Лекция 7. Управление потоками

ИУ8

October 21, 2016

Содержание

На текущей лекции рассмотрим:

- thread
- mutex

Параллелизм

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.

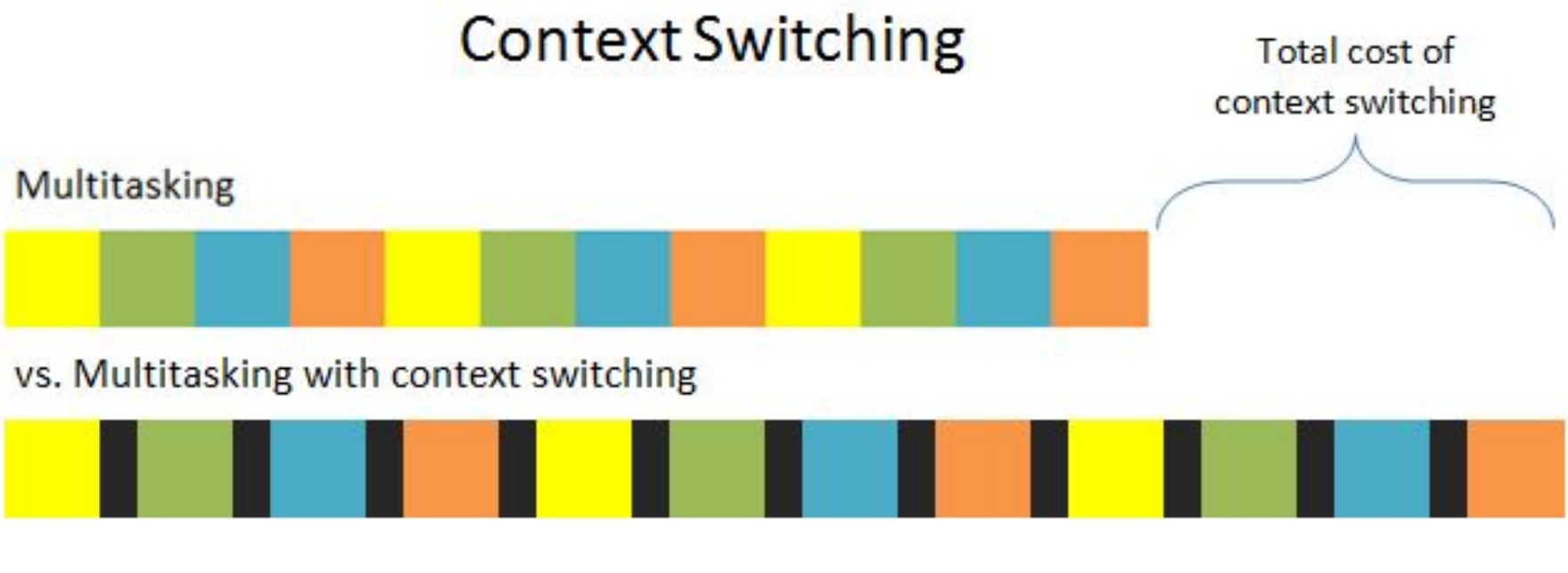
Оборудование с несколькими процессорами (или несколькими ядрами на одном процессоре) может выполнять несколько задач одновременно. Это называется аппаратным параллелизмом.

Переключение контекста

В системе количество задач может превышать число ядер, тогда будет применяться механизм переключения задач.

В произвольный момент времени ОС передает процессору на исполнение поток. Этот поток исполняется в течение некоторого временного интервала. После завершения этого интервала контекст ОС переключается на другой поток. При этом происходит следующее:

1. Значения регистров процессора исполняющегося в данный момент потока сохраняются в структуре контекста, которая располагается в ядре потока
2. Из набора имеющихся потоков выделяется тот, которому будет передано управление.
3. Значения из выбранной структуры контекста потока загружаются в регистры процессора.


Context Switching
Total cost of
context switching
Multitasking
vs. Multitasking with context switching

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 всём этом использование всего одного потока для выполнения своих задач является ошибкой.

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.

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 была 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".

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 явно 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.

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.

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

Ни функцию join, ни функцию detach нельзя вызвать для объектов, с которыми не связан никакой поток. Чтобы определить связан ли с объектом поток, используйте функцию std::thread::joinable

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
```

Вызов метода класса

```
1  struct Object {
2    void print() {
3      std::cout << "blablabla" << std::endl;
4    }
5  };
6
7  int main() {
8    Object o;
9    std::thread(
10       &Object::print, // pass a pointer to the method that want to call
11       o // pass a copy of the object for which you want to invoke a method
12   ).join();
13 }
```

Подробности смотри в 04.cpp

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.

Идентификация потоков

Идентификатор потока имеет тип std::thread::id. Получить его можно двумя способами:

- вызвать функцию std::this_thread::get_id()
- вызвать метод get_id() у объекта типа std::thread

Если с объектом std::thread не связан никакой поток, то будет возвращен std::thread::id, созданный конструктором по умолчанию.

Для объектов типа std::thread::id задано отношения полного порядка. Это позволяет использовать этот тип в качестве ключей ассоциативных контейнеров, сортировать и сравнивать любым интересующим способом.

Data race

В параллельном программировании под состоянием гонки понимается любая ситуация, исход которой зависит от относительного порядка выполнения операций в более чем в одном потоке

Гонки приводят к ошибкам в случае, если они (гонки) приводят к нарушению инвариантов.

Инвариант - утверждение о структуре данных, которое всегда должно быть истинным.

В стандарте C++ определен термин гонка за данными (data race), означающий ситуацию, когда гонка возникает при одновременной модификации одного объекта.

Устранение состояний гонок

Чтобы избавиться от проблематичных гонок, структуру данных можно снабдить неким защитным механизмом, который гарантирует, что только один поток может видеть промежуточные состояния, когда нарушены инварианты.

Другой способ избежать проблем - изменить дизайн структуры данных и её инварианты так, чтобы модификация представляла собой последовательность неделимых изменений, каждое из которых сохраняет инварианты. Такой подход называется программированием без блокировок.

Mutual exclusion

Взаимоисключающая блокировка - простейший сбособ защиты разделяемых данных.

Мьютексы - наиболее общий механизм защиты данных с C++.

```
1 class mutex{
2 public:
3     void lock();
4     bool try_lock();
5     void unlock();
6     native_handle_type native_handle();
7 };
```

Работа с std::mutex

- для захвата мьютекса служит функция lock();
- для освобождения - unlock().

Но следует помнить, что необходимо освобождать мьютекс на каждом пути выхода из функции, в том числе и при исключениях

Для этого следует использовать идиому RAII. Именно с этой целью в стандарт внесен класс std::lock_guard.

Пример 10.cpp

Ассоциируйте мьютекс с защищаемыми данными

В соответствии с правилами ООП, мьютекс и защищаемые данные принято помещать в один класс.

Тогда кажется, что если все методы такого класса будут перед обращением к данным захватывать мьютекс, то данные остаются защищенными

Но это ложное чувство безопасности. Если какой-либо метод возвращает указатели или ссылку на защищаемые данные, то уже не важно, правильно ли реализовано управлением мьютексом.

Любой код, имеющий доступ к этому указателю, может модифицировать или прочитать защищенные данные, не захватывая мьютекс.

Dead lock

Взаимная блокировка - ситуация, когда два или более потока ожидают завершения друг друга.

Например, такая ситуация возможна, когда для выполнения процедуры потоки должны захватить два мьютекса, но по каким-то причинам каждый поток захватил только по одному мьютексу.

Общая рекомендация - всегда захватывать мьютексы в одном порядке.

std::lock

В стандартной библиотеке есть функция, которая захватывает сразу несколько мьютексов - std::lock.

Функция std::lock обеспечивает семантику "всё или ничего".

Если std::lock успешно захватила первый мьютекс, но во время попытки захвата второго мьютекса произошло исключение, то первый мьютекс освобождается.

```
1 std::mutex m_a;
2 std::mutex m_b;
3
4 std::lock(m_a, m_b);
5 std::lock_guard<std::mutex> lk_a(m_a, std::adopt_lock);
6 std::lock_guard<std::mutex> lk_b(m_b, std::adopt_lock);
```

Рекомендации, как избежать взаимоблокировки

- Избегать вложенных блокировок. Не захватывайте мьютекс, если уже захватили другой
- Не вызывать пользовательский код, когда удерживаете мьютекс.
- Захватывать мьютексы в фиксированном порядке.
- Использовать иерархию блокировок. Идея состоит в том, чтобы каждому мьютексу присвоить численное значение уровня, и позволять захватывать потоку только мьютексы с большим значением. Тем самым обеспечивается строгий порядок захвата мьютексов.

std::unique_lock

Класс std::unique_lock обладает большей гибкостью, чем std::lock_guard.

Во-первых std::unique_lock реализует семантику перемещения.

Во-вторых std::unique_lock позволяет управлять ассоциированным с ним мьютексом. Т.е. у std::unique_lock есть методы lock, try_lock, unlock.

Так как std::unique_lock является lockable объектом, то его можно передавать в функцию std::lock

shared_mutex

Если несколько потоков только считывают данные и не модифицируют их, то гонок за данными не возникает.

Поэтому разумно предоставлять доступ для чтения к разделяемым данным нескольким потоком одновременно. Если же какой-то поток пытается модифицировать данные, то ему следует предоставлять монопольный доступ.

Для такой ситуации в библиотеке boost существует класс shared_mutex

```
1  boost::shared_mutex m;
2  T read() {
3    // many threads can read data, so use shared_lock
4    boost::shared_lock<boost::shared_mutex> lk(m);
5    // get data
6  }
7  void modify(){
8    // only one thread can write data, so use lock_guard
9    std::lock_guard<boost::shared_mutex> lk(_m);
10   // modify data
11 }
```

The end

Самостоятельное изучение

- thread_local
- call_once
- инициализация статичных переменных в многопоточных приложениях
- std::recursive_mutex

Список литературы

- Уильямс - Параллельное программирование на C++ в действии
- C++11/C++14 Thread 1. Creating Threads
- Multithreading in C++