# Лекция 7. Синхронизация параллельных операций

ИУ8

October 27, 2016

# Содержание

## На текущей лекции рассмотрим:

- std::async
- std::future
- std::promise
- std::packaged_task
- std::condition_variable

## Высокоуровневый интерфейс

Своеобразный "конкурент" классу std::thread составляют функция async и класс future<T>

Функция **async** обеспечивает интерфейс для вызываемого объекта.

Класс future<T> позволяет ожидать завершения потока и предоставляет доступ к его результату: возвращаемому значению или исключению.

## Использование async

```
1 #include <future>
2 int doSomething();
3
4 future<int> f = std::async(std::launch::async,
5                 doSomething);
6 // ...
7 std::cout << f.get() << std::endl;
```

## Стратегии запуска std::async

- async
- deferred
- async|deferred (по умолчанию)

## async

Создает новый поток, в котором выполняется функция, переданая вторым параметров в std::async

## deferred

Отложенный запуск функции, переданой вторым параметров в std::async *ленивое выполнение*. Функция выполняется при вызове методов get или wait соответствующий объектов future

## async|deferred

Значение по умолчанию. Как именно будет запущена задача зависит от компилятора.

## Example launch::async

```
int getting_data() {
  std::this_thread::sleep_for(std::chrono::seconds(10));
  return 42;
}

// launch getting_data here
std::future<int> f = std::async(std::launch::async, getting_data);
std::this_thread::sleep_for(std::chrono::seconds(5));
std::cout << f.get() << std::endl;
// total time is about 10s
```

## Example launch::deferred

```
int getting_data() {
  std::this_thread::sleep_for(std::chrono::seconds(10));
  return 42;
}

std::future<int> f = std::async(std::launch::deferred, getting_data);
std::this_thread::sleep_for(std::chrono::seconds(5));
std::cout << f.get() << std::endl; // actually function is launched here
// total time is about 15s
```

## std::future::get

```
1 auto f = std::async(getting_data);
```

При вызове функции std::future::get() могут произойти три события:

1. Если выполнение функции getting_data было начато async в отдельном потоке и уже закончилось, то результат получится немедленно
2. Если выполнение функции getting_data было начато async в отдельном потоке, но еще не закончилось, то функция get() блокирует поток и ждет, пока выполнение функции getting_data будет закончено, чтобы получить результат
3. Если выполнение функции getting_data еще не начиналось, то она начнет выполняться как обычная синхронная функция.

see 03.cpp

## 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.

```
1 auto f = std::async([](){
2   throw 42;
3 });
4
5 try{
6   f.get();
7 } catch(int i) {
8   std::cout << i;
9 }
```

see 04.cpp

## std::shared_future

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

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

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

see 03a.cpp

## std::promise

Рассмотрим, как можно передавать параметры и обработку исключений между потоками. Это позволит понять, как реализован высокоуровневый интерфейс.

Для передачи результатов и исключений в качестве результата выполнения потока можно использовать класс std::promise.

В то время как std::future позволяет извлечь данные, std::promise позволяет установить данные

```
1 void summ(int a, int b, std::promise<int>& accumulate_promise)
2 {
3   accumulate_promise.set_value(a + b);
4 }
5
6 std::promise<int> pr;
7 auto f = pr.get_future();
8 std::thread t1(summ, 10, 11, std::ref(pr));
```

see 05.cpp

## std::packeged_task

Функция async предоставляет разработчику инструмент для работы с результатом выполнения задачи, которую он пытается запустить немедленно (или отложенно, но синхронно) запустить в фоновом режиме. Но возникает необходимость обработать результат фоновой задачи, которую не обязательно запускать немедленно.

Таким образом, как и когда запускать задачу, решает другой класс, например, пул потоков. Тем самым программист абстрагирует специфику задачи - а планировщик имеет дело только с экземплярами std::packeged_task

```
1 std::packaged_task<double(double)>
2   pt([](double d) -> double {
3       return d * d; });
4
5 auto f = pt.get_future();
6 std::thread(std::move(pt), 10.).detach();
7 std::cout << f.get() << std::endl;
```

see 08.cpp

## std::condition_variable

Иногда задачи, выполняемые в разных потоках, должны ожидать друг друга.

Один из механизмов, который можно использовать для реализации такого поведения, - это std::future (see 06.cpp). Однако, std::future может передавать сигнал от потока другому только один раз.

Для синхронизации логических зависимостей, которыми можно многократно обмениваться между потоками, можно использовать **условные переменные** (std::condition_variable)

Условные переменные предоставляют самый простой механизм ожидания события, возникающего в другом потоке.

Стандартная библиотека C++ предлагает две реализации условный переменных std::condition_variable и std::condition_variable_any

Первый класс работает только с std::mutex, второй - с любым классом, который отвечает минимальным требованиям "мьютексоподобия". Т.к std::condition_variable_any более общий, то его использование обойдется дороже с точки зрения потребляемой памяти, производительности и ресурсов ОС.

```
std::mutex m;
std::string str;
std::condition_variable cv;
```

```
void waiting_string() {
  std::unique_lock<std::mutex> lk
       (m);
  cv.wait(lk, [](){
    return !str.empty();
  });
  // use str
}
```

```
void read_string() {
  std::lock_guard<std::mutex> lk(
       m);
  str = read_data();
  cv.notify_one(); // data is
       ready
}
```

see 01.cpp, 02.cpp

## Пояснения к примеру

В примере есть два потока: функция waiting_string ожидает пока будет получена строка, функция read_string заполняет строку и сообщает функции waiting_string, что строка готова для использования.

Функция read_string захватывает мьютекс необходимый для защиты данных. Затем производит модификацию разделяемых данных. После извещает ожидающий поток, используя метод notify_one

В waiting_string самым интересным является метод wait. Эта функция проверяет условие, вызывая второй аргумент.

- Если условие возвращает true - фукция возвращает управление.
- Если же условие не выполнено, то wait освобождает мьютекс и переводит поток в состояние ожидания. Когда условная переменная получает уведомление, поток обработки пробудится, вновь захватит мьютекс и проверит условие. Если условие выполнено, то wait вернет управление, при чем мьютекс будет захвачен. Если условие не выполнено, то поток опять переходит в состояние ожидания.

## Нюансы

Метод wait захватывает и освобождает мьютекс, поэтому требует для своей работы именно unique_lock, а не lock_guard.

Внутри wait условная переменная может проверять условие многократно, но каждый раз это делается после захвата мьютекса.

Если функция проверки условия вернет true, то wait возвращает управление вызывающей программе.

Ситуация, когда ожидающий поток проверяет условие не в ответ на извещение от другого потока, называется **ложным пробуждением**. Количество и частота ложных пробуждений недетерменированы.

## В качестве заключения

Мы рассмотрели средства для синхронизации и параллельных операций. Эти строительные блоки можно и надо использовать для реализации своих многопоточных приложениях. В современных системах ошибочно использовать всего один поток, одно ядро, один процессор для работы программного обеспечения.
Кроме этого с помощью этих блоков можно реализовывать еще более продвинутые системы: пул потоков, потоки с прерыванием выполнения. Эта тема будет рассмотрена на следующих лекциях.

# The end

## Самостоятельное изучение

- std::chrono

## Список литературы


- Джосаттис - Стандартная библиотека C++. 2-е издание
- Уильямс - Параллельное программирование на C++ в действии