

Лекция 6. Синхронизация потоков

ИУ8

November 13, 2018

На текущей лекции рассмотрим:

- различные виды блокировок
- шаблон consumer-producer
- условные переменные
- атомарные операции

Задача

Пусть есть одна очередь. И есть два потока, которые добавляют в эту очередь данные.

Data race

```
1 template<class T>
2 struct Queue {
3     struct Node {
4         T data;
5         Node * next;
6         // ...
7     };
8     Node * First;
9     Node * Last;
10
11     void push(const T& data) {
12         if (!empty()) {
13             Node * newNode = new Node();
14             newNode->data = data;
15             Last->next = newNode;
16             Last = newNode;
17         } else {
18             // ...
19         }
20     }
21 };
```

```
1 Queue<std::string> queue;
2 //...
3 void foo(const std::string& filename) {
4     auto fl = open_file(filename);
5     while(fl) {
6         queue.push(fl.read_line());
7     }
8 }
9
10 int main() {
11     std::thread t1(foo, "one.txt");
12     std::thread t2(foo, "one.txt");
13
14     t1.join();
15     t2.join();
16
17     // ...
18 }
```

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

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 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 возникновения гонки при модификации объекта несколькими сторонами одновременно.

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 при нарушении инвариантов.

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 при нарушении инвариантов.

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 способ защиты разделяемых данных.

Мьютексы - наиболее общий механизм защиты данных в 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()`.

Работа с `std::mutex`

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

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

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 внесён класс `std::lock_guard`.

std::lock_guard

```
1 explicit lock_guard( mutex_type& m );  
2 lock_guard( mutex_type& m, std::adopt_lock_t t );  
3 lock_guard( const lock_guard& ) = delete;
```

Конструктор с двумя аргументами принимает во владение мьютекс не пытаясь его закатить. Если мьютекс не принадлежит текущему потоку, то поведение не определено.

`std::unique_lock`

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

`std::unique_lock`

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

Во-первых, `std::unique_lock` реализует семантику перемещения.

`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`.

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.

```
1 unique_lock( mutex_type& m, std::defer_lock_t t ) noexcept;  
2 unique_lock( mutex_type& m, std::try_to_lock_t t );  
3 unique_lock( mutex_type& m, std::adopt_lock_t t );
```

- конструктор с defer_lock_t не захватывает мьютекс
- конструктор с try_to_lock_t пытается захватить мьютекс с помощью функции try_lock()
- конструктор с adopt_lock_t используется в случае, если мьютекс уже захвачен

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.

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 монополярный доступ.

Для такой ситуации существует класс `shared_mutex`

```
1 // C++17 or boost::shared_mutex
2 std::shared_mutex m;
3 T read() {
4     // many threads can read data, so use shared_lock
5     std::shared_lock<std::shared_mutex> lk(m);
6     // get data
7 }
8 void modify(){
9     // only one thread can write data, so use lock_guard
10    std::lock_guard<std::shared_mutex> lk(m);
11    // modify data
12 }
```

Пример

```
1 template <class T>
2 class Queue {
3     std::queue<T> Data;
4     std::mutex Mutex;
5
6 public:
7     void Push(const T& item) {
8         std::lock_guard<std::mutex> lk(Mutex);
9         Data.push(item);
10    }
11    // ...
12};
```

Dead lock

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

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`.

`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.

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

Рекомендации, как избежать взаимоблокировки

- Избегать вложенных блокировок. Не захватывайте мьютекс, если уже захватили другой
- Не вызывать пользовательский код, когда удерживаете мьютекс.

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.

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.
- Попробуйте свести задачу к однопоточному режиму работы.

Задача

Существует N потоков, которые получают данные, и M потоков, которые обрабатывают полученные данные. Требуется реализовать механизм взаимодействия между этими потоками.

`std::condition_variable`

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

`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`. Однако `std::future` может передавать сигнал от одного потока другому только один раз.

`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`. Однако `std::future` может передавать сигнал от одного 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.

Consumer-producer

```
1 std::mutex m;  
2 std::string str;  
3 std::condition_variable cv;
```

```
1 // producer  
2 void read_string() {  
3     std::lock_guard<std::mutex> lk(  
4         m);  
5     str = read_data();  
6     cv.notify_one(); // data is  
7     ready  
8 }
```

```
1 // consumer  
2 void waiting_string() {  
3     std::unique_lock<std::mutex> lk  
4         (m);  
5     cv.wait(lk, [](){  
6         return !str.empty();  
7     });  
8     // use str  
9 }
```

see 02.cpp, 03.cpp

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.

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.

Producer

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

Consumer

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

- Если условие возвращает `true` - функция `wait` возвращает управление. Функция `waiting_string` продолжит свое выполнение.
- Если же условие не выполнено, то `wait` освобождает мьютекс и переводит поток в состояние ожидания. Когда условная переменная получает уведомление, поток обработки возобновится, снова захватит мьютекс и проверит условие. Если условие выполнено, то `wait` вернет управление, причем 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`.

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.

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.

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 **недетерминированы**.

Стандартная библиотека C++ предлагает две реализации условных 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.

Атомарные объекты

Появление многопоточности вносит сложность с загрузкой и сохранением данных.

Атомарные объекты

Появление многопоточности вносит сложность с загрузкой и сохранением данных.

Не любая структура данных может быть загружена/сохранена в память одной процессорной операцией. Более того, то, что может быть загружено/сохранено в память одной операцией на одной архитектуре процессора, не может быть на другой

Атомарные объекты

Появление многопоточности вносит сложность с загрузкой и сохранением данных.

Не любая структура данных может быть загружена/сохранена в память одной процессорной операцией. Более того, то, что может быть загружено/сохранено в память одной операцией на одной архитектуре процессора, не может быть на другой

Атомарный объект – это такой объект, операции над которым можно считать неделимыми, т.е. такими, которые не могут быть прерваны или результат которых не может быть получен до окончания операции.

Атомарные объекты в C++11

Когда один поток сохраняет данные в объекте атомарного типа, а другой хочет их прочесть, поведение программы определено стандартом, в отличие от ситуаций, когда используются не атомарные типы.

Атомарные объекты в C++11

Когда один поток сохраняет данные в объекте атомарного типа, а другой хочет их прочесть, поведение программы определено стандартом, в отличие от ситуаций, когда используются не атомарные типы.

В C++11 появилось два типа атомарных объектов: `std::atomic<T>` и `std::atomic_flag`

Эффективность atomic vs mutex

Конечно же, атомарные операции можно реализовать с помощью мьютекса.

Недостатком такой реализации является вероятное снижение эффективности, так как захват и освобождение мьютекса не самые простые операции.

Эффективность atomic vs mutex

Конечно же, атомарные операции можно реализовать с помощью мьютекса.

Недостатком такой реализации является вероятное снижение эффективности, так как захват и освобождение мьютекса не самые простые операции.

Атомарные объекты **могут** быть гораздо эффективнее.

Эффективность atomic vs mutex

Конечно же, атомарные операции можно реализовать с помощью мьютекса.

Недостатком такой реализации является вероятное снижение эффективности, так как захват и освобождение мьютекса не самые простые операции.

Атомарные объекты **могут** быть гораздо эффективнее.

Атомарный доступ может понадобиться к структуре любой сложности и размера, но настоящей атомарной операцией над данными можно считать лишь ту, которую процессор определенной архитектуры может выполнить одной командой.

Методы `std::atomic<T>`

- `store` – Кладет новое значение в объект.
- `load` – Извлекает значение из объекта.
- `exchange` – Заменяет значение в объекте на новое и возвращает старое.
- `compare_exchange_strong(expected, desired)` – Если объект равен `expected`, тогда `desired` помещается в объект. В противном случае объект помещается в `expected`.
- `compare_exchange_weak(expected, desired)` – Если объект равен `expected`, тогда `desired` помещается в объект. В противном случае объект помещается в `expected`.

```
std::atomic<integral>
```

Отличительной особенностью этой версии является наличие дополнительных операций, которые можно осуществлять с атомарным интегральным объектом: `fetch_add`, `fetch_sub`, `fetch_and`, etc.

```
std::atomic<integral>
```

Отличительной особенностью этой версии является наличие дополнительных операций, которые можно осуществлять с атомарным интегральным объектом: `fetch_add`, `fetch_sub`, `fetch_and`, etc.

```
std::atomic<T*>
```

Этот тип используется для всех указателей, работа с которыми должна быть атомарна. В этом типе есть оператор разыменовывания указателя.

Заключение

Мы рассмотрели средства для синхронизации параллельных операций. Эти строительные блоки можно и необходимо использовать в своих многопоточных приложениях.

Также с помощью этих блоков можно реализовывать более продвинутые системы: пул потоков, потоки с прерыванием выполнения и модели и шаблоны, используемые в многопоточном программировании.