В первой части этой статьи основное внимание будет уделено потокам и блокировкам в С++11, условные переменные во всей своей красе будут подробно рассмотрены во [второй части](http://habrahabr.ru/post/182626/)… 

Потоки

В C++11, работа с потокам осуществляется по средствам класса std::thread (доступного из заголовочного файла <thread>), который может работать с регулярными функциями, лямбдами и функторами. Кроме того, он позволяет вам передавать любое число параметров в функцию потока.

**#include <thread>**

**void** **threadFunction**()

{

*// do smth*

}

**int** **main**()

{

std::thread **thr**(threadFunction);

thr.join();

**return** 0;

}

В этом примере, thr — это объект, представляющий поток, в котором будет выполняться функция threadFunction(). Вызов join блокирует вызывающий поток (в нашем случае — поток main) до тех пор, пока thr (а точнее threadFunction()) не выполнит свою работу. Если функция потока возвращает значение — оно будет проигнорировано. Однако принять функция может любое количество параметров.

**void** **threadFunction**(**int** i, **double** d, **const** std::string &s)

{

std::cout << i << ", " << d << ", " << s << std::endl;

}

**int** **main**()

{

std::thread **thr**(threadFunction, 1, 2.34, "example");

thr.join();

**return** 0;

}

Несмотря на то, что передавать можно любое число параметров, все они были переданы по значению Если в функцию необходимо передать параметры по ссылке, они должны быть обернуты в std::ref или std::cref, как в примере:

**void** **threadFunction**(**int** &a)

{

a++;

}

**int** **main**()

{

**int** a = 1;

std::thread **thr**(threadFunction, std::ref(a));

thr.join();

std::cout << a << std::endl;

**return** 0;

}

Программа напечатает в консоль 2. Если не использовать std::ref, то результатом работы программы будет 1.  
  
Помимо метода join, следует рассмотреть еще один, похожий метод — detach.   
detach позволяет отсоединить поток от объекта, иными словами, сделать его фоновым. К отсоединенным потокам больше нельзя применять join.

**int** **main**()

{

std::thread **thr**(threadFunction);

thr.detach();

**return** 0;

}

Также следует отметить, что если функция потока кидает исключение, то оно не будет поймано try-catch блоком. Т.е. следующий код не будет работать (точнее работать то будет, но не так как было задумано: без перехвата исключений):

**try**

{

std::thread **thr1**(threadFunction);

std::thread **thr2**(threadFunction);

thr1.join();

thr2.join();

}

**catch** (**const** std::exception &ex)

{

std::cout << ex.what() << std::endl;

}

Для передачи исключений между потоками, необходимо ловить их в функции потока и хранить их где-то, чтобы, в дальнейшем, получить к ним доступ.

std::mutex g\_mutex;

std::vector<std::exception\_ptr> g\_exceptions;

**void** **throw\_function**()

{

**throw** std::exception("something wrong happened");

}

**void** **threadFunction**()

{

**try**

{

throw\_function();

}

**catch** (...)

{

std::lock\_guard<std::mutex> lock(g\_mutex);

g\_exceptions.push\_back(std::current\_exception());

}

}

**int** **main**()

{

g\_exceptions.clear();

std::thread **thr**(threadFunction);

thr.join();

**for**(**auto** &e: g\_exceptions)

{

**try**

{

**if**(e != nullptr)

std::rethrow\_exception(e);

}

**catch** (**const** std::exception &e)

{

std::cout << e.what() << std::endl;

}

}

**return** 0;

}

Прежде, чем двигаться дальше, хочу отметить некоторые полезные функции, предоставляемые <thread>, в пространстве имен std::this\_thread: 

* [get\_id](http://en.cppreference.com/w/cpp/thread/get_id): возвращает id текущего потока
* [yield](http://en.cppreference.com/w/cpp/thread/yield): говорит планировщику выполнять другие потоки, может использоваться при активном ожидании
* [sleep\_for](http://en.cppreference.com/w/cpp/thread/sleep_for): блокирует выполнение текущего потока в течение установленного периода
* [sleep\_until](http://en.cppreference.com/w/cpp/thread/sleep_until): блокирует выполнение текущего потока, пока не будет достигнут указанный момент времени

Блокировки

В последнем примере, я должен был синхронизировать доступ к вектору g\_exceptions, чтобы быть уверенным, что только один поток одновременно может вставить новый элемент. Для этого я использовал мьютекс и блокировку на мьютекс. Мьютекс — базовый элемент синхронизации и в С++11 представлен в 4 формах в заголовочном файле <mutex>:

* [mutex](http://en.cppreference.com/w/cpp/thread/mutex): обеспечивает базовые функции [lock()](http://en.cppreference.com/w/cpp/thread/mutex/lock) и [unlock()](http://en.cppreference.com/w/cpp/thread/mutex/unlock) и не блокируемый метод [try\_lock()](http://en.cppreference.com/w/cpp/thread/mutex/try_lock)
* [recursive\_mutex](http://en.cppreference.com/w/cpp/thread/recursive_mutex): может войти «сам в себя»
* [timed\_mutex](http://en.cppreference.com/w/cpp/thread/timed_mutex): в отличие от обычного мьютекса, имеет еще два метода: [try\_lock\_for()](http://en.cppreference.com/w/cpp/thread/timed_mutex/try_lock_for) и [try\_lock\_until()](http://en.cppreference.com/w/cpp/thread/timed_mutex/try_lock_until)
* [recursive\_timed\_mutex](http://en.cppreference.com/w/cpp/thread/recursive_timed_mutex): это комбинация timed\_mutex и recursive\_mutex

Приведу пример использования std::mutex с упомянутыми ранее функциями-помощниками get\_id() и sleep\_for():

**#include <iostream>**

**#include <chrono>**

**#include <thread>**

**#include <mutex>**

std::mutex g\_lock;

**void** **threadFunction**()

{

g\_lock.lock();

std::cout << "entered thread " << std::this\_thread::get\_id() << std::endl;

std::this\_thread::sleep\_for(std::chrono::seconds(rand()%10));

std::cout << "leaving thread " << std::this\_thread::get\_id() << std::endl;

g\_lock.unlock();

}

**int** **main**()

{

srand((**unsigned** **int**)time(0));

std::thread **t1**(threadFunction);

std::thread **t2**(threadFunction);

std::thread **t3**(threadFunction);

t1.join();

t2.join();

t3.join();

**return** 0;

}

Программа должна выдавать примерно следующее:

entered thread 10144

leaving thread 10144

entered thread 4188

leaving thread 4188

entered thread 3424

leaving thread 3424

Перед обращением к общим данным, мьютекс должен быть заблокирован методом lock, а после окончания работы с общими данными — разблокирован методом unlock.  
  
Следующий пример показывает простой потокобезопасный контейнер (реализованный на базе std::vector), имеющий методы add() для добавления одного элемента и addrange() для добавления нескольких элементов.  
*Примечание*: и всё же этот контейнер не является полностью потокобезопасным по нескольким причинам, включая использование va\_args. Также, метод dump() не должен принадлежать контейнеру, а должен быть автономной функцией. Цель этого примера в том, что показать основные концепции использования мьютексов, а не не сделать полноценный, безошибочный, потокобезопасный контейнер.

**template** <**typename** T>

**class** container

{

std::mutex \_lock;

std::vector<T> \_elements;

**public**:

**void** **add**(T element)

{

\_lock.lock();

\_elements.push\_back(element);

\_lock.unlock();

}

**void** **addrange**(**int** num, ...)

{

va\_list arguments;

va\_start(arguments, num);

**for** (**int** i = 0; i < num; i++)

{

\_lock.lock();

add(va\_arg(arguments, T));

\_lock.unlock();

}

va\_end(arguments);

}

**void** **dump**()

{

\_lock.lock();

**for**(**auto** e: \_elements)

std::cout << e << std::endl;

\_lock.unlock();

}

};

**void** **threadFunction**(container<**int**> &c)

{

c.addrange(3, rand(), rand(), rand());

}

**int** **main**()

{

srand((**unsigned** **int**)time(0));

container<**int**> cntr;

std::thread **t1**(threadFunction, std::ref(cntr));

std::thread **t2**(threadFunction, std::ref(cntr));

std::thread **t3**(threadFunction, std::ref(cntr));

t1.join();

t2.join();

t3.join();

cntr.dump();

**return** 0;

}

При выполнении этой программы произойдет **deadlock** (взаимоблокировка, т.е. заблокированный поток так и останется ждать). Причиной является то, что контейнер пытается получить мьютекс несколько раз до его освобождения (вызова unlock), что невозможно. Здесь и выходит на сцену std::recursive\_mutex, который позволяет получать тот же мьютекс несколько раз. Максимальное количество получения мьютекса не определено, но если это количество будет достигно, то lock бросит исключение [std::system\_error](http://en.cppreference.com/w/cpp/error/system_error). Поэтому, решение проблемы в коде выше (кроме изменения реализации addrange(), чтобы не вызывались lock и unlock), заключается в замене мьютекса на std::recursive\_mutex.

**template** <**typename** T>

**class** container

{

std::recursive\_mutex \_lock;

*// ...*

};

Теперь, результат работы программы будет следующего вида:

6334

18467

41

6334

18467

41

6334

18467

41

Вы, наверное, заметили, что при вызове threadFunction(), генерируются одни и те же числа. Это происходит потому, что функция void srand (unsigned int seed); инициализирует seed только для потока main. В других потоках, генератор псевдо-случайных чисел не инициализируется и получаются каждый раз одни и те же числа.  
Явная блокировка и разблокировка могут привести к ошибкам, например, если вы забудете разблокировать поток или, наоборот, будет неправильный порядок блокировок — все это вызовет deadlock. Std предоставляет несколько классов и функций для решения этой проблемы.  
Классы «обертки» позволяют непротиворечиво использовать мьютекс в [RAII](http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization)-стиле с автоматической блокировкой и разблокировкой в рамках одного блока. Эти классы:

* [lock\_guard](http://en.cppreference.com/w/cpp/thread/lock_guard): когда объект создан, он пытается получить мьютекс (вызывая lock()), а когда объект уничтожен, он автоматически освобождает мьютекс (вызывая unlock())
* [unique\_lock](http://en.cppreference.com/w/cpp/thread/unique_lock): в отличие от lock\_guard, также поддерживает отложенную блокировку, временную блокировку, рекурсивную блокировку и использование условных переменных

С учетом этого, мы можем переписать класс контейнер следующим образом:

**template** <**typename** T>

**class** container

{

std::recursive\_mutex \_lock;

std::vector<T> \_elements;

**public**:

**void** **add**(T element)

{

std::lock\_guard<std::recursive\_mutex> locker(\_lock);

\_elements.push\_back(element);

}

**void** **addrange**(**int** num, ...)

{

va\_list arguments;

va\_start(arguments, num);

**for** (**int** i = 0; i < num; i++)

{

std::lock\_guard<std::recursive\_mutex> locker(\_lock);

add(va\_arg(arguments, T));

}

va\_end(arguments);

}

**void** **dump**()

{

std::lock\_guard<std::recursive\_mutex> locker(\_lock);

**for**(**auto** e: \_elements)

std::cout << e << std::endl;

}

};

Можно поспорить насчет того, что метод dump() должен быть константным, ибо не изменяет состояние контейнера. Попробуйте сделать его таковым и получите ошибку при компиляции:

‘std::lock\_guard<\_Mutex>::lock\_guard(\_Mutex &)' : cannot convert parameter 1 from ‘const std::recursive\_mutex'

to ‘std::recursive\_mutex &'

Мьютекс (не зависимо от формы реализации), должен быть получен и освобожден, а это подразумевает использование не константных методов lock() и unlock(). Таким образом, аргумент lock\_guard не может быть константой. Решение этой проблемы заключается в том, чтобы сделать мьютекс mutable, тогда спецификатор const будет игнорироваться и это позволит изменять состояние из константных функций.

**template** <**typename** T>

**class** container

{

**mutable** std::recursive\_mutex \_lock;

std::vector<T> \_elements;

**public**:

**void** **dump**() **const**

{

std::lock\_guard<std::recursive\_mutex> locker(\_lock);

**for**(**auto** e: \_elements)

std::cout << e << std::endl;

}

};

Конструкторы классов «оберток» могут принимать параметр, определяющий политику блокировки:

* defer\_lock типа defer\_lock\_t: не получать мьютекс
* try\_to\_lock типа try\_to\_lock\_t: попытаться получить мьютекс без блокировки
* adopt\_lock типа adopt\_lock\_t: предполагается, что у вызывающего потока уже есть мьютекс

Объявлены они следующим образом:

**struct** **defer\_lock\_t** { };

**struct** **try\_to\_lock\_t** { };

**struct** **adopt\_lock\_t** { };

**constexpr** std::**defer\_lock\_t** defer\_lock = std::**defer\_lock\_t**();

**constexpr** std::**try\_to\_lock\_t** try\_to\_lock = std::**try\_to\_lock\_t**();

**constexpr** std::**adopt\_lock\_t** adopt\_lock = std::**adopt\_lock\_t**();

Помимо «оберток» для мьютексов, std также предоставляет несколько методов для блокировки одного или нескольких мьютексов:

* [lock](http://en.cppreference.com/w/cpp/thread/lock): блокирует мьютекс, используя алгоритм избегания deadlock'ов (используя lock(), try\_lock() и unlock())
* [try\_lock](http://en.cppreference.com/w/cpp/thread/try_lock): пытается блокировать мьютексы в порядке, в котором они были указаны

Вот типичный пример возникновения взаимоблокировки (deadlock): у нас есть некий контейнер с элементами и функция exchange(), которая меняет местами два элемента разных контейнеров. Для потокобезопасности, функция синхронизирует доступ к этим контейнерам, получая мьютекс, связанный с каждым контейнером. 

**template** <**typename** T>

**class** container

{

**public**:

std::mutex \_lock;

std::set<T> \_elements;

**void** **add**(T element)

{

\_elements.insert(element);

}

**void** **remove**(T element)

{

\_elements.erase(element);

}

};

**void** **exchange**(container<**int**> &c1, container<**int**> &c2, **int** value)

{

c1.\_lock.lock();

std::this\_thread::sleep\_for(std::chrono::seconds(1)); *// симулируем deadlock*

c2.\_lock.lock();

c1.remove(value);

c2.add(value);

c1.\_lock.unlock();

c2.\_lock.unlock();

}

Предположим, что эта функция вызвана из двух разных потоков, из первого потока: элемент удаляется из 1 контейнера и добавляется во 2, из второго потока, наоборот, элемент удаляется из 2 контейнера и добавляется в 1. Это может вызвать deadlock (если контекст потока переключается от одного потока к другому, сразу после первой блокировки).

**int** **main**()

{

srand((**unsigned** **int**)time(NULL));

container<**int**> cntr1;

cntr1.add(1);

cntr1.add(2);

cntr1.add(3);

container<**int**> cntr2;

cntr2.add(4);

cntr2.add(5);

cntr2.add(6);

std::thread **t1**(exchange, std::ref(cntr1), std::**ref**(cntr2), 3);

std::thread **t2**(exchange, std::ref(cntr2), std::**ref**(cntr1), 6);

t1.join();

t2.join();

**return** 0;

}

Для решения этой проблемы можно использовать std::lock, который гарантирует блокировку безопасным (с точки зрения взаимоблокировки) способом:

**void** **exchange**(container<**int**> &c1, container<**int**> &c2, **int** value)

{

std::lock(c1.\_lock, c2.\_lock);

c1.remove(value);

c2.add(value);

c1.\_lock.unlock();

c2.\_lock.unlock();

}

Для более полного понимания этой статьи, рекомендуется прочитать ее [первую часть](http://habrahabr.ru/post/182610/), где основное внимание было уделено потокам и блокировкам, в ней объяснено много моментов (терминов, функций и т.д.), которые без пояснения будут использованы здесь.  
В данной статье будут рассмотрены условные переменные… 

Условные переменные

Помимо описанных [ранее](http://habrahabr.ru/post/182610/) способов синхронизации, C++11 предоставляет поддержку условных переменных, которые позволяют блокировать один или более потоков, пока либо не будет получено уведомление от другого потока, либо не произойдет ~~мифическое~~ [spurious wakeup](http://en.wikipedia.org/wiki/Spurious_wakeup) («ложное/случайное пробуждение»).  
Есть две реализации условных переменных, доступных в заголовке <condition\_variable>:

* [condition\_variable](http://en.cppreference.com/w/cpp/thread/condition_variable): требует от любого потока перед ожиданием сначала выполнить std::unique\_lock
* [condition\_variable\_any](http://en.cppreference.com/w/cpp/thread/condition_variable_any): более общая реализация, которая работает с любым типом, который можно заблокировать. Эта реализация может быть более дорогим (с точки зрения ресурсов и производительности) для использования, поэтому ее следует использовать только если необходима те дополнительные возможности, которые она обеспечивает

Опишу, как работают условные переменные:

* Должен быть хотя бы один поток, ожидающий, пока какое-то условие станет истинным. Ожидающий поток должен сначала выполнить unique\_lock. Эта блокировка передается методу wait(), который освобождает мьютекс и приостанавливает поток, пока не будет получен сигнал от условной переменной. Когда это произойдет, поток пробудится и снова выполнится lock.
* Должен быть хотя бы один поток, сигнализирующий о том, что условие стало истинным. Сигнал может быть послан с помощью [notify\_one()](http://en.cppreference.com/w/cpp/thread/condition_variable/notify_one), при этом будет разблокирован один (любой) поток из ожидающих, или [notify\_all()](http://en.cppreference.com/w/cpp/thread/condition_variable/notify_all), что разблокирует все ожидающие потоки.
* В виду некоторых сложностей при создании пробуждающего условия, которое может быть предсказуемых в многопроцессорных системах, могут происходить ложные пробуждения (**spurious wakeup**). Это означает, что поток может быть пробужден, даже если никто не сигнализировал условной переменной. Поэтому необходимо еще проверять, верно ли условие пробуждение уже после то, как поток был пробужден. Т.к. ложные пробуждения могут происходить многократно, такую проверку необходимо организовывать в цикле.

Код ниже демонстрирует пример использования условной переменной, для синхронизации потоков: во время работы некоторых потоков (назовем их «рабочими») могут произойти ошибку, при этом они помещаются в очередь. Поток «регистратора» обрабатывает эти ошибки (получая их из очереди) и печатает их. «Рабочие» сигнализируют «регистратору», когда происходит ошибка. Регистратор ожидает сигнала условной переменной. Чтобы избежать ложных пробуждений, ожидание происходит в цикле, где проверяется булевское условие.

**#include <condition\_variable>**

**#include <iostream>**

**#include <random>**

**#include <thread>**

**#include <mutex>**

**#include <queue>**

std::mutex g\_lockprint;

std::mutex g\_lockqueue;

std::condition\_variable g\_queuecheck;

std::queue<**int**> g\_codes;

**bool** g\_done;

**bool** g\_notified;

**void** **workerFunc**(**int** id, std::mt19937 &generator)

{

*// стартовое сообщение*

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "[worker " << id << "]\trunning..." << std::endl;

}

*// симуляция работы*

std::this\_thread::sleep\_for(std::chrono::seconds(1 + generator() % 5));

*// симуляция ошибки*

**int** errorcode = id\*100+1;

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "[worker " << id << "]\tan error occurred: " << errorcode << std::endl;

}

*// сообщаем об ошибке*

{

std::unique\_lock<std::mutex> locker(g\_lockqueue);

g\_codes.push(errorcode);

g\_notified = true;

g\_queuecheck.notify\_one();

}

}

**void** **loggerFunc**()

{

*// стартовое сообщение*

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "[logger]\trunning..." << std::endl;

}

*// до тех пор, пока не будет получен сигнал*

**while**(!g\_done)

{

std::unique\_lock<std::mutex> locker(g\_lockqueue);

**while**(!g\_notified) *// от ложных пробуждений*

g\_queuecheck.wait(locker);

*// если есть ошибки в очереди, обрабатывать их*

**while**(!g\_codes.empty())

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "[logger]\tprocessing error: " << g\_codes.front() << std::endl;

g\_codes.pop();

}

g\_notified = false;

}

}

**int** **main**()

{

*// инициализация генератора псевдо-случайных чисел*

std::mt19937 **generator**((**unsigned** **int**)std::chrono::system\_clock::**now**().**time\_since\_epoch**().**count**());

*// запуск регистратора*

std::thread **loggerThread**(loggerFunc);

*// запуск рабочих*

std::vector<std::thread> threads;

**for**(**int** i = 0; i < 5; ++i)

threads.push\_back(std::thread(workerFunc, i+1, std::ref(generator)));

**for**(**auto** &t: threads)

t.join();

*// сообщаем регистратору о завершении и ожидаем его*

g\_done = true;

loggerthread.join();

**return** 0;

}

Выполнение этого кода даст примерно следующий результат (результат каждый раз будет разным, т.к. рабочие потоки работают (точнее спят) случайные интервалы времени):

[logger] running...

[worker 1] running...

[worker 2] running...

[worker 3] running...

[worker 4] running...

[worker 5] running...

[worker 1] an error occurred: 101

[worker 2] an error occurred: 201

[logger] processing error: 101

[logger] processing error: 201

[worker 5] an error occurred: 501

[logger] processing error: 501

[worker 3] an error occurred: 301

[worker 4] an error occurred: 401

[logger] processing error: 301

[logger] processing error: 401

У метода wait, обозначенного выше, есть две перегрузки:

* та, что использует только unique\_lock; он (метод) блокирует поток и добавляет его в очередь потоков, ожидающих сигнала от этой условной переменной; поток пробуждается, когда будет получен сигнал от условной переменной или в случае ложного пробуждения.
* та, что в дополнении к unique\_lock, принимает предикат, используемый в цикле до тех пор, пока он не вернет false; эта перегрузка может использоваться, чтобы избежать ложных пробуждений. В общем случае это эквивалентно такому циклу:
* **while**(!predicate())
* wait(lock);

Таким образом, используя вторую перегрузку, можно избежать использования булевского флага g\_notified в примере выше:

**void** **workerFunc**(**int** id, std::mt19937 &generator)

{

*// стартовое сообщение*

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "[worker " << id << "]\trunning..." << std::endl;

}

*// симуляция работы*

std::this\_thread::sleep\_for(std::chrono::seconds(1 + generator() % 5));

*// симуляция ошибки*

**int** errorcode = id\*100+1;

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "[worker " << id << "]\tan error occurred: " << errorcode << std::endl;

}

*// сообщаем об ошибке*

{

std::unique\_lock<std::mutex> locker(g\_lockqueue);

g\_codes.push(errorcode);

g\_queuecheck.notify\_one();

}

}

**void** **loggerFunc**()

{

*// стартовое сообщение*

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "[logger]\trunning..." << std::endl;

}

*// до тех пор, пока не будет получен сигнал*

**while**(!g\_done)

{

std::unique\_lock<std::mutex> locker(g\_lockqueue);

g\_queuecheck.wait(locker, [&](){**return** !g\_codes.empty();});

*// если есть ошибки в очереди, обрабатывать их*

**while**(!g\_codes.empty())

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "[logger]\tprocessing error: " << g\_codes.front() << std::endl;

g\_codes.pop();

}

}

}

В дополнении к перегруженному методу wait(), есть еще два похожих метода с такой же перегрузкой для предиката:

* [wait\_for](http://en.cppreference.com/w/cpp/thread/condition_variable/wait_for): блокирует поток до тех пор, пока не будет получен сигнал условной переменной
* [wait\_until](http://en.cppreference.com/w/cpp/thread/condition_variable/wait_until): блокирует поток до тех пор, пока не будет получен сигнал условной переменной или не будет достигнут определенный момент времени

Перегрузка этих методов без предиката возвращает [cv\_status](http://en.cppreference.com/w/cpp/thread/cv_status), показывающий, произошел ли таймаут, или пробуждение произошло из-за сигнала условной переменной, или это ложное пробуждение.  
  
Std также предоставляет функцию [notify\_all\_at\_thread\_exit](http://en.cppreference.com/w/cpp/thread/notify_all_at_thread_exit), которая реализует механизм уведомления других потоков о том, что данных поток завершил свою работу, включая уничтожение всех объектов thread\_local. Ожидание потоков механизмом, отличным от join, может привести к неправильному поведению, когда thread\_locals уже были использованы, а их деструкторы могли вызываться после того, как поток был пробужден или после того, как уже завершился (см. [N3070](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3070.html) и [N2880](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2880.html). Как правило, вызов этой функции должен произойти непосредственно до того, как поток начнет свое существование. Ниже приведен пример, как notify\_all\_at\_thread\_exit может использоваться с условными переменными для синхронизации двух потоков:

std::mutex g\_lockprint;

std::mutex g\_lock;

std::condition\_variable g\_signal;

**bool** g\_done;

**void** **workerFunc**(std::mt19937 &generator)

{

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "worker running..." << std::endl;

}

std::this\_thread::sleep\_for(std::chrono::seconds(1 + generator() % 5));

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "worker finished..." << std::endl;

}

std::unique\_lock<std::mutex> lock(g\_lock);

g\_done = true;

std::notify\_all\_at\_thread\_exit(g\_signal, std::move(lock));

}

**int** **main**()

{

std::mt19937 **generator**((**unsigned** **int**)std::chrono::system\_clock::**now**().**time\_since\_epoch**().**count**());

std::cout << "main running..." << std::endl;

std::thread **worker**(workerFunc, std::ref(generator));

worker.detach();

std::cout << "main crunching..." << std::endl;

std::this\_thread::sleep\_for(std::chrono::seconds(1 + generator() % 5));

{

std::unique\_lock<std::mutex> locker(g\_lockprint);

std::cout << "main waiting for worker..." << std::endl;

}

std::unique\_lock<std::mutex> lock(g\_lock);

**while**(!g\_done) *// против ложных пробуждений*

g\_signal.wait(lock);

std::cout << "main finished..." << std::endl;

**return** 0;

}

Если worker заканчивает свою работу перед потоком main, то результат будет таким:

main running...

worker running...

main crunching...

worker finished...

main waiting for worker...

main finished...

Если поток main заканчивает свою работу перед потоком worker, то результат будет таким:

main running...

worker running...

main crunching...

main waiting for worker...

worker finished...

main finished...

В качестве заключения

Стандарт C++11 позволяет разработчикам C++ писать многопоточный код стандартным, платформонезависимым способом. Эта статья — всего лишь «пробежка» по потокам и механизмам синхронизации от std. Заголовок <thread> предоставляет класс с тем же именем (и много дополнительных функций), представляющий потоки. Заголовок <mutex> обеспечивает реализацию нескольких мьютексов и «оберток» для синхронизации доступа к потокам. Заголовок <condition\_variable> предоставляет две реализации условных переменных, которые позволяют блокировать один или более потоков, до получение уведомления от другого потока или до ложного пробуждения. Для более подробной информации и понимания сути дела, конечно же, рекомендуется прочитать дополнительную литературу :)