include <thread>
// создание потока, вычисляющего doAsyncWork()
std::thread t(doAsyncWork);
```

- использование `std::async`.

```
#include <future>
// использование std::async
std::future<int> fut = std::async(doAsyncWork);
int res = fut.get();
```

`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) выполнение.
  1. `std::launch::async`
  2. `std::launch::deferred`
- По умолчанию имеет стратегию: `std::launch::async | std::launch::deferred`

```
// гарантирует асинхронное выполнение
std::future<int> fut =
    std::async(std::launch::async, doAsyncWork);
int res = fut.get();
```

- 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.

std::thread

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

```
// переменная для возвращаемого значения
int res = 0;
std::thread t([&res](){res = doAsyncWork();});
t.join();
```

- 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.
- `native_handle()` возвращает дескриптор потока.

Синхронизация

```
double shared = 0;       // разделяемая переменная
std::mutex mtx;          // мьютекс для shared

void compute(int begin, int end)  {
    for (int i = begin; i != end; ++i) {
        double current = someFunction(i);
        // критическая секция
        std::lock_guard<std::mutex> lck(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;
}
```

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;
};
```

Общие советы и замечания

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения (в т.ч. при возникновении исключений).

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения (в т.ч. при возникновении исключений).
- При использовании `std::thread` следите за тем, чтобы исключения не покидали функцию потока.

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения (в т.ч. при возникновении исключений).
- При использовании `std::thread` следите за тем, чтобы исключения не покидали функцию потока.
- Используйте `std::atomic` вместо мьютекса, когда синхронизация нужна только для одной целочисленной переменной.

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения (в т.ч. при возникновении исключений).
- При использовании `std::thread` следите за тем, чтобы исключения не покидали функцию потока.
- Используйте `std::atomic` вместо мьютекса, когда синхронизация нужна только для одной целочисленной переменной.
- Делайте константные методы безопасными в смысле потоков (например, при кешировании).

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения (в т.ч. при возникновении исключений).
- При использовании `std::thread` следите за тем, чтобы исключения не покидали функцию потока.
- Используйте `std::atomic` вместо мьютекса, когда синхронизация нужна только для одной целочисленной переменной.
- Делайте константные методы безопасными в смысле потоков (например, при кешировании).
- `volatile` — это не про многопоточность.