Лекция 8. Управление потоками

ИУ8

October 24, 2017

Содержание

На текущей лекции рассмотрим:

- std::thread
- std::async
- std::future

Параллелизм

Определение

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

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.

Параллелизм

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.

Другая причина использования параллелизма - повышение производительности. Сейчас существуют многопроцессорные ЭВМ с 16 и более ядрами на каждом кристалле. При этом использование только одного потока для выполнения всех задач является ошибкой.

Параллелизм

Существует два способа применить распараллеливание для повышения производительности: распараллеливание по задачам и распараллеливание по данным

Распараллеливание по задачам

В случае распараллеливания по задачам задача разбивается на части (подзадачи), которые запускаются параллельно, тем самым уменьшая общее время выполнения.

Распараллеливание по данным

В случае распараллеливания по данным каждый поток выполняет одну и ту же операцию, но с разными данными.

Параллелизм

Стандарт C++11

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

Кроме того, в новом стандарте определена совершенно новая модель памяти с поддержкой многопоточности

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 разработчики при проектировании библиотеки многопоточности, была минимизация платы за абстрагирование.

Еще одна задача, стоящая перед комитетом стандартизации, - это предоставление низкоуровневых средств для работы на уровне "железа".

std::thread

Hello, World!

```
1 #include <iostream>
2 #include <thread>
3 int main() {
4   std::thread([](){
5     std::cout << "Hello, World!";
6   }).join();
7 }
```

Будь решителен

Программист обязан гарантировать, что поток корректно будет "присоединен" либо "отсоединен" до вызова деструктора объекта std::thread

```
~thread() {
  if(joinable())
    std::terminate();
}
```

Поэтому после запуска потока необходимо решить, ждать его завершения (присоединившись к нему) или предоставить ему возможность выполняться независимо и самостоятельно (отсоединив его)

Решение следует принять именно до уничтожения объекта std::thread, к самому потоку оно (решение) не имеет никакого отношения.

std::thread

.join

Чтобы дождаться завершения потока, следует вызвать функцию join. Вызов join очищает всю ассоциированную с потоком память; это значит, что для каждого потока вызвать функцию join можно только один раз. Если требуется более гибкий контроль над ожиданием потока (ждать ограниченное время или проверить, завершился ли поток), то следует прибегать к другим средствам.

.detach

Вызов функции detach оставляет поток работать в фоновом режиме. Отсоединенные потоки работают в фоне, и за их управление отвечает библиотека времени выполнения C++.

Ожидаем завершения

```
1 std::thread([](){
2   std::cout << "Hello, World!";})
3   .join();
```

Отсоединяем поток

```
1 std::thread([](){
2   std::cout << "Hello, World!";})
3   .detach();
```

Передача аргументов

```
1 void foo(const std::string & s);
2 std::thread(foo, "hello").join();
```

Из примера видно, что передача аргументов в функцию сводится к передаче дополнительных параметров конструктору std::thread. Однако следует учитывать, что эти аргументы **копируются** в память объекта, где они доступны созданному потоку. *Причем так происходит даже в том случае, когда функция ожидает на месте соответствующего параметра ссылку*

```
1 void foo(std::string & s) { s += s;}
2 std::string s = "foo";
3 std::thread(foo, s).join();
4 std::cout << s; // foo
5 std::thread(foo, std::ref(s)).join();
6 std::cout << s; // foofoo
```

Передача владения потоком

Класс std::thread является перемещающимся типом, т.е. для него определены оператор перемещения и конструктор перемещения, но не определены функции копирования.

Это позволяет передавать владение потоком "из рук в руки", например, возвращать из функции или передавать в функцию в качестве аргумента.

Кроме того, это позволяет хранить объекты типа std::thread в стандартных контейнерах.

std::async

Высокоуровневый интерфейс

Своеобразными "конкурентами" классу std::thread являются 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;
```

std::async

Стратегии запуска std::async

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

async

Создает новый поток, в котором выполняется функция, переданная вторым параметром в std::async

deferred

Отложенный запуск функции, переданной вторым параметром в std::async *ленивое выполнение*. Функция выполняется при вызове методов get или wait соответствующего объекта future

async|deferred

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

std::async

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::lanch::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 08.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 09.cpp

std::future

std::shared_future

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

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 08a.cpp

The end

Самостоятельное изучение

- thread_local
- call_once
- инициализация статичных переменных в многопоточных приложениях

Список литературы

- Уильямс - Параллельное программирование на C++ в действии
- Джосаттис - Стандартная библиотека C++. 2-е издание
- http://www.bogotobogo.com/cplusplus/C11/1_C11_creating_thread.p
- https://www.tutorialcup.com/cplusplus/multithreading.htm