159.341 Programming Languages, Algorithms & Concurrency

C++ std::threads

Daniel Playne
d.p.playne@massey.ac.nz

C++ `std::threads`

Threads were introduced to the C++ language with the release of the C++11 standard. The goal was tough, to provide a system for threads that is as efficient as a natively provided OS library but is portable between different operating systems.

The major advantage of threads being supported by the language is that only one implementation of a program needs to be developed and can simply be compiled for different operating systems.

C++ `std::threads`

Much of the C++ threading functionality is available in the header:

```
#include <thread>
```

The main object to look at is the `std::thread` which is a ***thread*** object or ***worker*** thread. It can be used to create a new thread and start executing a function.

C++ std::threads

Example 1:

```
#include<iostream>
#include<thread>

void thread_function() {
    std::cout << "Hello from thread" << std::endl;
}

int main() {
    std::thread t(thread_function);
    std::cout << "Hello from main" << std::endl;
    t.join();
}
```

Output (maybe):

```
Hello from main
Hello from thread
```

C++ `std::threads`

When an `std::thread` is created, an associated thread of control will be created and begin execution immediately (pending scheduling by the OS).

In general, each `std::thread` object is associated with one thread of execution and a thread of execution may be associated with at most one `std::thread` object.

C++ `std::threads`

There are four cases when an `std::thread` will not be associated with a thread of execution.

- Default construction
- After *move* construction (from)
- `join()` was called
- `detach()` was called

If an `std::thread` object is destructed while it is associated with a thread of execution, `std::terminate` will be called.

C++ `std::threads`

The most common usage is shown in the previous example where a thread is created and given a function to execute. The thread will run this function concurrently with the main thread still running the main function.

When the main thread wishes to end, it will call `join()` on the thread object `t` which will block until the thread of execution associated with `t` has ended.

If `join` is not called then `t` will still be associated with a thread of execution when it is destructed and cause an error.

C++ std::threads

Example 1 (no join):

```
#include<iostream>
#include<thread>

void thread_function() {
   std::cout << "Hello from thread" << std::endl;
}

int main() {
   std::thread t(thread_function);
   std::cout << "Hello from main" << std::endl;
}
```

Output (maybe):

```
Hello from main
libc++abi.dylib: terminating
Abort trap: 6
```

C++ `std::threads`

In most cases you will `join` your threads at the end of the program. However, you may encounter a case where you must destruct the `std::thread` object but allow the thread of execution to continue.

In this case you can `detach` a thread which will allow it to execute independently from the `std::thread` object used to create it.

C++ std::threads

Example 2:

```
#include<iostream>
#include<thread>

void thread_function() {
    std::cout << "Hello from thread" << std::endl;
}

int main() {
    {
        std::thread t(thread_function);
        t.detach();
    }
    std::cout << "Hello from main" << std::endl;
    std::this_thread::sleep_for (std::chrono::seconds(1));
}
```

Output (maybe):

```
Hello from main
Hello from thread
```

C++ std::threads

Once an `std::thread` has been detached it will no longer be associated with a thread of control and `join` cannot be called.

Detaching a thread does not mean it will be able to continue to execute if the main thread ends.

Usually, when the `main` function terminates the enclosing process will be destroyed and any associated threads terminated (without calling `std::terminate` however).

C++ std::threads

The other cases where an `std::thread` object is not attached to a thread of control are default construction and a `move` construction or assignment.

Example 3:

```
int main() {
   std::cout << "Hello from main" << std::endl;

   std::thread t1;                      // t1 has no thread
   std::thread t2(thread_function);     // t2 has thread
   std::thread t3(std::move(t2));       // thread moved to t3
   t1 = std::move(t3);                  // thread moved to t1

   std::this_thread::sleep_for (std::chrono::seconds(4));

   t1.join();
   std::cout << "Goodbye from main" << std::endl;
}
```

C++ `std::threads` - Parameters

C++ syntax makes passing parameters to threads significantly simpler than a C API such as `pthreads`.

Example 4:

```
void thread_function(int start, int end) {
   for(int i = start; i < end; i++) {
      std::cout << i << std::endl;
   }
}

int main() {
   std::thread t1(thread_function,  0,  5);
   std::thread t2(thread_function,  5, 10);
   std::thread t3(thread_function, 10, 15);

   t1.join();  t2.join();  t3.join();
}
```

What will the output of this program be?

Parameters for C++ `std::threads`

Possible output:

```
05

110

116
7
2
3
4
8
9

12
13
14
```

or

```
0510

11
12
13
14

1
2
3
4
6
7
8
9
```

or

```
05

10
11
12
13
14
1
2
6
7
8
9
3
4
```

C++ std::threads - Parameters

Parameters can be passed by reference but must use `std::ref` to create a `std::reference_wrapper`.

Example 5:

```
void thread_function(int start, int end, long long int &total) {
    for(int i = start; i < end; ++i) {
        total += i;
    }
}

int main() {
    long long int total = 0;
    std::thread t0(thread_function,   0,  500, std::ref(total));
    std::thread t1(thread_function, 500, 1000, std::ref(total));
    t0.join();
    t1.join();
    std::cout << total << std::endl;
}
```

What will the output of this program be?

C++ `std::threads` - Parameters

Output (maybe):

```
378845
```

or

```
499500
```

or

```
445626
```

Lambda Functions with C++ `std::threads`

Threads can also be created to execute lambda functions:

Example 6:

```
int main() {
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; i++) {
        threads.push_back(std::thread([i] {
            std::cout << i << std::endl;
        }));
    }
    for(std::thread &t : threads) {
        t.join();
    }
}
```

Output (maybe):

```
01
4

23
```

C++ - `std::mutex`

To provide mutual exclusion, C++ provides mutex objects.

There are four basic types provided:

- `mutex`
- `timed_mutex`
- `recursive_mutex`
- `recursive_timed_mutex`

C++ - `std::mutex`

The `mutex` primitive represents a basic mutex and has three main functions

- `lock()` - locks the mutex and blocks the thread if mutex is not available.
- `try_lock()` - tries to lock the mutex and returns `false` if the mutex was not obtained.
- `unlock()` - releases the lock on the mutex.

C++ - `std::mutex`

This mutex provides exclusive, non-recursive semantics:

- A thread *holds* the mutex from the time it successfully calls `lock` or `try_lock` and until it calls `unlock`.
- When a thread *holds* a mutex, any other thread will block (`lock`) or return false (`try_lock`) if they attempt to claim it.
- A thread must not already *hold* a mutex before calling `lock` or `try_lock`.

C++ - std::mutex

Example 7:

```
std::mutex mtx;

void thread_function(int id) {
    mtx.lock();
    std::cout << id << " counting to 3" << std::endl;
    for(int i = 0; i < 3; ++i) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
    mtx.unlock();
}
int main() {
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; i++) {
        threads.push_back(std::thread(thread_function, i));
    }
    for(std::thread &t : threads) {
        t.join();
    }
}
```

Output (maybe):

```
Thread: 1 counting to 3
0 1 2
Thread: 2 counting to 3
0 1 2
Thread: 0 counting to 3
0 1 2
Thread: 3 counting to 3
0 1 2
Thread: 4 counting to 3
0 1 2
```

C++ - `std::timed_mutex`

A thread may wish to obtain a lock on a mutex, however there may be cases when that mutex may be held by another thread for a significant period of time.

In such situations, being able to specify a timeout period can be useful to stop threads waiting too long to obtain a lock on a mutex.

The `timed_mutex` primitive extends `mutex` by including functions that allow a thread to include a maximum amount of time to wait when trying to claim a mutex.

C++ - std::timed_mutex

In addition to the functions provided by `mutex` it also provides:

- `try_lock_for()` - try to lock the mutex and return `false` if the mutex has been unavailable for a specified duration.
- `try_lock_until()` - try to lock the mutex and return `false` if the mutex has been unavailable until a specified timepoint has been reached.

C++ - std::timed_mutex

Example 8:

```
std::timed_mutex tmtx;

void thread_function(int id) {
    if(tmtx.try_lock_for(std::chrono::milliseconds(1000))) {
        std::this_thread::sleep_for(std::chrono::milliseconds(330));
        std::cout << "Thread " << id << " done." << std::endl;
        tmtx.unlock();
    } else {
        std::cout << "Thread " << id << " gave up." << std::endl;
    }
}
int main() {
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; i++) {
        threads.push_back(std::thread(thread_function, i));
    }
    for(std::thread &t : threads) {
        t.join();
    }
}
```

C++ - `std::timed_mutex`

Output (maybe):

```
Thread 1 done.
Thread 3 done.
Thread 2 done.
Thread 4 gave up.
Thread 0 gave up.
```

or

```
Thread 1 done.
Thread 0 done.
Thread 3 done.
Thread 4 gave up.
Thread 2 done.
```

C++ - `std::recursive_mutex`

Sometimes (often in a recursive function) it is not always straightforward to determine whether a thread has already obtained a lock on a mutex or not.

For this case, `C++` provides another type of mutex called `std::recursive_mutex`.

This mutex may be locked multiple times (by the same thread) and is considered to be unlocked when the matching number of unlock calls have been made.

C++ - `std::recursive_mutex`

This mutex provides exclusive, recursive semantics:

- A thread *holds* the recursive mutex from the time it successfully calls `lock` or `try_lock`. Once held, the thread may make additional calls to `lock` and `try_lock` and will continue to hold the mutex until the matching number of calls to `unlock` have been made.
- When a recursive mutex is held by a thread, calls to `lock` or `try_lock` made by any other thread will block or return false.
- The maximum number of times a `recursive_mutex` may be locked by a thread is unspecified but will throw `std::system_error` (`lock`) or return false `try_lock` when reached.

Example 9:

```
std::recursive_mutex rmtx;

void thread_function(int id) {
    rmtx.lock();
    std::cout << id << " ";
    if(id > 0) {
        thread_function(id - 1);
    } else {
        std::cout << std::endl;
    }
    rmtx.unlock();
}
int main() {
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; i++) {
        threads.push_back(std::thread(thread_function, i));
    }
    for(std::thread &t : threads) {
        t.join();
    }
}
```

C++ - `std::recursive_mutex`

Output (maybe):

```
1 0
0
3 2 1 0
4 3 2 1 0
2 1 0
```

or

```
1 0
2 1 0
3 2 1 0
4 3 2 1 0
0
```

C++ - `std::lock`

Another mechanism for locking a mutex is to use the function `std::lock` which allows multiple mutexes to be locked at the same time while avoiding deadlock (order does not matter).

```
std::mutex mtx1, mtx2;

void thread_function_a() {
   std::lock(mtx1, mtx2);
   std::cout << "Thread a" << std::endl;
   mtx1.unlock(); mtx2.unlock();
}

void thread_function_b() {
   std::lock(mtx2, mtx1);
   std::cout << "Thread b" << std::endl;
   mtx1.unlock(); mtx2.unlock();
}

int main() {
   std::thread t_a(thread_function_a);
   std::thread t_b(thread_function_b);
   t_a.join();
   t_b.join();
}
```

C++ - `std::lock`

The definition of `std::lock` states that all mutexes will be locked via a sequence of calls to `lock`, `try_lock` and `unlock` and will not result in deadlock. The exact method for avoiding deadlock is not specified (to avoid constraining implementations).

The definition must be interpreted with care, using `std::lock` to lock mutexes does not mean that deadlock cannot occur globally throughout the program.

It means that within a single call to `std::lock` the thread will not hold the lock on any mutex while waiting for another to become available.

C++ - `std::lock`

Deadlock can still occur when multiple calls to `std::lock` are made.

```
std::mutex mtx1, mtx2, mtx3, mtx4;

void thread_function_a() {
   cout << "Wait - 1,2 (a)" << endl;
   std::lock(mtx1, mtx2);
   cout << "Wait - 3,4 (a)" << endl;
   std::lock(mtx3, mtx4);
   mtx1.unlock(); mtx2.unlock();
   mtx3.unlock(); mtx4.unlock();
}
```

```
void thread_function_b() {
   cout << "Wait - 3,4 (b)" << endl;
   std::lock(mtx3, mtx4);
   cout << "Wait - 1,2 (b)" << endl;
   std::lock(mtx1, mtx2);
   mtx1.unlock(); mtx2.unlock();
   mtx3.unlock(); mtx4.unlock();
}
```

```
int main() {
   std::thread t_a(thread_function_a);
   std::thread t_b(thread_function_b);
   t_a.join();
   t_b.join();
}
```

C++ std::threads - Mutexes

Mutexes are generally not intended to be used directly and more commonly utlised through `lock_guard` or `std::unique_lock`.

A `lock_guard` is an RAII (Resource Allocation Is Initialisation) mechanism for taking ownership of a mutex for the duration of a scoped block.

A `unique_lock` is a general-purpose mutex ownership wrapper that supports deferred locking, time-constrained attempts at locking, recursive locking, transfer of ownership and use with condition variables.

Summary

C++ threads:

- `thread`
- `mutex`
- `lock`