4 Synchronizing concurrent operations

This chapter covers

- Waiting for an event
- Waiting for one-off events with futures
- Waiting with a time limit
- Using synchronization of operations to simplify code

In the last chapter, we looked at various ways of protecting data that's shared between threads. But sometimes you don't just need to protect the data but also to synchronize actions on separate threads. One thread might need to wait for another thread to complete a task before the first thread can complete its own, for example. In general, it's common to want a thread to wait for a specific event to happen or a condition to be `true`. Although it would be possible to do this by periodically checking a "task complete" flag or something similar stored in shared data, this is far from ideal. The need to synchronize operations between threads like this is such a common scenario that the C++ Standard Library provides facilities to handle it, in the form of *condition variables* and *futures*.

In this chapter I'll discuss how to wait for events with condition variables and futures and how to use them to simplify the synchronization of operations.

4.1 Waiting for an event or other condition

Suppose you're traveling on an overnight train. One way to ensure you get off at the right station would be to stay awake all night and pay attention to where the train stops. You wouldn't miss your station, but you'd be tired when you got there. Alternatively, you could look at the timetable to see when the train is supposed to arrive, set your alarm a bit before, and go to sleep. That would be OK; you wouldn't miss your stop, but if the train got delayed, you'd wake up too early. There's also the possibility that your alarm clock's batteries would die, and you'd sleep too long and miss your station. What would be ideal is if you could just go to sleep and have somebody or something wake you up when the train gets to your station, whenever that is.

How does that relate to threads? Well, if one thread is waiting for a second thread to complete a task, it has several options. First, it could just keep checking a flag in shared data (protected by a mutex) and have the second thread set the flag when it completes the task. This is wasteful on two counts: the thread consumes valuable processing time repeatedly checking the flag, and when the mutex is locked by the waiting thread, it can't be locked by any other thread. Both of these work against the thread doing the waiting, because they limit the resources available to the thread being waited for and even prevent it from setting the flag when it's done. This is akin to staying awake all night talking to the train driver: he has to drive the train more slowly because you keep distracting him, so it takes longer to get there. Similarly, the waiting thread is consuming resources that could be used by other threads in the system and may end up waiting longer than necessary.

A second option is to have the waiting thread sleep for small periods between the checks using the `std::this_thread::sleep_for()` function (see section 4.3):

```
bool flag;
std::mutex m;

void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lk.lock();
    }
}
```

❶ Unlock the mutex

❷ Sleep for 100 ms

❸ Relock the mutex

In the loop, the function unlocks the mutex ❶ before the sleep ❷ and locks it again afterward ❸, so another thread gets a chance to acquire it and set the flag.

This is an improvement, because the thread doesn't waste processing time while it's sleeping, but it's hard to get the sleep period right. Too short a sleep in between checks and the thread still wastes processing time checking; too long a sleep and the thread will keep on sleeping even when the task it's waiting for is complete, introducing a delay. It's rare that this oversleeping will have a direct impact on the operation of

the program, but it could mean dropped frames in a fast-paced game or overrunning a time slice in a real-time application.

The third, and preferred, option is to use the facilities from the C++ Standard Library to wait for the event itself. The most basic mechanism for waiting for an event to be triggered by another thread (such as the presence of additional work in the pipeline mentioned previously) is the *condition variable*. Conceptually, a condition variable is associated with some event or other *condition*, and one or more threads can *wait* for that condition to be satisfied. When some thread has determined that the condition is satisfied, it can then *notify* one or more of the threads waiting on the condition variable, in order to wake them up and allow them to continue processing.

4.1.1 Waiting for a condition with condition variables

The Standard C++ Library provides not one but *two* implementations of a condition variable: `std::condition_variable` and `std::condition_variable_any`. Both of these are declared in the `<condition_variable>` library header. In both cases, they need to work with a mutex in order to provide appropriate synchronization; the former is limited to working with `std::mutex`, whereas the latter can work with anything that meets some minimal criteria for being mutex-like, hence the `_any` suffix. Because `std::condition_variable_any` is more general, there's the potential for additional costs in terms of size, performance, or operating system resources, so `std::condition_variable` should be preferred unless the additional flexibility is required.

So, how do you use a `std::condition_variable` to handle the example in the introduction—how do you let the thread that's waiting for work sleep until there's data to process? The following listing shows one way you could do this with a condition variable.

Listing 4.1 Waiting for data to process with a `std::condition_variable`

```
std::mutex mut;
std::queue<data_chunk> data_queue;     <-- 1
std::condition_variable data_cond;

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);                <-- 2
        data_cond.notify_one();    <-- 3
    }
}

void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut);    <-- 4
```

```
        data_cond.wait(
            lk,[]{return !data_queue.empty();});    <-5
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();          <-6
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

First off, you have a queue ❶ that's used to pass the data between the two threads. When the data is ready, the thread preparing the data locks the mutex protecting the queue using a `std::lock_guard` and pushes the data onto the queue ❷. It then calls the `notify_one()` member function on the `std::condition_variable` instance to notify the waiting thread (if there is one) ❸.

On the other side of the fence, you have the processing thread. This thread first locks the mutex, but this time with a `std::unique_lock` rather than a `std::lock_guard` ❹—you'll see why in a minute. The thread then calls `wait()` on the `std::condition_variable`, passing in the lock object and a lambda function that expresses the condition being waited for ❺. Lambda functions are a new feature in C++11 that allows you to write an anonymous function as part of another expression, and they're ideally suited for specifying predicates for standard library functions such as `wait()`. In this case, the simple lambda function `[]{return !data_queue.empty();}` checks to see if the `data_queue` is not `empty()`—that is, there's some data in the queue ready for processing. Lambda functions are described in more detail in appendix A, section A.5.

The implementation of `wait()` then checks the condition (by calling the supplied lambda function) and returns if it's satisfied (the lambda function returned `true`). If the condition isn't satisfied (the lambda function returned `false`), `wait()` unlocks the mutex and puts the thread in a blocked or waiting state. When the condition variable is notified by a call to `notify_one()` from the data-preparation thread, the thread wakes from its slumber (unblocks it), reacquires the lock on the mutex, and checks the condition again, returning from `wait()` with the mutex still locked if the condition has been satisfied. If the condition hasn't been satisfied, the thread unlocks the mutex and resumes waiting. This is why you need the `std::unique_lock` rather than the `std::lock_guard`—the waiting thread must unlock the mutex while it's waiting and lock it again afterward, and `std::lock_guard` doesn't provide that flexibility. If the mutex remained locked while the thread was sleeping, the data-preparation thread wouldn't be able to lock the mutex to add an item to the queue, and the waiting thread would never be able to see its condition satisfied.

Listing 4.1 uses a simple lambda function for the wait ❺, which checks to see if the queue is not empty, but any function or callable object could be passed. If you already have a function to check the condition (perhaps because it's more complicated than a simple test like this), then this function can be passed in directly; there's no need

to wrap it in a lambda. During a call to `wait()`, a condition variable may check the supplied condition any number of times; however, it always does so with the mutex locked and will return immediately if (and only if) the function provided to test the condition returns `true`. When the waiting thread reacquires the mutex and checks the condition, if it isn't in direct response to a notification from another thread, it's called a *spurious wake*. Because the number and frequency of any such spurious wakes are by definition indeterminate, it isn't advisable to use a function with side effects for the condition check. If you do so, you must be prepared for the side effects to occur multiple times.

The flexibility to unlock a `std::unique_lock` isn't just used for the call to `wait()`; it's also used once you have the data to process but before processing it ❻. Processing data can potentially be a time-consuming operation, and as you saw in chapter 3, it's a bad idea to hold a lock on a mutex for longer than necessary.

Using a queue to transfer data between threads as in listing 4.1 is a common scenario. Done well, the synchronization can be limited to the queue itself, which greatly reduces the possible number of synchronization problems and race conditions. In view of this, let's now work on extracting a generic thread-safe queue from listing 4.1.

4.1.2 Building a thread-safe queue with condition variables

If you're going to be designing a generic queue, it's worth spending a few minutes thinking about the operations that are likely to be required, as you did with the thread-safe stack back in section 3.2.3. Let's look at the C++ Standard Library for inspiration, in the form of the `std::queue<>` container adaptor shown in the following listing.

Listing 4.2 `std::queue` interface

```
template <class T, class Container = std::deque<T> >
class queue {
public:
    explicit queue(const Container&);
    explicit queue(Container&& = Container());

    template <class Alloc> explicit queue(const Alloc&);
    template <class Alloc> queue(const Container&, const Alloc&);
    template <class Alloc> queue(Container&&, const Alloc&);
    template <class Alloc> queue(queue&&, const Alloc&);

    void swap(queue& q);

    bool empty() const;
    size_type size() const;

    T& front();
    const T& front() const;
    T& back();
    const T& back() const;

    void push(const T& x);
    void push(T&& x);
```

```
    void pop();
    template <class... Args> void emplace(Args&&... args);
};
```

If you ignore the construction, assignment and swap operations, you're left with three groups of operations: those that query the state of the whole queue (`empty()` and `size()`), those that query the elements of the queue (`front()` and `back()`), and those that modify the queue (`push()`, `pop()` and `emplace()`). This is the same as you had back in section 3.2.3 for the stack, and therefore you have the same issues regarding race conditions inherent in the interface. Consequently, you need to combine `front()` and `pop()` into a single function call, much as you combined `top()` and `pop()` for the stack. The code from listing 4.1 adds a new nuance, though: when using a queue to pass data between threads, the receiving thread often needs to wait for the data. Let's provide two variants on `pop()`: `try_pop()`, which tries to pop the value from the queue but always returns immediately (with an indication of failure) even if there wasn't a value to retrieve, and `wait_and_pop()`, which will wait until there's a value to retrieve. If you take your lead for the signatures from the stack example, your interface looks like the following.

Listing 4.3 The interface of your `threadsafe_queue`

```
#include <memory>                          <- For std::shared_ptr

template<typename T>
class threadsafe_queue
{
public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue&);
    threadsafe_queue& operator=(
        const threadsafe_queue&) = delete;      <- Disallow assignment for simplicity

    void push(T new_value);

    bool try_pop(T& value);                     <- 1
    std::shared_ptr<T> try_pop();               <- 2

    void wait_and_pop(T& value);
    std::shared_ptr<T> wait_and_pop();

    bool empty() const;
};
```

As you did for the stack, you've cut down on the constructors and eliminated assignment in order to simplify the code. You've also provided two versions of both `try_pop()` and `wait_for_pop()`, as before. The first overload of `try_pop()` (1) stores the retrieved value in the referenced variable, so it can use the return value for status; it returns `true` if it retrieved a value and `false` otherwise (see section A.2). The second overload (2) can't do this, because it returns the retrieved value directly. But the returned pointer can be set to `NULL` if there's no value to retrieve.

So, how does all this relate to listing 4.1? Well, you can extract the code for `push()` and `wait_and_pop()` from there, as shown in the next listing.

Listing 4.4 Extracting `push()` and `wait_and_pop()` from listing 4.1

```
#include <queue>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
    std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }
};

threadsafe_queue<data_chunk> data_queue;    <-- 1

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        data_queue.push(data);                <-- 2
    }
}

void data_processing_thread()
{
    while(true)
    {
        data_chunk data;
        data_queue.wait_and_pop(data);        <-- 3
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

The mutex and condition variable are now contained within the `threadsafe_queue` instance, so separate variables are no longer required ❶, and no external synchronization is required for the call to `push()` ❷. Also, `wait_and_pop()` takes care of the condition variable wait ❸.

The other overload of `wait_and_pop()` is now trivial to write, and the remaining functions can be copied almost verbatim from the stack example in listing 3.5. The final queue implementation is shown here.

Listing 4.5 Full class definition for a thread-safe queue using condition variables

```
#include <queue>
#include <memory>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    threadsafe_queue(threadsafe_queue const& other)
    {
        std::lock_guard<std::mutex> lk(other.mut);
        data_queue=other.data_queue;
    }

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }

    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
        data_queue.pop();
        return res;
    }
```

❶ **The mutex must be mutable**

```
    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=data_queue.front();
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
        data_queue.pop();
        return res;
    }

    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

Even though `empty()` is a `const` member function, and the `other` parameter to the copy constructor is a `const` reference, other threads may have non-`const` references to the object, and be calling mutating member functions, so we still need to lock the mutex. Since locking a mutex is a mutating operation, the mutex object must be marked `mutable` ❶ so it can be locked in `empty()` and in the copy constructor.

Condition variables are also useful where there's more than one thread waiting for the same event. If the threads are being used to divide the workload, and thus only one thread should respond to a notification, exactly the same structure as shown in listing 4.1 can be used; just run multiple instances of the data—processing thread. When new data is ready, the call to `notify_one()` will trigger one of the threads currently executing `wait()` to check its condition and thus return from `wait()` (because you've just added an item to the `data_queue`). There's no guarantee which thread will be notified or even if there's a thread waiting to be notified; all the processing threads might be still processing data.

Another possibility is that several threads are waiting for the same event, and all of them need to respond. This can happen where shared data is being initialized, and the processing threads can all use the same data but need to wait for it to be initialized (although there are better mechanisms for this; see section 3.3.1 in chapter 3), or where the threads need to wait for an update to shared data, such as a periodic reinitialization. In these cases, the thread preparing the data can call the `notify_all()` member function on the condition variable rather than `notify_one()`. As the name suggests, this causes *all* the threads currently executing `wait()` to check the condition they're waiting for.

If the waiting thread is going to wait only once, so when the condition is `true` it will never wait on this condition variable again, a condition variable might not be the best