Learn C++ Multi-Threading in 5 Minutes

Summary: A crash course on the C++14 multi-threading constructs in a very non-verbose manner

Ari Saif Follow

Oct 28, 2018 · 5 min read

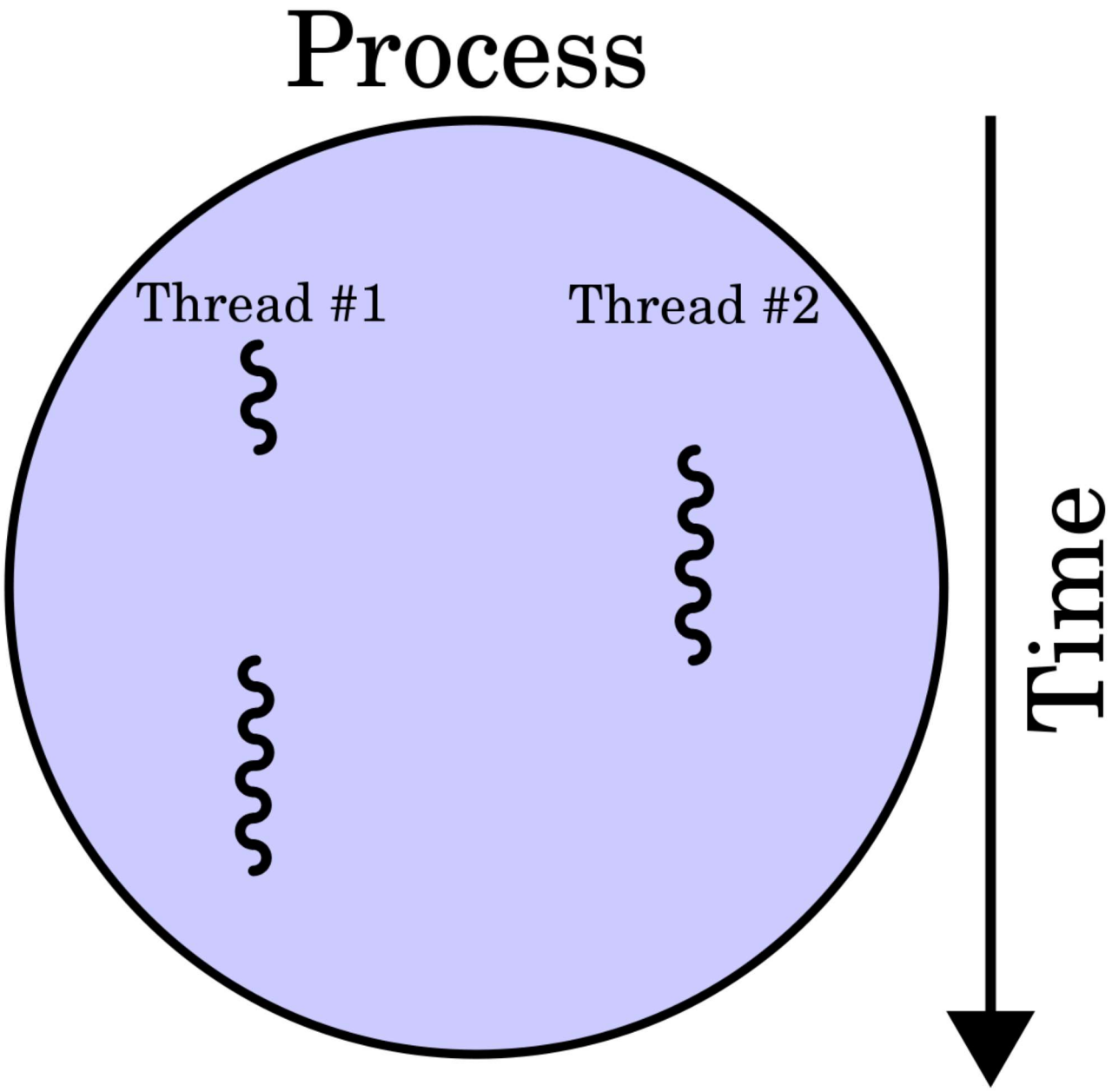

Photo credit: Wikipedia

The new C++ multi-threading constructs are very easy to learn. If you are familiar with C or C++ and want to start writing multithreaded programs, this article is for you!

I use C++14 as a reference, but what I describe is also supported in C++17 . I only cover common constructs. You should be able to write your own multithreaded programs after reading this.

Creating Threads

A thread can be created in several ways:

1. Using a function pointer
2. Using a functor
3. Using a lambda function

These methods are very similar with minor differences. I explain each method and their differences next.

Using a function pointer

Consider the following function which takes a vector reference `v` , a reference to the result `acm` , and two indices in the vector `v` . The function adds all elements between `beginIndex` and `endIndex` .

```
void accumulator_function2(const std::vector<int> &v, unsigned long long &acm,
                           unsigned int beginIndex, unsigned int endIndex)
{
    acm = 0;
    for (unsigned int i = beginIndex; i < endIndex; ++i)
    {
        acm += v[i];
    }
}
```

functionPointerDef.cpp hosted with ♡ by GitHub view raw

A function calculating the sum of all elements between `beginIndex and endIndex in a vector v`

Now lets say you want to partition the vector in two sections and calculate the total sum of each section in a separate thread `t1` and `t2` :

```
//Pointer to function
    {
        unsigned long long acm1 = 0;
        unsigned long long acm2 = 0;
        std::thread t1(accumulator_function2, std::ref(v),
                       std::ref(acm1), 0, v.size() / 2);
        std::thread t2(accumulator_function2, std::ref(v),
                       std::ref(acm2), v.size() / 2, v.size());
        t1.join();
        t2.join();

        std::cout << "acm1: " << acm1 << endl;
        std::cout << "acm2: " << acm2 << endl;
        std::cout << "acm1 + acm2: " << acm1 + acm2 << endl;
    }
```

mtPointerToFunction.cpp hosted with ♡ by GitHub view raw

Creating threads using function pointers

What do you need to take away?

1. `std::thread` creates a new thread. The first parameter is the name of the function pointer `accumulator_function2` . Therefore, each thread will execute this function.
2. The rest of the parameters passed to `std::thread` constructor are the parameters that we need to pass to `accumulator_function2` .
3. **Important:** All parameters passed to `accumulator_function2` are passed by value unless you wrap them in `std::ref` . That's why we wrapped `v` , `acm1` , and `acm2` in `std::ref` .
4. Threads created by `std::thread` do not have return values. If you want to return something, you should store it in one of the parameters passed by reference, i.e. `acm` .
5. Each thread starts as soon as it gets created.
6. We use `join()` function to wait for a thread to finish

Using Functors

You can do exactly the same thing using functors. The following is the code that uses a functor:

```
class CAccumulatorFunctor3
{
  public:
    void operator()(const std::vector<int> &v,
                    unsigned int beginIndex, unsigned int endIndex)
    {
        _acm = 0;
        for (unsigned int i = beginIndex; i < endIndex; ++i)
        {
            _acm += v[i];
        }
    }
    unsigned long long _acm;
};
```

MTFunctorDef.cpp hosted with ♡ by GitHub view raw

Functor Definition

And the code that creates the threads is:

```
//Creating Thread using Functor
    {

        CAccumulatorFunctor3 accumulator1 = CAccumulatorFunctor3();
        CAccumulatorFunctor3 accumulator2 = CAccumulatorFunctor3();
        std::thread t1(std::ref(accumulator1),
            std::ref(v), 0, v.size() / 2);
        std::thread t2(std::ref(accumulator2),
            std::ref(v), v.size() / 2, v.size());
        t1.join();
        t2.join();

        std::cout << "acm1: " << accumulator1._acm << endl;
        std::cout << "acm2: " << accumulator2._acm << endl;
        std::cout << "accumulator1._acm + accumulator2._acm : " <<
            accumulator1._acm + accumulator2._acm << endl;
    }
```

MTFunctor.cpp hosted with ♡ by GitHub view raw

Creating threads using functors

What do you need to take away?

Everything is very similar to function pointer, except that:

1. The first parameter is the functor object.
2. Instead of passing a reference to the functor to store the result, we can store its return value in a member variable inside the functor, i.e. in `_acm`.

Using Lambda Functions

As the third alternative we can define each thread in a lambda function as shown below:

```
{
        unsigned long long acm1 = 0;
        unsigned long long acm2 = 0;
        std::thread t1([&acm1, &v] {
            for (unsigned int i = 0; i < v.size() / 2; ++i)
            {
                acm1 += v[i];
            }
        });
        std::thread t2([&acm2, &v] {
            for (unsigned int i = v.size() / 2; i < v.size(); ++i)
            {
                acm2 += v[i];
            }
        });
        t1.join();
        t2.join();

        std::cout << "acm1: " << acm1 << endl;
        std::cout << "acm2: " << acm2 << endl;
        std::cout << "acm1 + acm2: " << acm1 + acm2 << endl;
    }
```

MTLambdaDef.cpp hosted with ♡ by GitHub view raw

Creating threads using lambda functions

Again, everything is very similar to function pointer, except that:

1. As an alternative to pass a parameter, we can pass references to lambda functions using lambda capture.

Tasks, Futures, and Promises

As an alternative to `std::thread`, you can use `tasks`.

Tasks work very similar to threads, but the main difference is that they can return a value. So, you can remember them as a more abstract way of defining your threads and use them when the threads return a value.

Below is the same example written using tasks:

```
#include <future>
//Tasks, Future, and Promises
    {
        auto f1 = [](std::vector<int> &v,
            unsigned int left, unsigned int right) {
            unsigned long long acm = 0;
            for (unsigned int i = left; i < right; ++i)
            {
                acm += v[i];
            }

            return acm;
        };

        auto t1 = std::async(f1, std::ref(v),
            0, v.size() / 2);
        auto t2 = std::async(f1, std::ref(v),
            v.size() / 2, v.size());

        //You can do other things here!
        unsigned long long acm1 = t1.get();
        unsigned long long acm2 = t2.get();

        std::cout << "acm1: " << acm1 << endl;
        std::cout << "acm2: " << acm2 << endl;
        std::cout << "acm1 + acm2: " << acm1 + acm2 << endl;
    }
```

MTTaskDef.cpp hosted with ♡ by GitHub view raw

What do you need to take away?

1. Tasks are defined and created using `std::async` , (instead of threads that are created using `std::thread`)
2. The returned value from `std::async` is called a `std::future` . Don't get scared by its name. It just means `t1` and `t2` are variables whose value will be assigned to in the future. We get their values by calling `t1.get()` and `t2.get()`
3. If the future values are not ready, upon calling `get()` the main thread blocks until the future value becomes ready (similar to `join()`).
4. Notice that the function that we passed to `std::async` returns a value. This value is passed through a type called `std::promise` . Again, don't get scared by its name. For the most part, you don't need to know details of `std::promise` or define any variable of type `std::promise`. The C++ library does that behind the scenes.
5. Each task by default starts as soon as it is created (there is a way to change this which I don't cover).

Summary of Creating Threads

There you have it. Creating threads is as simple as what I explained above. You can either use `std::thread` :

1. Use function pointers
2. Use functors
3. Use lambda functions

Or you can use `std::async` to create a task and get the return values in a `std::future` . Tasks can get also use a function pointer, a functor, or a lambda function.

Shared Memory and Shared Resources

In short, threads should be careful when they read/write into shared memory and resources (such as files) to avoid race conditions.

C++14 provides several constructs to synchronize threads to avoid such race conditions.

Using Mutex, lock,() and unlock() (Not recommended)

The following code shows how we create a critical section such that each thread accesses `std::cout` exclusively:

```
1  std::mutex g_display_mutex;
2  thread_function()
3  {
4
5      g_display_mutex.lock();
6      std::thread::id this_id = std::this_thread::get_id();
7      std::cout << "My thread id is: " << this_id  << endl;
8      g_display_mutex.unlock();
9  }
```

mutex.cpp hosted with ♡ by GitHub view raw

What do you need to take away?

1. A mutex is created `std::mutex`
2. A critical section (i.e. guaranteed to be run only by a single thread at each time) is created using `lock()`
3. The critical section ends upon calling `unlock()`
4. Each thread waits at `lock()` and only enters the critical section if no other thread is inside that section.

While the above method works, it is not recommended because:

1. It is not exception safe: if the code before lock generates an exception, `unlock()` will not be executed, and we never release the mutex which might cause deadlock
2. We always have to be careful not to forget to call `unlock()`

Using std::lock_guard (recommended)

Don't get scared by its name `lock_guard`. It's just a more abstract way of creating critical sections.

Below is the same critical section using `lock_guard`.

```
std::mutex g_display_mutex;
thread_function()
{
    std::lock_guard<std::mutex> guard(g_display_mutex);
    std::thread::id this_id = std::this_thread::get_id();
    std::cout << "From thread " << this_id  << endl;
}
```

lock_guard.cpp hosted with ♡ by GitHub view raw

critical section using `lock_guard`

What do you need to take away?

1. The code coming after `std::lock_guard` creation is automatically locked. No need for explicit `lock()` and `unlock()` function calls.
2. The critical section automatically ends when `std::lock_guard` goes out of scope. This makes it exception safe, and also we don't need to remember to call `unlock()`
3. `lock_guard` still requires using a variable of type `std::mutex` in its constructor.

How Many Threads Should We Create?

You can create as many threads as you want, but it would probably be pointless if the number of active threads is more than the number of available CPU cores.

In order to get the maximum number of cores you can call: `std::thread::hardware_concurrency()` as shown below:

```
{
    unsigned int c = std::thread::hardware_concurrency();
    std::cout << " number of cores: " << c << endl;;
}
```

hardware_concurrency.cpp hosted with ♡ by GitHub view raw

What I Didn't Cover

I covered most of what you need to create threads. There are several other details that are less common which I don't include here, but you can study them on your own:

1. std::move
2. details of std::promise
3. std::packaged_task
4. Conditional variables

. . .

Hope this helps you learning C++ multi threading quickly.

If you liked this article please click on the clap and give me feedback.

Programming Multithreading Cplusplus Learn C Plus Cpp