C++ Concurrency IN ACTION

Practical Multithreading

Anthony Williams

MANNING

Table of Contents

9 Advanced thread management

This chapter covers

- Thread pools
- Handling dependencies between pool tasks
- Work stealing for pool threads
- Interrupting threads

In earlier chapters, we've been explicitly managing threads by creating `std::thread` objects for every thread. In a couple of places you've seen how this can be undesirable, because you then have to manage the lifetime of the thread objects, determine the number of threads appropriate to the problem and to the current hardware, and so forth. The ideal scenario would be that you could just divide the code into the smallest pieces that can be executed concurrently, pass them over to the compiler and library, and say, "Parallelize this for optimal performance."

Another recurring theme in several of the examples is that you might use several threads to solve a problem but require that they finish early if some condition is met. This might be because the result has already been determined, or because an error has occurred, or even because the user has explicitly requested that the operation be aborted. Whatever the reason, the threads need to be sent a "Please stop" request so that they can give up on the task they were given, tidy up, and finish as soon as possible.

In this chapter, we'll look at mechanisms for managing threads and tasks, starting with the automatic management of the number of threads and the division of tasks between them.

9.1 Thread pools

In many companies, employees who would normally spend their time in the office are occasionally required to visit clients or suppliers or attend a trade show or conference. Although these trips might be necessary, and on any given day there might be several people making such a trip, it may well be months or even years between such trips for any particular employee. Since it would therefore be rather expensive and impractical for each employee to have a company car, companies often offer a *car pool* instead; they have a limited number of cars that are available to all employees. When an employee needs to make an off-site trip, they book one of the pool cars for the appropriate time and return it for others to use when they return to the office. If there are no pool cars free on a given day, the employee will have to reschedule their trip for a subsequent date.

A *thread pool* is a similar idea, except that *threads* are being shared rather than cars. On most systems, it's impractical to have a separate thread for every task that can potentially be done in parallel with other tasks, but you'd still like to take advantage of the available concurrency where possible. A thread pool allows you to accomplish this; tasks that can be executed concurrently are submitted to the pool, which puts them on a queue of pending work. Each task is then taken from the queue by one of the *worker threads*, which executes the task before looping back to take another from the queue.

There are several key design issues when building a thread pool, such as how many threads to use, the most efficient way to allocate tasks to threads, and whether or not you can wait for a task to complete. In this section we'll look at some thread pool implementations that address these design issues, starting with the simplest possible thread pool.

9.1.1 The simplest possible thread pool

At its simplest, a thread pool is a fixed number of *worker threads* (typically the same number as the value returned by `std::thread::hardware_concurrency()`) that process work. When you have work to do, you call a function to put it on the queue of pending work. Each worker thread takes work off the queue, runs the specified task, and then goes back to the queue for more work. In the simplest case there's no way to wait for the task to complete. If you need to do this, you have to manage the synchronization yourself.

The following listing shows a sample implementation of such a thread pool.

Listing 9.1 Simple thread pool

```
class thread_pool
{
    std::atomic_bool done;
    thread_safe_queue<std::function<void()> > work_queue;        ❶
    std::vector<std::thread> threads;                           ❷
    join_threads joiner;                ❸

    void worker_thread()
    {
        while(!done)        ❹
        {
            std::function<void()> task;
            if(work_queue.try_pop(task))        ❺
            {
                task();        ❻
            }
            else
            {
                std::this_thread::yield();        ❼
            }
        }
    }
public:
    thread_pool():
        done(false),joiner(threads)
    {
        unsigned const thread_count=std::thread::hardware_concurrency();   ❽

        try
        {
            for(unsigned i=0;i<thread_count;++i)
            {
                threads.push_back(
                    std::thread(&thread_pool::worker_thread,this));    ❾
            }
        }
        catch(...)
        {
            done=true;        ❿
            throw;
        }
    }

    ~thread_pool()
    {
        done=true;          ⓫
    }

    template<typename FunctionType>
    void submit(FunctionType f)
    {
        work_queue.push(std::function<void()>(f));      ⓬
    }
};
```

This implementation has a vector of worker threads ❷ and uses one of the thread-safe queues from chapter 6 ❶ to manage the queue of work. In this case, users can't wait for the tasks, and they can't return any values, so you can use `std::function<void()>` to encapsulate your tasks. The `submit()` function then wraps whatever function or callable object is supplied inside a `std::function<void()>` instance and pushes it on the queue ⓬.

The threads are started in the constructor: you use `std::thread::hardware_concurrency()` to tell you how many concurrent threads the hardware can support ❽, and you create that many threads running your `worker_thread()` member function ❾.

Starting a thread can fail by throwing an exception, so you need to ensure that any threads you've already started are stopped and cleaned up nicely in this case. This is achieved with a `try-catch` block that sets the `done` flag when an exception is thrown ❿, alongside an instance of the `join_threads` class from chapter 8 ❸ to join all the threads. This also works with the destructor: you can just set the `done` flag ⓫, and the `join_threads` instance will ensure that all the threads have completed before the pool is destroyed. Note that the order of declaration of the members is important: both the `done` flag and the `worker_queue` must be declared before the `threads` vector, which must in turn be declared before the `joiner`. This ensures that the members are destroyed in the right order; you can't destroy the queue safely until all the threads have stopped, for example.

The `worker_thread` function itself is quite simple: it sits in a loop waiting until the `done` flag is set ❹, pulling tasks off the queue ❺ and executing them ❻ in the meantime. If there are no tasks on the queue, the function calls `std::this_thread::yield()` to take a small break ❼ and give another thread a chance to put some work on the queue before it tries to take some off again the next time around.

For many purposes such a simple thread pool will suffice, especially if the tasks are entirely independent and don't return any values or perform any blocking operations. But there are also many circumstances where such a simple thread pool may not adequately address your needs and yet others where it can cause problems such as deadlock. Also, in the simple cases you may well be better served using `std::async` as in many of the examples in chapter 8. Throughout this chapter, we'll look at more complex thread pool implementations that have additional features either to address user needs or reduce the potential for problems. First up: waiting for the tasks we've submitted.

9.1.2 Waiting for tasks submitted to a thread pool

In the examples in chapter 8 that explicitly spawned threads, after dividing the work between threads, the master thread always waited for the newly spawned threads to finish, to ensure that the overall task was complete before returning to the caller. With thread pools, you'd need to wait for the *tasks* submitted to the thread pool to complete, rather than the worker threads themselves. This is similar to the way that the `std::async`-based examples in chapter 8 waited for the futures. With the simple thread pool from listing 9.1, you'd have to do this manually using the techniques from chapter 4: condition variables and futures. This adds complexity to the code; it would be better if you could wait for the tasks directly.

By moving that complexity into the thread pool itself, you *can* wait for the tasks directly. You can have the `submit()` function return a task handle of some description that you can then use to wait for the task to complete. This task handle would wrap the use of condition variables or futures, thus simplifying the code that uses the thread pool.

A special case of having to wait for the spawned task to finish occurs when the main thread needs a result computed by the task. You've seen this in examples throughout the book, such as the `parallel_accumulate()` function from chapter 2. In this case, you can combine the waiting with the result transfer through the use of futures. Listing 9.2 shows the changes required to the simple thread pool that allows you to wait for tasks to complete and then pass return values from the task to the waiting thread. Since `std::packaged_task<>` instances are not *copyable*, just *movable*, you can no longer use `std::function<>` for the queue entries, because `std::function<>` requires that the stored function objects are copy-constructible. Instead, you must use a custom function wrapper that can handle move-only types. This is a simple type-erasure class with a function call operator. You only need to handle functions that take no parameters and return `void`, so this is a straightforward virtual call in the implementation.

Listing 9.2 A thread pool with waitable tasks

```
class function_wrapper
{
    struct impl_base {
        virtual void call()=0;
        virtual ~impl_base() {}
    };
    std::unique_ptr<impl_base> impl;
    template<typename F>
    struct impl_type: impl_base
    {
        F f;
        impl_type(F&& f_): f(std::move(f_)) {}
        void call() { f(); }
    };
public:
    template<typename F>
    function_wrapper(F&& f):
        impl(new impl_type<F>(std::move(f)))
    {}

    void operator()() { impl->call(); }

    function_wrapper() = default;

    function_wrapper(function_wrapper&& other):
        impl(std::move(other.impl))
    {}
    function_wrapper& operator=(function_wrapper&& other)
    {
        impl=std::move(other.impl);
        return *this;
    }

    function_wrapper(const function_wrapper&)=delete;
    function_wrapper(function_wrapper&)=delete;
    function_wrapper& operator=(const function_wrapper&)=delete;
};

class thread_pool
{
    thread_safe_queue<function_wrapper> work_queue;   <─┐
                                                        │
    void worker_thread()                                │  Use function_
    {                                                   │  wrapper rather
        while(!done)                                    │  than std::function
        {                                               │
            function_wrapper task;                    <─┘
            if(work_queue.try_pop(task))
            {
                task();
            }
            else
            {
                std::this_thread::yield();
            }
        }
    }
public:
    template<typename FunctionType>
    std::future<typename std::result_of<FunctionType()>::type>     <─❶
        submit(FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type
            result_type;                                          <─❷

        std::packaged_task<result_type()> task(std::move(f));         <─❸
        std::future<result_type> res(task.get_future());          <─❹
        work_queue.push(std::move(task));                 <─❺
        return res;                        <─❻
    }
    // rest as before
};
```

First, the modified `submit()` function ❶ returns a `std::future<>` to hold the return value of the task and allow the caller to wait for the task to complete. This requires that you know the return type of the supplied function `f`, which is where `std::result_of<>` comes in: `std::result_of<FunctionType()>::type` is the type of the result of invoking an instance of type `FunctionType` (such as `f`) with no arguments. You use the same `std::result_of<>` expression for the `result_type typedef` ❷ inside the function.

You then wrap the function `f` in a `std::packaged_task<result_type()>` ❸, because `f` is a function or callable object that takes no parameters and returns an instance of type `result_type`, as we just deduced. You can now get your future from the `std::packaged_task<>` ❹, before pushing the task onto the queue ❺ and returning the future ❻. Note that you have to use `std::move()` when pushing the task onto the queue, because `std::packaged_task<>` isn't copyable. The queue now stores `function_wrapper` objects rather than `std::function<void()>` objects in order to handle this.

This pool thus allows you to wait for your tasks and have them return results. The next listing shows what the `parallel_accumulate` function looks like with such a thread pool.

Listing 9.3 `parallel_accumulate` using a thread pool with waitable tasks

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);

    if(!length)
        return init;

    unsigned long const block_size=25;
    unsigned long const num_blocks=(length+block_size-1)/block_size;     ❶

    std::vector<std::future<T> > futures(num_blocks-1);
    thread_pool pool;

    Iterator block_start=first;
    for(unsigned long i=0;i<(num_blocks-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        futures[i]=pool.submit(accumulate_block<Iterator,T>());     ❷
        block_start=block_end;
    }
    T last_result=accumulate_block<Iterator,T>()(block_start,last);
    T result=init;
    for(unsigned long i=0;i<(num_blocks-1);++i)
    {
        result+=futures[i].get();
    }
    result += last_result;
    return result;
}
```

When you compare this against listing 8.4, there are a couple of things to notice. First, you're working in terms of the number of blocks to use (`num_blocks` ❶) rather than the number of threads. In order to make the most use of the scalability of your thread pool, you need to divide the work into the smallest blocks that it's worth working with concurrently. When there are only a few threads in the pool, each thread will process many blocks, but as the number of threads grows with the hardware, the number of blocks processed in parallel will also grow.

You need to be careful when choosing the "smallest blocks that it's worth working with concurrently." There's an inherent overhead to submitting a task to a thread pool, having the worker thread run it, and passing the return value through a `std::future<>`, and for small tasks it's not worth the payoff. *If you choose too small a task size, the code may run more slowly with a thread pool than with one thread.*

Assuming the block size is sensible, you don't have to worry about packaging the tasks, obtaining the futures, or storing the `std::thread` objects so you can join with the threads later; the thread pool takes care of that. All you need to do is call `submit()` with your task ❷.

The thread pool takes care of the exception safety too. Any exception thrown by the task gets propagated through the future returned from `submit()`, and if the function exits with an exception, the thread pool destructor abandons any not-yet-completed tasks and waits for the pool threads to finish.

This works well for simple cases like this, where the tasks are independent. But it's not so good for situations where the tasks depend on other tasks also submitted to the thread pool.

9.1.3 Tasks that wait for other tasks

The Quicksort algorithm is an example that I've used throughout this book. It's simple in concept: the data to be sorted is partitioned into those items that go before a pivot item and those that go after it in the sorted sequence. These two sets of items are recursively sorted and then stitched back together to form a fully sorted set. When parallelizing this algorithm, you need to ensure that these recursive calls make use of the available concurrency.

Back in chapter 4, when I first introduced this example, we used `std::async` to run one of the recursive calls at each stage, letting the library choose between running it on a new thread and running it synchronously when the relevant `get()` was called. This works well, because each task is either running on its own thread or will be invoked when required.

When we revisited the implementation in chapter 8, you saw an alternative structure that used a fixed number of threads related to the available hardware concurrency. In this case, you used a stack of pending chunks that needed sorting. As each thread partitioned the data it was sorting, it added a new chunk to the stack for one of the sets of data and then sorted the other one directly. At this point, a straightforward wait for the sorting of the other chunk to complete would potentially deadlock, because you'd be consuming one of your limited number of threads waiting. It would be very easy to end up in a situation where all of the threads were waiting for chunks to be sorted and no threads were actually doing any sorting. We addressed this issue by having the threads pull chunks off the stack and sort them while the particular chunk they were waiting for was unsorted.

You'd get the same problem if you substituted a simple thread pool like the ones you've seen so far in this chapter instead of `std::async` in the example from chapter 4. There are now only a limited number of threads, and they might end up all waiting for tasks that haven't been scheduled because there are no free threads. You therefore need to use a solution similar to the one you used in chapter 8: process outstanding chunks while you're waiting for your chunk to complete. If you're using the thread pool to manage the list of tasks and their association with threads—which is, after all, the whole point of using a thread pool—you don't have access to the task list to do this. What you need to do is modify the thread pool to do this automatically.

The simplest way to do this is to add a new function on `thread_pool` to run a task from the queue and manage the loop yourself, so we'll go with that. Advanced thread pool implementations might add logic into the wait function or additional wait functions to handle this case, possibly prioritizing the task being waited for. The following listing shows the new `run_pending_task()` function, and a modified Quicksort to make use of it is shown in listing 9.5.

Listing 9.4 An implementation of `run_pending_task()`

```
void thread_pool::run_pending_task()
{
    function_wrapper task;
    if(work_queue.try_pop(task))
    {
        task();
    }
    else
    {
        std::this_thread::yield();
    }
}
```

This implementation of `run_pending_task()` is lifted straight out of the main loop of the `worker_thread()` function, which can now be modified to call the extracted `run_pending_task()`. This tries to take a task of the queue and run it if there is one; otherwise, it yields to allow the OS to reschedule the thread. The Quicksort implementation next is a lot simpler than the corresponding version from listing 8.1, because all the thread-management logic has been moved to the thread pool.

Listing 9.5 A thread pool–based implementation of Quicksort

```
template<typename T>
struct sorter                    <-1
{
    thread_pool pool;            <-2

    std::list<T> do_sort(std::list<T>& chunk_data)
    {
        if(chunk_data.empty())
        {
            return chunk_data;
        }

        std::list<T> result;
        result.splice(result.begin(),chunk_data,chunk_data.begin());
        T const& partition_val=*result.begin();
        typename std::list<T>::iterator divide_point=
            std::partition(chunk_data.begin(),chunk_data.end(),
                           [&](T const& val){return val<partition_val;});

        std::list<T> new_lower_chunk;
        new_lower_chunk.splice(new_lower_chunk.end(),
                               chunk_data,chunk_data.begin(),
                               divide_point);

        std::future<std::list<T> > new_lower=            ◁—❸
            pool.submit(std::bind(&sorter::do_sort,this,
                                  std::move(new_lower_chunk)));

        std::list<T> new_higher(do_sort(chunk_data));

        result.splice(result.end(),new_higher);
        while(!new_lower.wait_for(std::chrono::seconds(0)) ==
            std::future_status::timeout)
        {
            pool.run_pending_task();    ◁—❹
        }

        result.splice(result.begin(),new_lower.get());
        return result;
    }
};

template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    sorter<T> s;

    return s.do_sort(input);
}
```

Just as in listing 8.1, you've delegated the real work to the `do_sort()` member function of the `sorter` class template ❶, although in this case the class is only there to wrap the `thread_pool` instance ❷.

Your thread and task management is now reduced to submitting a task to the pool ❸ and running pending tasks while waiting ❹. This is much simpler than in listing 8.1, where you had to explicitly manage the threads and the stack of chunks to sort. When submitting the task to the pool, you use `std::bind()` to bind the `this` pointer to `do_sort()` and to supply the chunk to sort. In this case, you call `std::move()` on the `new_lower_chunk` as you pass it in, to ensure that the data is moved rather than copied.

Although this has now addressed the crucial deadlock-causing problem with tasks that wait for other tasks, this thread pool is still far from ideal. For starters, every call to `submit()` and every call to `run_pending_task()`accesses the same queue. You saw in chapter 8 how having a single set of data modified by multiple threads can have a detrimental effect on performance, so you need to somehow address this problem.

9.1.4 Avoiding contention on the work queue

Every time a thread calls `submit()` on a particular instance of the thread pool, it has to push a new item onto the single shared work queue. Likewise, the worker threads are continually popping items off the queue in order to run the tasks. This means that as the number of processors increases, there's increasing contention on the queue. This can be a real performance drain; even if you use a lock-free queue so there's no explicit waiting, cache ping-pong can be a substantial time sink.

One way to avoid cache ping-pong is to use a separate work queue per thread. Each thread then posts new items to its own queue and takes work from the global work queue only if there's no work on its own individual queue. The following listing shows an implementation that makes use of a `thread_local` variable to ensure that each thread has its own work queue, as well as the global one.

Listing 9.6 A thread pool with thread-local work queues

```
class thread_pool
{
    thread_safe_queue<function_wrapper> pool_work_queue;

    typedef std::queue<function_wrapper> local_queue_type;      ◁─❶
    static thread_local std::unique_ptr<local_queue_type>
        local_work_queue;        ◁─❷

    void worker_thread()
    {
        local_work_queue.reset(new local_queue_type);      ◁─❸

        while(!done)
        {
            run_pending_task();
        }
    }

public:
    template<typename FunctionType>
    std::future<typename std::result_of<FunctionType()>::type>
        submit(FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type result_type;

        std::packaged_task<result_type()> task(f);
        std::future<result_type> res(task.get_future());
        if(local_work_queue)                                   ◁─❹
        {
            local_work_queue->push(std::move(task));
        }
        else
        {
            pool_work_queue.push(std::move(task));      ◁─❺
        }
        return res;
    }
    void run_pending_task()
    {
        function_wrapper task;
        if(local_work_queue && !local_work_queue->empty())     ⑥
        {
            task=std::move(local_work_queue->front());
            local_work_queue->pop();
            task();
        }
        else if(pool_work_queue.try_pop(task))     ⑦
        {
            task();
        }
        else
        {
            std::this_thread::yield();
        }
    }
    // rest as before
};
```

We've used a `std::unique_ptr<>` to hold the thread-local work queue ❷ because we don't want non-pool threads to have one; this is initialized in the `worker_thread()` function before the processing loop ❸. The destructor of `std::unique_ptr<>` will ensure that the work queue is destroyed when the thread exits.

`submit()` then checks to see if the current thread has a work queue ❹. If it does, it's a pool thread, and you can put the task on the local queue; otherwise, you need to put the task on the pool queue as before ❺.

There's a similar check in `run_pending_task()` ❻, except this time you also need to check to see if there are any items on the local queue. If there are, you can take the front one and process it; notice that the local queue can be a plain `std::queue<>` ❶ because it's only ever accessed by the one thread. If there are no tasks on the local queue, you try the pool queue as before ❼.

This works fine for reducing contention, but when the distribution of work is uneven, it can easily result in one thread having a lot of work on its queue while the others have no work do to. For example, with the Quicksort example, only the top-most chunk would make it to the pool queue, because the remaining chunks would end up on the local queue of the worker thread that processed that one. This defeats the purpose of using a thread pool.

Thankfully, there is a solution to this: allow the threads to *steal* work from each other's queues if there's no work in their queue and no work in the global queue.

9.1.5 Work stealing

In order to allow a thread with no work to do to take work from another thread with a full queue, the queue must be accessible to the thread doing the stealing from `run_pending_tasks()`. This requires that each thread register its queue with the thread pool or be given one by the thread pool. Also, you must ensure that the data in the work queue is suitably synchronized and protected, so that your invariants are protected.

It's possible to write a lock-free queue that allows the owner thread to push and pop at one end while other threads can steal entries from the other, but the implementation of such a queue is beyond the scope of this book. In order to demonstrate the idea, we'll stick to using a mutex to protect the queue's data. We hope work stealing is a rare event, so there should be little contention on the mutex, and such a simple queue should therefore have minimal overhead. A simple lock-based implementation is shown here.

Listing 9.7 Lock-based queue for work stealing

```
class work_stealing_queue
{
private:
    typedef function_wrapper data_type;
    std::deque<data_type> the_queue;          <-1
    mutable std::mutex the_mutex;

public:
    work_stealing_queue()
    {}

    work_stealing_queue(const work_stealing_queue& other)=delete;
    work_stealing_queue& operator=(
        const work_stealing_queue& other)=delete;

    void push(data_type data)          <-2
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        the_queue.push_front(std::move(data));
    }

    bool empty() const
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        return the_queue.empty();
    }

    bool try_pop(data_type& res)          <-3
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        if(the_queue.empty())
        {
            return false;
        }

        res=std::move(the_queue.front());
        the_queue.pop_front();
        return true;
    }

    bool try_steal(data_type& res)          <-4
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        if(the_queue.empty())
        {
            return false;
        }

        res=std::move(the_queue.back());
        the_queue.pop_back();
        return true;
    }
};
```

This queue is a simple wrapper around a `std::deque<function_wrapper>` ❶ that protects all accesses with a mutex lock. Both `push()` ❷ and `try_pop()` ❸ work on the front of the queue, while `try_steal()` ❹ works on the back.

This actually means that this "queue" is a last-in-first-out stack for its own thread; the task most recently pushed on is the first one off again. This can help improve performance from a cache perspective, because the data related to that task is more likely to still be in the cache than the data related to a task pushed on the queue previously. Also, it maps nicely to algorithms such as Quicksort. In the previous implementation, each call to `do_sort()` pushes one item on the stack and then waits for it. By processing the most recent item first, you ensure that the chunk needed for the current call to complete is processed before the chunks needed for the other branches, thus reducing the number of active tasks and the total stack usage. `try_steal()` takes items from the opposite end of the queue to `try_pop()` in order to minimize contention; you could potentially use the techniques discussed in chapters 6 and 7 to enable concurrent calls to `try_pop()` and `try_steal()`.

OK, so you have your nice sparkly work queue that permits stealing; how do you use it in your thread pool? Here's one potential implementation.

Listing 9.8 A thread pool that uses work stealing

```
class thread_pool
{
    typedef function_wrapper task_type;

    std::atomic_bool done;
    thread_safe_queue<task_type> pool_work_queue;
    std::vector<std::unique_ptr<work_stealing_queue> > queues;      <-❶
    std::vector<std::thread> threads;
    join_threads joiner;

    static thread_local work_stealing_queue* local_work_queue;      <-❷
    static thread_local unsigned my_index;

    void worker_thread(unsigned my_index_)
    {
        my_index=my_index_;
        local_work_queue=queues[my_index].get();      <-❸
        while(!done)
        {
            run_pending_task();
        }
    }
    bool pop_task_from_local_queue(task_type& task)
    {
        return local_work_queue && local_work_queue->try_pop(task);
    }

    bool pop_task_from_pool_queue(task_type& task)
    {
        return pool_work_queue.try_pop(task);
    }

    bool pop_task_from_other_thread_queue(task_type& task)      <-❹
    {
        for(unsigned i=0;i<queues.size();++i)
        {
            unsigned const index=(my_index+i+1)%queues.size();      <-❺
            if(queues[index]->try_steal(task))
            {
                return true;
            }
        }

        return false;
    }

public:
    thread_pool():
        done(false),joiner(threads)
    {
        unsigned const thread_count=std::thread::hardware_concurrency();

        try
        {
            for(unsigned i=0;i<thread_count;++i)
            {
                queues.push_back(std::unique_ptr<work_stealing_queue>(      <-❻
                                     new work_stealing_queue));
                threads.push_back(
                    std::thread(&thread_pool::worker_thread,this,i));
            }
        }
        catch(...)
        {
            done=true;
            throw;
        }
    }

    ~thread_pool()
    {
        done=true;
    }

    template<typename FunctionType>
    std::future<typename std::result_of<FunctionType()>::type> submit(
        FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type result_type;
        std::packaged_task<result_type()> task(f);
        std::future<result_type> res(task.get_future());
        if(local_work_queue)
        {
            local_work_queue->push(std::move(task));
        }
        else
        {
            pool_work_queue.push(std::move(task));
        }
        return res;
    }

    void run_pending_task()
    {
        task_type task;
        if(pop_task_from_local_queue(task) ||     <-❼
           pop_task_from_pool_queue(task) ||          <-❽
           pop_task_from_other_thread_queue(task))        <-❾
        {
            task();
        }
        else
        {
            std::this_thread::yield();
        }
    }
};
```

This code is very similar to listing 9.6. The first difference is that each thread has a `work_stealing_queue` rather than a plain `std::queue<>` ❷. When each thread is created, rather than allocating its own work queue, the pool constructor allocates one ❻, which is then stored in the list of work queues for this pool ❶. The index of the queue in the list is then passed in to the thread function and used to retrieve the pointer to the queue ❸. This means that the thread pool can access the queue when trying to steal a task for a thread that has no work to do. `run_pending_task()` will now try to take a task from its thread's own queue ❼, take a task from the pool queue ❽, or take a task from the queue of another thread ❾.

`pop_task_from_other_thread_queue()` ❹ iterates through the queues belonging to all the threads in the pool, trying to steal a task from each in turn. In order to avoid every thread trying to steal from the first thread in the list, each thread starts at the next thread in the list, by offsetting the index of the queue to check by its own index ❺.

Now you have a working thread pool that's good for many potential uses. Of course, there are still a myriad of ways to improve it for any particular usage, but that's left as an exercise for the reader. One aspect in particular that hasn't been explored is the idea of dynamically resizing the thread pool to ensure that there's optimal CPU usage even when threads are blocked waiting for something such as I/O or a mutex lock.

Next on the list of "advanced" thread-management techniques is interrupting threads.

9.2 Interrupting threads

In many situations it's desirable to signal to a long-running thread that it's time to stop. This might be because it's a worker thread for a thread pool and the pool is now being destroyed, or it might be because the work being done by the thread has been explicitly canceled by the user, or a myriad of other reasons. Whatever the reason, the idea is the same: you need to signal from one thread that another should stop before it reaches the natural end of its processing, and you need to do this in a way that allows that thread to terminate nicely rather than abruptly pulling the rug from under it.

You could potentially design a separate mechanism for every case where you need to do this, but that would be overkill. Not only does a common mechanism make it easier to write the code on subsequent occasions, but it can allow you to write code that can be interrupted, without having to worry about where that code is being used. The C++11 Standard doesn't provide such a mechanism, but it's relatively straightforward to build one. Let's look at how you can do that, starting from the point of view of the interface for launching and interrupting a thread rather than that of the thread being interrupted.

9.2.1 Launching and interrupting another thread

To start with, let's look at the external interface. What do you need from an interruptible thread? At the basic level, all you need is the same interface as you have for `std::thread`, with an additional `interrupt()` function:

```
class interruptible_thread
{
public:
    template<typename FunctionType>
    interruptible_thread(FunctionType f);
    void join();
    void detach();
    bool joinable() const;
    void interrupt();
};
```

Internally, you can use `std::thread` to manage the thread itself and use some custom data structure to handle the interruption. Now, what about from the point of view of the thread itself? At the most basic level you want to be able to say, "I can be interrupted here"—you want an *interruption point*. For this to be usable without having to pass down additional data, it needs to be a simple function that can be called without any parameters: `interruption_point()`. This implies that the interruption-specific data structure needs to be accessible through a `thread_local` variable that's set when the thread is started, so that when a thread calls your `interruption_point()` function, it checks the data structure for the currently executing thread. We'll look at the implementation of `interruption_point()` later.

This `thread_local` flag is the primary reason you can't just use plain `std::thread` to manage the thread; it needs to be allocated in a way that the `interruptible_thread` instance can access, as well as the newly started thread. You can do this by wrapping the supplied function before you pass it to `std::thread` to actually launch the thread in the constructor, as shown in the next listing.

Listing 9.9 Basic implementation of `interruptible_thread`

```
class interrupt_flag
{
public:
    void set();
    bool is_set() const;
};
thread_local interrupt_flag this_thread_interrupt_flag;     ❶

class interruptible_thread
{
    std::thread internal_thread;
    interrupt_flag* flag;
public:
    template<typename FunctionType>
    interruptible_thread(FunctionType f)
    {
        std::promise<interrupt_flag*> p;     ❷
        internal_thread=std::thread([f,&p]{              ❸
                p.set_value(&this_thread_interrupt_flag);
                f();                                      ❹
            });
        flag=p.get_future().get();     ❺
    }
    void interrupt()
    {
        if(flag)
        {
            flag->set();     ❻
        }
    }
};
```

The supplied function `f` is wrapped in a lambda function ❸ that holds a copy of `f` and a reference to the local promise `p` ❷. The lambda sets the value of the promise to the address of the `this_thread_interrupt_flag` (which is declared `thread_local` ❶) for the new thread before invoking the copy of the supplied function ❹. The calling thread then waits for the future associated with the promise to become ready and stores the result in the `flag` member variable ❺. Note that even though the lambda is running on the new thread and has a dangling reference to the local variable `p`, this is OK because the `interruptible_thread` constructor waits until `p` is no longer referenced by the new thread before returning. Note that this implementation doesn't take account of handling joining with the thread, or detaching it. You need to ensure that the `flag` variable is cleared when the thread exits, or is detached, to avoid a dangling pointer.

The `interrupt()` function is then relatively straightforward: if you have a valid pointer to an interrupt flag, you have a thread to interrupt, so you can just set the flag ❻. It's then up to the interrupted thread what it does with the interruption. Let's explore that next.

9.2.2 Detecting that a thread has been interrupted

You can now set the interruption flag, but that doesn't do you any good if the thread doesn't actually check whether it's being interrupted. In the simplest case you can do this with an `interruption_point()` function; you can call this function at a point where it's safe to be interrupted, and it throws a `thread_interrupted` exception if the flag is set:

```
void interruption_point()
{
    if(this_thread_interrupt_flag.is_set())
    {
        throw thread_interrupted();
    }
}
```

You can use such a function by calling it at convenient points within your code:

```
void foo()
{
    while(!done)
    {
        interruption_point();
        process_next_item();
    }
}
```

Although this works, it's not ideal. Some of the best places for interrupting a thread are where it's blocked waiting for something, which means that the thread isn't running in order to call `interruption_point()`! What you need here is a means for waiting for something in an interruptible fashion.

9.2.3 Interrupting a condition variable wait

OK, so you can detect interruptions at carefully chosen places in your code, with explicit calls to `interruption_point()`, but that doesn't help when you want to do a blocking wait, such as waiting for a condition variable to be notified. You need a new function—`interruptible_wait()`—which you can then overload for the various things you might want to wait for, and you can work out how to interrupt the waiting. I've already mentioned that one thing you might be waiting for is a condition variable, so let's start there: what do you need to do in order to be able to interrupt a wait on a condition variable? The simplest thing that would work is to notify the condition variable once you've set the interrupt flag, and put an interruption point immediately after the wait. But for this to work, you'd have to notify all threads waiting on the condition variable in order to ensure that your thread of interest wakes up. Waiters have to handle spurious wake-ups anyway, so other threads would handle this the same as a spurious wake-up—they wouldn't be able to tell the difference. The `interrupt_flag` structure would need to be able to store a pointer to a condition variable so that it can be notified in a call to `set()`. One possible implementation of `interruptible_wait()` for condition variables might look like the following listing.

Listing 9.10 A broken version of `interruptible_wait` for `std::condition_variable`

```
void interruptible_wait(std::condition_variable& cv,
                        std::unique_lock<std::mutex>& lk)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);      ◁─❶
    cv.wait(lk);                                          ◁─❷
    this_thread_interrupt_flag.clear_condition_variable();      ◁─❸
    interruption_point();
}
```

Assuming the presence of some functions for setting and clearing an association of a condition variable with an interrupt flag, this code is nice and simple. It checks for interruption, associates the condition variable with the `interrupt_flag` for the current thread ❶, waits on the condition variable ❷, clears the association with the condition variable ❸, and checks for interruption again. If the thread is interrupted during the wait on the condition variable, the interrupting thread will broadcast the condition variable and wake you from the wait, so you can check for interruption. Unfortunately, this code is *broken*: there are two problems with it. The first problem is relatively obvious if you have your exception safety hat on: `std::condition_variable::wait()` can throw an exception, so you might exit the function without removing the association of the interrupt flag with the condition variable. This is easily fixed with a structure that removes the association in its destructor.

The second, less-obvious problem is that there's a race condition. If the thread is interrupted after the initial call to `interruption_point()`, but before the call to `wait()`, then it doesn't matter whether the condition variable has been associated with the interrupt flag, because *the thread isn't waiting and so can't be woken by a notify on the condition variable*. You need to ensure that the thread can't be notified between the last check for interruption and the call to `wait()`. Without delving into the internals of `std::condition_variable`, you have only one way of doing that: use the mutex held by `lk` to protect this too, which requires passing it in on the call to `set_condition_variable()`. Unfortunately, this creates its own problems: you'd be passing a reference to a mutex whose lifetime you don't know to another thread (the thread doing the interrupting) for that thread to lock (in the call to `interrupt()`), without knowing whether that thread has locked the mutex already when it makes the call. This has the potential for deadlock *and* the potential to access a mutex after it has already been destroyed, so it's a nonstarter. It would be rather too restrictive if you couldn't *reliably* interrupt a condition variable wait—you can do almost as well without a special `interruptible_wait()`—so what other options do you have? One option is to put a timeout on the wait; use `wait_for()` rather than `wait()` with a small timeout value (such as 1 ms). This puts an upper limit on how long the thread will have to wait before it sees the interruption (subject to the tick granularity of the clock). If you do this, the waiting thread will see rather more "spurious" wakes resulting from the timeout, but it can't easily be helped. Such an implementation is shown in the next listing, along with the corresponding implementation of `interrupt_flag`.

Listing 9.11 Using a timeout in `interruptible_wait` for `std::condition_variable`

```
class interrupt_flag
{
    std::atomic<bool> flag;
    std::condition_variable* thread_cond;
    std::mutex set_clear_mutex;

public:
    interrupt_flag():
        thread_cond(0)
    {}

    void set()
    {
        flag.store(true,std::memory_order_relaxed);
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        if(thread_cond)
        {
            thread_cond->notify_all();
        }
    }

    bool is_set() const
    {
        return flag.load(std::memory_order_relaxed);
    }

    void set_condition_variable(std::condition_variable& cv)
    {
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        thread_cond=&cv;
    }

    void clear_condition_variable()
    {
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        thread_cond=0;
    }

    struct clear_cv_on_destruct
    {
        ~clear_cv_on_destruct()
        {
            this_thread_interrupt_flag.clear_condition_variable();
        }
    };
};

void interruptible_wait(std::condition_variable& cv,
                        std::unique_lock<std::mutex>& lk)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);
    interrupt_flag::clear_cv_on_destruct guard;
    interruption_point();
    cv.wait_for(lk,std::chrono::milliseconds(1));
    interruption_point();
}
```

If you have the predicate that's being waited for, then the 1 ms timeout can be completely hidden inside the predicate loop:

```
template<typename Predicate>
void interruptible_wait(std::condition_variable& cv,
                        std::unique_lock<std::mutex>& lk,
                        Predicate pred)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);
    interrupt_flag::clear_cv_on_destruct guard;
    while(!this_thread_interrupt_flag.is_set() && !pred())
    {
        cv.wait_for(lk,std::chrono::milliseconds(1));
    }
    interruption_point();
}
```

This will result in the predicate being checked more often than it might otherwise be, but it's easily used in place of a plain call to `wait()`. The variants with timeouts are easily implemented: wait either for the time specified, or 1 ms, whichever is shortest. OK, so `std::condition_variable` waits are now taken care of; what about `std::condition_variable_any`? Is this the same, or can you do better?

9.2.4 Interrupting a wait on std::condition_variable_any

`std::condition_variable_any` differs from `std::condition_variable` in that it works with *any* lock type rather than just `std::unique_lock<std::mutex>`. It turns out that this makes things much easier, and you *can* do better with `std::condition_variable_any` than you could with `std::condition_variable`. Because it works with *any* lock type, you can build your own lock type that locks/unlocks both the internal `set_clear_mutex` in your `interrupt_flag` *and* the lock supplied to the wait call, as shown here.

Listing 9.12 `interruptible_wait` for `std::condition_variable_any`

```
class interrupt_flag
{
    std::atomic<bool> flag;
    std::condition_variable* thread_cond;
    std::condition_variable_any* thread_cond_any;
    std::mutex set_clear_mutex;

public:
    interrupt_flag():
```

```
    thread_cond(0),thread_cond_any(0)
{}

void set()
{
    flag.store(true,std::memory_order_relaxed);
    std::lock_guard<std::mutex> lk(set_clear_mutex);
    if(thread_cond)
    {
        thread_cond->notify_all();
    }
    else if(thread_cond_any)
    {
        thread_cond_any->notify_all();
    }
}

template<typename Lockable>
void wait(std::condition_variable_any& cv,Lockable& lk)
{
    struct custom_lock
    {
        interrupt_flag* self;
        Lockable& lk;

        custom_lock(interrupt_flag* self_,
                    std::condition_variable_any& cond,
                    Lockable& lk_):
            self(self_),lk(lk_)
        {
            self->set_clear_mutex.lock();          ❶
            self->thread_cond_any=&cond;            ❷
        }

        void unlock()        ❸
        {
            lk.unlock();
            self->set_clear_mutex.unlock();
        }

        void lock()
        {
            std::lock(self->set_clear_mutex,lk);     ❹
        }

        ~custom_lock()
        {
            self->thread_cond_any=0;           ❺
            self->set_clear_mutex.unlock();
        }
    };
    custom_lock cl(this,cv,lk);
    interruption_point();
    cv.wait(cl);
    interruption_point();
}
```

```
    // rest as before
};

template<typename Lockable>
void interruptible_wait(std::condition_variable_any& cv,
                        Lockable& lk)
{
    this_thread_interrupt_flag.wait(cv,lk);
}
```

Your custom lock type acquires the lock on the internal `set_clear_mutex` when it's constructed ❶ and then sets the `thread_cond_any` pointer to refer to the `std::condition_variable_any` passed in to the constructor ❷. The `Lockable` reference is stored for later; this must already be locked. You can now check for an interruption without worrying about races. If the interrupt flag is set at this point, it was set before you acquired the lock on `set_clear_mutex`. When the condition variable calls your `unlock()` function inside `wait()`,you unlock the `Lockable` object *and the internal* `set_clear_mutex` ❸. This allows threads that are trying to interrupt you to acquire the lock on `set_clear_mutex` and check the `thread_cond_any` pointer *once you're inside the* `wait()` *call* but not before. This is exactly what you were after (but couldn't manage) with `std::condition_variable`. Once `wait()` has finished waiting (either because it was notified or because of a spurious wake), it will call your `lock()` function, which again acquires the lock on the internal `set_clear_mutex` and the lock on the `Lockable` object ❹. You can now check again for interruptions that happened during the `wait()` call before clearing the `thread_cond_any` pointer in your `custom_lock` destructor ❺, where you also unlock the `set_clear_mutex`.

9.2.5 Interrupting other blocking calls

That rounds up interrupting condition variable waits, but what about other blocking waits: mutex locks, waiting for futures, and the like? In general you have to go for the timeout option you used for `std::condition_variable` because there's no way to interrupt the wait short of actually fulfilling the condition being waited for, without access to the internals of the mutex or future. But with those other things you do know what you're waiting for, so you can loop within the `interruptible_wait()` function. As an example, here's an overload of `interruptible_wait()` for a `std::future<>`:

```
template<typename T>
void interruptible_wait(std::future<T>& uf)
{
   while(!this_thread_interrupt_flag.is_set())
   {
       if(uf.wait_for(lk,std::chrono::milliseconds(1)==
           std::future_status::ready)
           break;
   }
   interruption_point();
}
```

This waits until either the interrupt flag is set or the future is ready but does a blocking wait on the future for 1 ms at a time. This means that on average it will be around

0.5 ms before an interrupt request is acknowledged, assuming a high-resolution clock. The `wait_for` will typically wait at least a whole clock tick, so if your clock ticks every 15 ms, you'll end up waiting around 15 ms rather than 1 ms. This may or may not be acceptable, depending on the circumstances. You can always reduce the timeout if necessary (and the clock supports it). The downside of reducing the timeout is that the thread will wake more often to check the flag, and this will increase the task-switching overhead.

OK, so we've looked at how you might detect interruption, with the `interruption_point()` and `interruptible_wait()` functions, but how do you handle that?

9.2.6 Handling interruptions

From the point of view of the thread being interrupted, an interruption is just a `thread_interrupted` exception, which can therefore be handled just like any other exception. In particular, you can catch it in a standard `catch` block:

```
try
{
    do_something();
}
catch(thread_interrupted&)
{
    handle_interruption();
}
```

This means that you could catch the interruption, handle it in some way, and then carry on regardless. If you do this, and another thread calls `interrupt()` again, your thread will be interrupted again the next time it calls an interruption point. You might want to do this if your thread is performing a series of independent tasks; interrupting one task will cause that task to be abandoned, and the thread can then move on to performing the next task in the list.

Because `thread_interrupted` is an exception, all the usual exception-safety precautions must also be taken when calling code that can be interrupted, in order to ensure that resources aren't leaked, and your data structures are left in a coherent state. Often, it will be desirable to let the interruption terminate the thread, so you can just let the exception propagate up. But if you let exceptions propagate out of the thread function passed to the `std::thread` constructor, `std::terminate()` will be called, and the whole program will be terminated. In order to avoid having to remember to put a `catch (thread_interrupted)` handler in every function you pass to `interruptible_thread`, you can instead put that `catch` block inside the wrapper you use for initializing the `interrupt_flag`. This makes it safe to allow the interruption exception to propagate unhandled, because it will then terminate just that individual thread. The initialization of the thread in the `interruptible_thread` constructor now looks like this:

```
internal_thread=std::thread([f,&p]{
        p.set_value(&this_thread_interrupt_flag);
```

```
            try
            {
                f();
            }
            catch(thread_interrupted const&)
            {}
        });
```

Let's now look at a concrete example where interruption is useful.

9.2.7 Interrupting background tasks on application exit

Consider for a moment a desktop search application. As well as interacting with the user, the application needs to monitor the state of the filesystem, identifying any changes and updating its index. Such processing is typically left to a background thread, in order to avoid affecting the responsiveness of the GUI. This background thread needs to run for the entire lifetime of the application; it will be started as part of the application initialization and left to run until the application is shut down. For such an application this is typically only when the machine itself is being shut down, because the application needs to run the whole time in order to maintain an up-to-date index. In any case, when the application is being shut down, you need to close down the background threads in an orderly manner; one way to do this is by interrupting them.

The following listing shows a sample implementation of the thread-management parts of such a system.

Listing 9.13 Monitoring the filesystem in the background

```
std::mutex config_mutex;
std::vector<interruptible_thread> background_threads;

void background_thread(int disk_id)
{
    while(true)
    {
        interruption_point();                          <-- 1
        fs_change fsc=get_fs_changes(disk_id);              <-- 2
        if(fsc.has_changes())
        {
            update_index(fsc);     <-- 3
        }
    }
}

void start_background_processing()
{
    background_threads.push_back(
        interruptible_thread(background_thread,disk_1));
    background_threads.push_back(
        interruptible_thread(background_thread,disk_2));
}

int main()
{
    start_background_processing();      ◁─❹
    process_gui_until_exit();                        ◁─❺
    std::unique_lock<std::mutex> lk(config_mutex);
    for(unsigned i=0;i<background_threads.size();++i)
    {
        background_threads[i].interrupt();     ◁─❻
    }
    for(unsigned i=0;i<background_threads.size();++i)
    {
        background_threads[i].join();     ◁─❼
    }
}
```

At startup, the background threads are launched ❹. The main thread then proceeds with handling the GUI ❺. When the user has requested that the application exit, the background threads are interrupted ❻, and then the main thread waits for each background thread to complete before exiting ❼. The background threads sit in a loop, checking for disk changes ❷ and updating the index ❸. Every time around the loop they check for interruption by calling `interruption_point()` ❶.

Why do you interrupt all the threads before waiting for any? Why not interrupt each and then wait for it before moving on to the next? The answer is *concurrency*. Threads will likely not finish immediately when they're interrupted, because they have to proceed to the next interruption point and then run any destructor calls and exception-handling code necessary before they exit. By joining with each thread immediately, you therefore cause the interrupting thread to wait, *even though it still has useful work it could do*—interrupt the other threads. Only when you have no more work to do (all the threads have been interrupted) do you wait. This also allows all the threads being interrupted to process their interruptions in parallel and potentially finish sooner.

This interruption mechanism could easily be extended to add further interruptible calls or to disable interruptions across a specific block of code, but this is left as an exercise for the reader.

9.3 Summary

In this chapter, we've looked at various "advanced" thread-management techniques: thread pools and interrupting threads. You've seen how the use of local work queues and work stealing can reduce the synchronization overhead and potentially improve the throughput of the thread pool and how running other tasks from the queue while waiting for a subtask to complete can eliminate the potential for deadlock.

We've also looked at various ways of allowing one thread to interrupt the processing of another, such as the use of specific interruption points and functions that perform what would otherwise be a blocking wait in a way that can be interrupted.