C++ Concurrency IN ACTION

Practical Multithreading

Anthony Williams

MANNING

Table of Contents

6 Designing lock-based concurrent data structures

This chapter covers

- What it means to design data structures for concurrency
- Guidelines for doing so
- Example implementations of data structures designed for concurrency

In the last chapter we looked at the low-level details of atomic operations and the memory model. In this chapter we'll take a break from the low-level details (although we'll need them for chapter 7) and think about data structures.

The choice of data structure to use for a programming problem can be a key part of the overall solution, and parallel programming problems are no exception. If a data structure is to be accessed from multiple threads, either it must be completely immutable so the data never changes and no synchronization is necessary, or the program must be designed to ensure that changes are correctly synchronized between threads. One option is to use a separate mutex and external locking to protect the data, using the techniques we looked at in chapters 3 and 4, and another is to design the data structure itself for concurrent access.

When designing a data structure for concurrency, you can use the basic building blocks of multithreaded applications from earlier chapters, such as mutexes

and condition variables. Indeed, you've already seen a couple of examples showing how to combine these building blocks to write data structures that are safe for concurrent access from multiple threads.

In this chapter we'll start by looking at some general guidelines for designing data structures for concurrency. We'll then take the basic building blocks of locks and condition variables and revisit the design of those basic data structures before moving on to more complex data structures. In chapter 7 we'll look at how to go right back to basics and use the atomic operations described in chapter 5 to build data structures without locks.

So, without further ado, let's look at what's involved in designing a data structure for concurrency.

6.1 What does it mean to design for concurrency?

At the basic level, designing a data structure for concurrency means that multiple threads can access the data structure concurrently, either performing the same or distinct operations, and each thread will see a self-consistent view of the data structure. No data will be lost or corrupted, all invariants will be upheld, and there'll be no problematic race conditions. Such a data structure is said to be *thread-safe*. In general, a data structure will be safe only for particular types of concurrent access. It may be possible to have multiple threads performing one type of operation on the data structure concurrently, whereas another operation requires exclusive access by a single thread. Alternatively, it may be safe for multiple threads to access a data structure concurrently if they're performing *different* actions, whereas multiple threads performing the *same* action would be problematic.

Truly designing for concurrency means more than that, though: it means providing the *opportunity for concurrency* to threads accessing the data structure. By its very nature, a mutex provides *mutual exclusion*: only one thread can acquire a lock on the mutex at a time. A mutex protects a data structure by explicitly *preventing* true concurrent access to the data it protects.

This is called *serialization*: threads take turns accessing the data protected by the mutex; they must access it serially rather than concurrently. Consequently, you must put careful thought into the design of the data structure to enable true concurrent access. Some data structures have more scope for true concurrency than others, but in all cases the idea is the same: the smaller the protected region, the fewer operations are serialized, and the greater the potential for concurrency.

Before we look at some data structure designs, let's have a quick look at some simple guidelines for what to consider when designing for concurrency.

6.1.1 Guidelines for designing data structures for concurrency

As I just mentioned, you have two aspects to consider when designing data structures for concurrent access: ensuring that the accesses are *safe* and *enabling* genuine concurrent access. I covered the basics of how to make the data structure thread-safe back in chapter 3:

- Ensure that no thread can see a state where the invariants of the data structure have been broken by the actions of another thread.
- Take care to avoid race conditions inherent in the interface to the data structure by providing functions for complete operations rather than for operation steps.
- Pay attention to how the data structure behaves in the presence of exceptions to ensure that the invariants are not broken.
- Minimize the opportunities for deadlock when using the data structure by restricting the scope of locks and avoiding nested locks where possible.

Before you think about any of these details, it's also important to think about what constraints you wish to put on the users of the data structure; if one thread is accessing the data structure through a particular function, which functions are safe to call from other threads?

This is actually quite a crucial question to consider. Generally constructors and destructors require exclusive access to the data structure, but it's up to the user to ensure that they're not accessed before construction is complete or after destruction has started. If the data structure supports assignment, `swap()`, or copy construction, then as the designer of the data structure, you need to decide whether these operations are safe to call concurrently with other operations or whether they require the user to ensure exclusive access even though the majority of functions for manipulating the data structure may be called from multiple threads concurrently without problem.

The second aspect to consider is that of enabling genuine concurrent access. I can't offer much in the way of guidelines here; instead, here's a list of questions to ask yourself as the data structure designer:

- Can the scope of locks be restricted to allow some parts of an operation to be performed outside the lock?
- Can different parts of the data structure be protected with different mutexes?
- Do all operations require the same level of protection?
- Can a simple change to the data structure improve the opportunities for concurrency without affecting the operational semantics?

All these questions are guided by a single idea: how can you minimize the amount of serialization that must occur and enable the greatest amount of true concurrency? It's not uncommon for data structures to allow concurrent access from multiple threads that merely read the data structure, whereas a thread that can modify the data structure must have exclusive access. This is supported by using constructs like `boost::shared_mutex`. Likewise, as you'll see shortly, it's quite common for a data structure to support concurrent access from threads performing different operations while serializing threads that try to perform the same operation.

The simplest thread-safe data structures typically use mutexes and locks to protect the data. Although there are issues with this, as you saw in chapter 3, it's relatively easy to ensure that only one thread is accessing the data structure at a time. To ease you into the design of thread-safe data structures, we'll stick to looking at such lock-based

data structures in this chapter and leave the design of concurrent data structures without locks for chapter 7.

6.2 Lock-based concurrent data structures

The design of lock-based concurrent data structures is all about ensuring that the right mutex is locked when accessing the data and ensuring that the lock is held for a minimum amount of time. This is hard enough when there's just one mutex protecting a data structure. You need to ensure that data can't be accessed outside the protection of the mutex lock and that there are no race conditions inherent in the interface, as you saw in chapter 3. If you use separate mutexes to protect separate parts of the data structure, these issues are compounded, and there's now also the possibility of deadlock if the operations on the data structure require more than one mutex to be locked. You therefore need to consider the design of a data structure with multiple mutexes even more carefully than the design of a data structure with a single mutex.

In this section you'll apply the guidelines from section 6.1.1 to the design of several simple data structures, using mutexes and locks to protect the data. In each case you'll seek out the opportunities for enabling greater concurrency while ensuring that the data structure remains thread-safe.

Let's start by looking at the stack implementation from chapter 3; it's one of the simplest data structures around, and it uses only a single mutex. Is it really thread-safe? How does it fare from the point of view of achieving true concurrency?

6.2.1 A thread-safe stack using locks

The thread-safe stack from chapter 3 is reproduced in the following listing. The intent is to write a thread-safe data structure akin to `std::stack<>`, which supports pushing data items onto the stack and popping them off again.

Listing 6.1 A class definition for a thread-safe stack

```
#include <exception>

struct empty_stack: std::exception
{
    const char* what() const throw();
};

template<typename T>
class threadsafe_stack
{
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack(){}
    threadsafe_stack(const threadsafe_stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;
```

```
    }
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));      <-- 1
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();    <-- 2
        std::shared_ptr<T> const res(
           std::make_shared<T>(std::move(data.top())));    <-- 3
        data.pop();          <-- 4
        return res;
    }
    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        value=std::move(data.top());          <-- 5
        data.pop();                  <-- 6
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

Let's look at each of the guidelines in turn, and see how they apply here.

First, as you can see, the basic thread safety is provided by protecting each member function with a lock on the mutex, `m`. This ensures that only one thread is actually accessing the data at any one time, so provided each member function maintains the invariants, no thread can see a broken invariant.

Second, there's a potential for a race condition between `empty()` and either of the `pop()` functions, but because the code explicitly checks for the contained stack being empty while holding the lock in `pop()`, this race condition isn't problematic. By returning the popped data item directly as part of the call to `pop()`, you avoid a potential race condition that would be present with separate `top()` and `pop()` member functions such as those in `std::stack<>`.

Next, there are a few potential sources of exceptions. Locking a mutex may throw an exception, but not only is this likely to be exceedingly rare (because it indicates a problem with the mutex or a lack of system resources), it's also the first operation in each member function. Because no data has been modified, this is safe. Unlocking a mutex can't fail, so that's always safe, and the use of `std::lock_guard<>` ensures that the mutex is never left locked.

The call to `data.push()` ❶ may throw an exception if either copying/moving the data value throws an exception or not enough memory can be allocated to extend the

underlying data structure. Either way, `std::stack<>` guarantees it will be safe, so that's not a problem either.

In the first overload of `pop()`, the code itself might throw an `empty_stack` exception ❷, but nothing has been modified, so that's safe. The creation of `res` ❸ might throw an exception though for a couple of reasons: the call to `std::make_shared` might throw because it can't allocate memory for the new object and the internal data required for reference counting, or the copy constructor or move constructor of the data item to be returned might throw when copying/moving into the freshly allocated memory. In both cases, the C++ runtime and Standard Library ensure that there are no memory leaks and the new object (if any) is correctly destroyed. Because you *still* haven't modified the underlying stack, you're still OK. The call to `data.pop()` ❹ is guaranteed not to throw, as is the return of the result, so this overload of `pop()` is exception-safe.

The second overload of `pop()` is similar, except this time it's the copy assignment or move assignment operator that can throw ❺ rather than the construction of a new object and a `std::shared_ptr` instance. Again, you don't actually modify the data structure until the call to `data.pop()` ❻, which is still guaranteed not to throw, so this overload is exception-safe too.

Finally, `empty()` doesn't modify any data, so that's exception-safe.

There are a couple of opportunities for deadlock here, because you call user code while holding a lock: the copy constructor or move constructor ❶, ❸ and copy assignment or move assignment operator ❺ on the contained data items, as well as potentially a user-defined `operator new`. If these functions either call member functions on the stack that the item is being inserted into or removed from or require a lock of any kind and another lock was held when the stack member function was invoked, there's the possibility of deadlock. However, it's sensible to require that users of the stack be responsible for ensuring this; you can't reasonably expect to add an item onto a stack or remove it from a stack without copying it or allocating memory for it.

Because all the member functions use a `std::lock_guard<>` to protect the data, it's safe for any number of threads to call the `stack` member functions. The only member functions that aren't safe are the constructors and destructors, but this isn't a particular problem; the object can be constructed only once and destroyed only once. Calling member functions on an incompletely constructed object or a partially destructed object is never a good idea whether done concurrently or not. As a consequence, the user must ensure that other threads aren't able to access the stack until it's fully constructed and must ensure that all threads have ceased accessing the stack before it's destroyed.

Although it's safe for multiple threads to call the member functions concurrently, because of the use of locks, only one thread is ever actually doing any work in the stack data structure at a time. This *serialization* of threads can potentially limit the performance of an application where there's significant contention on the `stack`: while a thread is waiting for the lock, it isn't doing any useful work. Also, the stack doesn't

provide any means for waiting for an item to be added, so if a thread needs to wait, it must periodically call `empty()` or just call `pop()` and catch the `empty_stack` exceptions. This makes this stack implementation a poor choice if such a scenario is required, because a waiting thread must either consume precious resources checking for data or the user must write external wait and notification code (for example, using condition variables), which might render the internal locking unnecessary and therefore wasteful. The queue from chapter 4 shows a way of incorporating such waiting into the data structure itself using a condition variable inside the data structure, so let's look at that next.

6.2.2 A thread-safe queue using locks and condition variables

The thread-safe queue from chapter 4 is reproduced in listing 6.2. Much like the stack was modeled after `std::stack<>`, this queue is modeled after `std::queue<>`. Again, the interface differs from that of the standard container adaptor because of the constraints of writing a data structure that's safe for concurrent access from multiple threads.

Listing 6.2 The full class definition for a thread-safe queue using condition variables

```
template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(data));
        data_cond.notify_one();          <-(1)
    }

    void wait_and_pop(T& value)    <-(2)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=std::move(data_queue.front());
        data_queue.pop();
    }

    std::shared_ptr<T> wait_and_pop()    <-(3)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});    <-(4)
        std::shared_ptr<T> res(
            std::make_shared<T>(std::move(data_queue.front())));
```

```
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=std::move(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>();    <-- 5
        std::shared_ptr<T> res(
            std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

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

The structure of the queue implementation shown in listing 6.2 is similar to the stack from listing 6.1, except for the call to `data_cond.notify_one()` in `push()` ❶ and the `wait_and_pop()` functions ❷, ❸. The two overloads of `try_pop()` are almost identical to the `pop()` functions from listing 6.1, except that they don't throw an exception if the queue is empty. Instead, they return either a `bool` value indicating whether a value was retrieved or a `NULL` pointer if no value could be retrieved by the pointer-returning overload ❺. This would also have been a valid way of implementing the stack. So, if you exclude the `wait_and_pop()` functions, the analysis you did for the stack applies just as well here.

The new `wait_and_pop()` functions are a solution to the problem of waiting for a queue entry that you saw with the stack; rather than continuously calling `empty()`, the waiting thread can just call `wait_and_pop()` and the data structure will handle the waiting with a condition variable. The call to `data_cond.wait()` won't return until the underlying queue has at least one element, so you don't have to worry about the possibility of an empty queue at this point in the code, and the data is still protected with the lock on the mutex. These functions don't therefore add any new race conditions or possibilities for deadlock, and the invariants will be upheld.

There's a slight twist with regard to exception safety in that if more than one thread is waiting when an entry is pushed onto the queue, only one thread will be

woken by the call to `data_cond.notify_one()`. However, if that thread then throws an exception in `wait_and_pop()`, such as when the new `std::shared_ptr<>` is constructed ❹, none of the other threads will be woken. If this isn't acceptable, the call is readily replaced with `data_cond.notify_all()`, which will wake all the threads but at the cost of most of them then going back to sleep when they find that the queue is empty after all. A second alternative is to have `wait_and_pop()` call `notify_one()` if an exception is thrown, so that another thread can attempt to retrieve the stored value. A third alternative is to move the `std::shared_ptr<>` initialization to the `push()` call and store `std::shared_ptr<>` instances rather than direct data values. Copying the `std::shared_ptr<>` out of the internal `std::queue<>` then can't throw an exception, so `wait_and_pop()` is safe again. The following listing shows the queue implementation revised with this in mind.

Listing 6.3 A thread-safe queue holding `std::shared_ptr<>` instances

```
template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<std::shared_ptr<T> > data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}

    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=std::move(*data_queue.front());          <-- 1
        data_queue.pop();
    }

    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=std::move(*data_queue.front());          <-- 2
        data_queue.pop();
        return true;
    }

    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=data_queue.front();     <-- 3
        data_queue.pop();
        return res;
    }
```

```
    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=data_queue.front();    <-- 4
        data_queue.pop();
        return res;
    }

    void push(T new_value)
    {
        std::shared_ptr<T> data(
            std::make_shared<T>(std::move(new_value)));    <-- 5
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }

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

The basic consequences of holding the data by `std::shared_ptr<>` are straightforward: the pop functions that take a reference to a variable to receive the new value now have to dereference the stored pointer ❶, ❷, and the pop functions that return a `std::shared_ptr<>` instance can just retrieve it from the queue ❸, ❹ before returning it to the caller.

If the data is held by `std::shared_ptr<>`, there's an additional benefit: the allocation of the new instance can now be done outside the lock in `push()` ❺, whereas in listing 6.2 it had to be done while holding the lock in `pop()`. Because memory allocation is typically quite an expensive operation, this can be very beneficial for the performance of the queue, because it reduces the time the mutex is held, allowing other threads to perform operations on the queue in the meantime.

Just like in the stack example, the use of a mutex to protect the entire data structure limits the concurrency supported by this queue; although multiple threads might be blocked on the queue in various member functions, only one thread can be doing any work at a time. However, part of this restriction comes from the use of `std::queue<>` in the implementation; by using the standard container you now have essentially one data item that's either protected or not. By taking control of the detailed implementation of the data structure, you can provide more fine-grained locking and thus allow a higher level of concurrency.

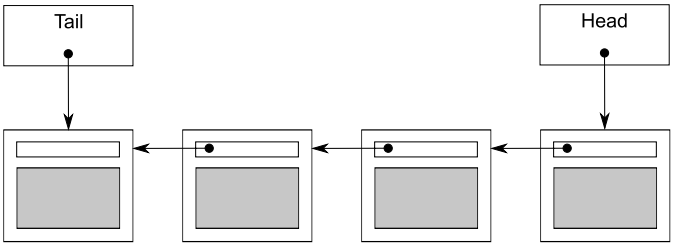

Figure 6.1 A queue represented using a single-linked list

6.2.3 A thread-safe queue using fine-grained locks and condition variables

In listings 6.2 and 6.3 you have one protected data item (`data_queue`) and thus one mutex. In order to use finer-grained locking, you need to look inside the queue at its constituent parts and associate one mutex with each distinct data item.

The simplest data structure for a queue is a singly linked list, as shown in figure 6.1. The queue contains a *head* pointer, which points to the first item in the list, and each item then points to the next item. Data items are removed from the queue by replacing the head pointer with the pointer to the next item and then returning the data from the old head.

Items are added to the queue at the other end. In order to do this, the queue also contains a *tail* pointer, which refers to the last item in the list. New nodes are added by changing the *next* pointer of the last item to point to the new node and then updating the tail pointer to refer to the new item. When the list is empty, both the head and tail pointers are `NULL`.

The following listing shows a simple implementation of such a queue based on a cut-down version of the interface to the queue in listing 6.2; you have only one `try_pop()` function and no `wait_and_pop()` because this queue supports only single-threaded use.

Listing 6.4 A simple single-threaded queue implementation

```
template<typename T>
class queue
{
private:
    struct node
    {
        T data;
        std::unique_ptr<node> next;

        node(T data_):
            data(std::move(data_))
        {}
    };

    std::unique_ptr<node> head;    <-- 1
    node* tail;                    <-- 2

public:
    queue()
    {}
```

```
    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;

    std::shared_ptr<T> try_pop()
    {
        if(!head)
        {
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> const res(
            std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head=std::move(head);
        head=std::move(old_head->next);
        return res;
    }

    void push(T new_value)
    {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail=p.get();
        if(tail)
        {
            tail->next=std::move(p);
        }
        else
        {
            head=std::move(p);
        }
        tail=new_tail;
    }
};
```


First off, note that listing 6.4 uses `std::unique_ptr<node>` to manage the nodes, because this ensures that they (and the data they refer to) get deleted when they're no longer needed, without having to write an explicit `delete`. This ownership chain is managed from `head`, with `tail` being a raw pointer to the last node.

Although this implementation works fine in a single-threaded context, a couple of things will cause you problems if you try to use fine-grained locking in a multi-threaded context. Given that you have two data items (`head` ❶ and `tail` ❷), you could in principle use two mutexes, one to protect `head` and one to protect `tail`, but there are a couple of problems with that.

The most obvious problem is that `push()` can modify both `head` ❺ and `tail` ❻, so it would have to lock both mutexes. This isn't too much of a problem, although it's unfortunate, because locking both mutexes would be possible. The critical problem is that both `push()` and `pop()` access the `next` pointer of a node: `push()` updates `tail->next` ❹, and `try_pop()` reads `head->next` ❸. If there's a single item in the queue, then `head==tail`, so both `head->next` and `tail->next` are the same object, which therefore requires protection. Because you can't tell if it's the same object without reading both `head` and `tail`, you now have to lock the same mutex in both `push()` and `try_pop()`, so you're no better off than before. Is there a way out of this dilemma?

ENABLING CONCURRENCY BY SEPARATING DATA

You can solve this problem by preallocating a dummy node with no data to ensure that there's always at least one node in the queue to separate the node being accessed at the head from that being accessed at the tail. For an empty queue, `head` and `tail` now both point to the dummy node rather than being `NULL`. This is fine, because `try_pop()` doesn't access `head->next` if the queue is empty. If you add a node to the queue (so there's one real node), then `head` and `tail` now point to separate nodes, so there's no race on `head->next` and `tail->next`. The downside is that you have to add an extra level of indirection to store the data by pointer in order to allow the dummy nodes. The following listing shows how the implementation looks now.

Listing 6.5 A simple queue with a dummy node

```
template<typename T>
class queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;     <-- 1
        std::unique_ptr<node> next;
    };

    std::unique_ptr<node> head;
    node* tail;

public:
    queue():
        head(new node),tail(head.get())    <-- 2
    {}

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

    std::shared_ptr<T> try_pop()
    {
        if(head.get()==tail)       <-- 3
        {
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> const res(head->data);        <-- 4
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);                  <-- 5
        return res;                    <-- 6
    }

    void push(T new_value)
    {
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));      <-- 7
        std::unique_ptr<node> p(new node);              <-- 8
        tail->data=new_data;                <-- 9
        node* const new_tail=p.get();
        tail->next=std::move(p);
```

```
        tail=new_tail;
    }
};
```

The changes to `try_pop()` are fairly minimal. First, you're comparing `head` against `tail` ❸ rather than checking for `NULL`, because the dummy node means that `head` is never `NULL`. Because `head` is a `std::unique_ptr<node>`, you need to call `head.get()` to do the comparison. Second, because the `node` now stores the data by pointer ❶, you can retrieve the pointer directly ❹ rather than having to construct a new instance of `T`. The big changes are in `push()`: you must first create a new instance of `T` on the heap and take ownership of it in a `std::shared_ptr<>` ❼ (note the use of `std::make_shared` to avoid the overhead of a second memory allocation for the reference count). The new node you create is going to be the new dummy node, so you don't need to supply the `new_value` to the constructor ❽. Instead, you set the data on the old dummy node to your newly allocated copy of the `new_value` ❾. Finally, in order to have a dummy node, you have to create it in the constructor ❷.

By now, I'm sure you're wondering what these changes buy you and how they help with making the queue thread-safe. Well, `push()` now accesses only `tail`, not `head`, which is an improvement. `try_pop()` accesses both `head` and `tail`, but `tail` is needed only for the initial comparison, so the lock is short-lived. The big gain is that the dummy node means `try_pop()` and `push()` are never operating on the same node, so you no longer need an overarching mutex. So, you can have one mutex for `head` and one for `tail`. Where do you put the locks?

You're aiming for the maximum opportunities for concurrency, so you want to hold the locks for the smallest possible length of time. `push()` is easy: the mutex needs to be locked across all accesses to `tail`, which means you lock the mutex after the new node is allocated ❽ and before you assign the data to the current tail node ❾. The lock then needs to be held until the end of the function.

`try_pop()` isn't so easy. First off, you need to lock the mutex on `head` and hold it until you're finished with `head`. In essence, this is the mutex to determine which thread does the popping, so you want to do that first. Once `head` is changed ❺, you can unlock the mutex; it doesn't need to be locked when you return the result ❻. That leaves the access to `tail` needing a lock on the tail mutex. Because you need to access `tail` only once, you can just acquire the mutex for the time it takes to do the read. This is best done by wrapping it in a function. In fact, because the code that needs the `head` mutex locked is only a subset of the member, it's clearer to wrap that in a function too. The final code is shown here.

Listing 6.6 A thread-safe queue with fine-grained locking

```
template<typename T>
class threadsafe_queue
{
private:
    struct node
```

```
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail()
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if(head.get()==get_tail())
        {
            return nullptr;
        }
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue():
        head(new node),tail(head.get())
    {}

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

    std::shared_ptr<T> try_pop()
    {
        std::unique_ptr<node> old_head=pop_head();
        return old_head?old_head->data:std::shared_ptr<T>();
    }

    void push(T new_value)
    {
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail=p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data=new_data;
        tail->next=std::move(p);
        tail=new_tail;
    }
};
```

Let's look at this code with a critical eye, thinking about the guidelines listed in section 6.1.1. Before you look for broken invariants, you should be sure what they are:

- `tail->next==nullptr`.
- `tail->data==nullptr`.
- `head==tail` implies an empty list.
- A single element list has `head->next==tail`.
- For each node `x` in the list, where `x!=tail`, `x->data` points to an instance of `T` and `x->next` points to the next node in the list. `x->next==tail` implies `x` is the last node in the list.
- Following the `next` nodes from `head` will eventually yield `tail`.

On its own, `push()` is straightforward: the only modifications to the data structure are protected by `tail_mutex`, and they uphold the invariant because the new tail node is an empty node and `data` and `next` are correctly set for the old tail node, which is now the last real node in the list.

The interesting part is `try_pop()`. It turns out that not only is the lock on `tail_mutex` necessary to protect the read of `tail` itself, but it's also necessary to ensure that you don't get a data race reading the data from the head. If you didn't have that mutex, it would be quite possible for a thread to call `try_pop()` and a thread to call `push()` concurrently, and there'd be no defined ordering on their operations. Even though each member function holds a lock on a mutex, they hold locks on *different* mutexes, and they potentially access the same data; all data in the queue originates from a call to `push()`, after all. Because the threads would be potentially accessing the same data without a defined ordering, this would be a data race, as you saw in chapter 5, and undefined behavior. Thankfully the lock on the `tail_mutex` in `get_tail()` solves everything. Because the call to `get_tail()` locks the same mutex as the call to `push()`, there's a defined order between the two calls. Either the call to `get_tail()` occurs before the call to `push()`, in which case it sees the old value of `tail`, or it occurs after the call to `push()`, in which case it sees the new value of `tail` *and the new data attached to the previous value of* `tail`.

It's also important that the call to `get_tail()` occurs inside the lock on `head_mutex`. If it didn't, the call to `pop_head()` could be stuck in between the call to `get_tail()` and the lock on the `head_mutex`, because other threads called `try_pop()` (and thus `pop_head()`) and acquired the lock first, thus preventing your initial thread from making progress:

```
std::unique_ptr<node> pop_head()
{
    node* const old_tail=get_tail();
    std::lock_guard<std::mutex> head_lock(head_mutex);

    if(head.get()==old_tail)
    {
        return nullptr;
    }
    std::unique_ptr<node> old_head=std::move(head);
    head=std::move(old_head->next);
    return old_head;
}
```

This is a broken implementation

1 Get old tail value outside lock on head_mutex

2

3

In this *broken* scenario, where the call to `get_tail(0)` ❶ is made outside the scope of the lock, you might find that both `head` and `tail` have changed by the time your initial thread can acquire the lock on `head_mutex`, and not only is the returned tail node no longer the tail, but it's no longer even part of the list. This could then mean that the comparison of `head` to `old_tail` ❷ fails, even if `head` really is the last node. Consequently, when you update `head` ❸ you may end up moving `head` beyond `tail` and off the end of the list, destroying the data structure. In the `correct` implementation from listing 6.6, you keep the call to `get_tail()` inside the lock on `head_mutex`. This ensures that no other threads can change `head`, and `tail` only ever moves further away (as new nodes are added in calls to `push()`), which is perfectly safe. `head` can never pass the value returned from `get_tail()`, so the invariants are upheld.

Once `pop_head()` has removed the node from the queue by updating `head`, the mutex is unlocked, and `try_pop()` can extract the data and delete the node if there was one (and return a `NULL` instance of `std::shared_ptr<>` if not), safe in the knowledge that it's the only thread that can access this node.

Next up, the external interface is a subset of that from listing 6.2, so the same analysis applies: there are no race conditions inherent in the interface.

Exceptions are more interesting. Because you've changed the data allocation patterns, the exceptions can now come from different places. The only operations in `try_pop()` that can throw exceptions are the mutex locks, and the data isn't modified until the locks are acquired. Therefore `try_pop()` is exception-safe. On the other hand, `push()` allocates a new instance of `T` on the heap and a new instance of `node`, either of which might throw an exception. However, both of the newly allocated objects are assigned to smart pointers, so they'll be freed if an exception is thrown. Once the lock is acquired, none of the remaining operations in `push()` can throw an exception, so again you're home and dry and `push()` is exception-safe too.

Because you haven't changed the interface, there are no new external opportunities for deadlock. There are no internal opportunities either; the only place that two locks are acquired is in `pop_head()`, which always acquires the `head_mutex` and then the `tail_mutex`, so this will never deadlock.

The remaining question concerns the actual possibilities for concurrency. This data structure actually has considerably more scope for concurrency than that from listing 6.2, because the locks are more fine-grained and *more is done outside the locks*. For example, in `push()`, the new node and new data item are allocated with no locks held. This means that multiple threads can be allocating new nodes and data items concurrently without a problem. Only one thread can add its new node to the list at a time, but the code to do so is only a few simple pointer assignments, so the lock isn't held for much time at all compared to the `std::queue<>`-based implementation where the lock is held around all the memory allocation operations internal to the `std::queue<>`.

Also, `try_pop()` holds the `tail_mutex` for only a short time, to protect a read from `tail`. Consequently, almost the entirety of a call to `try_pop()` can occur concurrently

with a call to `push()`. Also, the operations performed while holding the `head_mutex` are also quite minimal; the expensive `delete` (in the destructor of the node pointer) is outside the lock. This will increase the number of calls to `try_pop()` that can happen concurrently; only one thread can call `pop_head()` at a time, but multiple threads can then delete their old nodes and return the data safely.

WAITING FOR AN ITEM TO POP

OK, so listing 6.6 provides a thread-safe queue with fine-grained locking, but it supports only `try_pop()` (and only one overload at that). What about the handy `wait_and_pop()` functions back in listing 6.2? Can you implement an identical interface with your fine-grained locking?

Of course, the answer is, yes, but the real question is, how? Modifying `push()` is easy: just add the `data_cond.notify_one()` call at the end of the function, just like in listing 6.2. Actually, it's not quite that simple; you're using fine-grained locking because you want the maximum possible amount of concurrency. If you leave the mutex locked across the call to `notify_one()` (as in listing 6.2), then if the notified thread wakes up before the mutex has been unlocked, it will have to wait for the mutex. On the other hand, if you unlock the mutex *before* you call `notify_one()`, then the mutex is available for the waiting thread to acquire when it wakes up (assuming no other thread locks it first). This is a minor improvement, but it might be important in some cases.

`wait_and_pop()` is more complicated, because you have to decide where to wait, what the predicate is, and which mutex needs to be locked. The condition you're waiting for is "queue not empty," which is represented by `head!=tail`. Written like that, it would require both `head_mutex` and `tail_mutex` to be locked, but you've already decided in listing 6.6 that you only need to lock `tail_mutex` for the read of `tail` and not for the comparison itself, so you can apply the same logic here. If you make the predicate `head!=get_tail()`,you only need to hold the `head_mutex`, so you can use your lock on that for the call to `data_cond.wait()`. Once you've added the wait logic, the implementation is the same as `try_pop()`.

The second overload of `try_pop()` and the corresponding `wait_and_pop()` overload require careful thought. If you just replace the return of the `std::shared_ptr<>` retrieved from `old_head` with a copy assignment to the `value` parameter, there's a potential exception-safety issue. At this point, the data item has been removed from the queue and the mutex unlocked; all that remains is to return the data to the caller. However, if the copy assignment throws an exception (as it very well might), the data item is lost because it can't be returned to the queue in the same place.

If the actual type `T` used for the template argument has a no-throw move-assignment operator or a no-throw swap operation, you could use that, but you'd really like a general solution that could be used for any type `T`. In this case, you have to move the potential throwing inside the locked region, before the node is removed from the list. This means you need an extra overload of `pop_head()` that retrieves the stored value prior to modifying the list.

In comparison, `empty()` is trivial: just lock `head_mutex` and check for `head==get_tail()` (see listing 6.10). The final code for the queue is shown in listings 6.7, 6.8, 6.9, and 6.10.

Listing 6.7 A thread-safe queue with locking and waiting: internals and interface

```
template<typename T>
class threadsafe_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;
    std::condition_variable data_cond;
public:
    threadsafe_queue():
        head(new node),tail(head.get())
    {}
    threadsafe_queue(const threadsafe_queue& other)=delete;
    threadsafe_queue& operator=(const threadsafe_queue& other)=delete;

    std::shared_ptr<T> try_pop();
    bool try_pop(T& value);
    std::shared_ptr<T> wait_and_pop();
    void wait_and_pop(T& value);
    void push(T new_value);
    void empty();
};
```

Pushing new nodes onto the queue is fairly straightforward—the implementation (shown in the following listing) is close to that shown previously.

Listing 6.8 A thread-safe queue with locking and waiting: pushing new values

```
template<typename T>
void threadsafe_queue<T>::push(T new_value)
{
    std::shared_ptr<T> new_data(
        std::make_shared<T>(std::move(new_value)));
    std::unique_ptr<node> p(new node);
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data=new_data;
        node* const new_tail=p.get();
        tail->next=std::move(p);
        tail=new_tail;
    }
    data_cond.notify_one();
}
```

As already mentioned, the complexity is all in the *pop* side, which makes use of a series of helper functions to simplify matters. The next listing shows the implementation of `wait_and_pop()` and the associated helper functions.

Listing 6.9 A thread-safe queue with locking and waiting: `wait_and_pop()`

```
template<typename T>
class threadsafe_queue
{
private:
    node* get_tail()
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head()     <-(1)
    {
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);
        return old_head;
    }

    std::unique_lock<std::mutex> wait_for_data()     <-(2)
    {
        std::unique_lock<std::mutex> head_lock(head_mutex);
        data_cond.wait(head_lock,[&]{return head.get()!=get_tail();});
        return std::move(head_lock);          <-(3)
    }

    std::unique_ptr<node> wait_pop_head()
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data());     <-(4)
        return pop_head();
    }

    std::unique_ptr<node> wait_pop_head(T& value)
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data());     <-(5)
        value=std::move(*head->data);
        return pop_head();
    }
public:
    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_ptr<node> const old_head=wait_pop_head();
        return old_head->data;
    }

    void wait_and_pop(T& value)
    {
        std::unique_ptr<node> const old_head=wait_pop_head(value);
    }
};
```

The implementation of the pop side shown in listing 6.9 has several little helper functions to simplify the code and reduce duplication, such as `pop_head()` ❶ and `wait_for_data()` ❷, which modify the list to remove the head item and wait for the queue to have some data to pop, respectively. `wait_for_data()` is particularly noteworthy, because not only does it wait on the condition variable using a lambda function for the predicate, but it also returns the lock instance to the caller ❸. This is to ensure that the same lock is held while the data is modified by the relevant `wait_pop_head()` overload ❹, ❺. `pop_head()` is also reused by the `try_pop()` code shown in the next listing.

Listing 6.10 A thread-safe queue with locking and waiting: `try_pop()` and `empty()`

```
template<typename T>
class threadsafe_queue
{
private:
    std::unique_ptr<node> try_pop_head()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        if(head.get()==get_tail())
        {
            return std::unique_ptr<node>();
        }
        return pop_head();
    }

    std::unique_ptr<node> try_pop_head(T& value)
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        if(head.get()==get_tail())
        {
            return std::unique_ptr<node>();
        }
        value=std::move(*head->data);
        return pop_head();
    }

public:
    std::shared_ptr<T> try_pop()
    {
        std::unique_ptr<node> old_head=try_pop_head();
        return old_head?old_head->data:std::shared_ptr<T>();
    }

    bool try_pop(T& value)
    {
        std::unique_ptr<node> const old_head=try_pop_head(value);
        return old_head;
    }

    void empty()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
```

```
        return (head.get()==get_tail());
    }
};
```

This queue implementation will serve as the basis for the lock-free queue covered in chapter 7. It's an *unbounded* queue; threads can continue to push new values onto the queue as long as there's available memory, even if no values are removed. The alternative to an unbounded queue is a *bounded* queue, in which the maximum length of the queue is fixed when the queue is created. Once a bounded queue is full, attempts to push further elements onto the queue will either fail or block, until an element has been popped from the queue to make room. Bounded queues can be useful for ensuring an even spread of work when dividing work between threads based on tasks to be performed (see chapter 8). This prevents the thread(s) populating the queue from running too far ahead of the thread(s) reading items from the queue.

The unbounded queue implementation shown here can easily be extended to limit the length of the queue by waiting on the condition variable in `push()`. Rather than waiting for the queue to have items (as is done in `pop()`), you need to wait for the queue to have fewer than the maximum number of items. Further discussion of bounded queues is outside the scope of this book; for now let's move beyond queues and on to more complex data structures.

6.3 Designing more complex lock-based data structures

Stacks and queues are simple: the interface is exceedingly limited, and they're very tightly focused on a specific purpose. Not all data structures are that simple; most data structures support a variety of operations. In principle, this can then lead to greater opportunities for concurrency, but it also makes the task of protecting the data that much harder because the multiple access patterns need to be taken into account. The precise nature of the various operations that can be performed is important when designing such data structures for concurrent access.

To see some of the issues involved, let's look at the design of a lookup table.

6.3.1 Writing a thread-safe lookup table using locks

A lookup table or dictionary associates values of one type (the key type) with values of either the same or a different type (the mapped type). In general, the intention behind such a structure is to allow code to query the data associated with a given key. In the C++ Standard Library, this facility is provided by the associative containers: `std::map<>`, `std::multimap<>`, `std::unordered_map<>`, and `std::unordered_multimap<>`.

A lookup table has a different usage pattern than a stack or a queue. Whereas almost every operation on a stack or a queue modifies it in some way, either to add an element or remove one, a lookup table might be modified rarely. The simple DNS cache in listing 3.13 is one example of such a scenario, which features a greatly reduced interface compared to `std::map<>`. As you saw with the stack and queue, the

interfaces of the standard containers aren't suitable when the data structure is to be accessed from multiple threads concurrently, because there are inherent race conditions in the interface design, so they need to be cut down and revised.

The biggest problem with the `std::map<>` interface from a concurrency perspective is the iterators. Although it's possible to have an iterator that provides safe access into a container even when other threads can access (and modify) the container, this is a tricky proposition. Correctly handling iterators requires you to deal with issues such as another thread deleting the element that the iterator is referring to, which can get rather involved. For the first cut at a thread-safe lookup table interface, you'll skip the iterators. Given that the interface to `std::map<>` (and the other associative containers in the standard library) is so heavily iterator-based, it's probably worth setting them aside and designing the interface from the ground up.

There are only a few basic operations on a lookup table:

- Add a new key/value pair.
- Change the value associated with a given key.
- Remove a key and its associated value.
- Obtain the value associated with a given key if any.

There are also a few container-wide operations that might be useful, such as a check on whether the container is empty, a snapshot of the complete list of keys, or a snapshot of the complete set of key/value pairs.

If you stick to the simple thread-safety guidelines such as not returning references and put a simple mutex lock around the entirety of each member function, all of these are safe; they either come before some modification from another thread or come after it. The biggest potential for a race condition is when a new key/value pair is being added; if two threads add a new value, only one will be first, and the second will therefore fail. One possibility is to combine add and change into a single member function, as you did for the DNS cache in listing 3.13.

The only other interesting point from an interface perspective is the *if any* part of obtaining an associated value. One option is to allow the user to provide a "default" result that's returned in the case when the key isn't present:

```
mapped_type get_value(key_type const& key, mapped_type default_value);
```

In this case, a default-constructed instance of `mapped_type` could be used if the `default_value` wasn't explicitly provided. This could also be extended to return a `std::pair<mapped_type,bool>` instead of just an instance of `mapped_type`, where the `bool` indicates whether the value was present. Another option is to return a smart pointer referring to the value; if the pointer value is `NULL`, there was no value to return.

As already mentioned, once the interface has been decided, then (assuming no interface race conditions) the thread safety could be guaranteed by using a single mutex and a simple lock around every member function to protect the underlying data structure. However, this would squander the possibilities for concurrency provided by the separate functions for reading the data structure and modifying it. One

option is to use a mutex that supports multiple reader threads or a single writer thread, such as the `boost::shared_mutex` used in listing 3.13. Although this would indeed improve the possibilities for concurrent access, only one thread could modify the data structure at a time. Ideally, you'd like to do better than that.

DESIGNING A MAP DATA STRUCTURE FOR FINE-GRAINED LOCKING

As with the queue discussed in section 6.2.3, in order to permit fine-grained locking you need to look carefully at the details of the data structure rather than just wrapping a preexisting container such as `std::map<>`. There are three common ways of implementing an associative container like your lookup table:

- A binary tree, such as a red-black tree
- A sorted array
- A hash table

A binary tree doesn't provide much scope for extending the opportunities for concurrency; every lookup or modification has to start by accessing the root node, which therefore has to be locked. Although this lock can be released as the accessing thread moves down the tree, this isn't much better than a single lock across the whole data structure.

A sorted array is even worse, because you can't tell in advance where in the array a given data value is going to be, so you need a single lock for the whole array.

That leaves the hash table. Assuming a fixed number of buckets, which bucket a key belongs to is purely a property of the key and its hash function. This means you can safely have a separate lock per bucket. If you again use a mutex that supports multiple readers or a single writer, you increase the opportunities for concurrency *N*-fold, where *N* is the number of buckets. The downside is that you need a good hash function for the key. The C++ Standard Library provides the `std::hash<>` template, which you can use for this purpose. It's already specialized for the fundamental types such as `int` and common library types such as `std::string`, and the user can easily specialize it for other key types. If you follow the lead of the standard unordered containers and take the type of the function object to use for doing the hashing as a template parameter, the user can choose whether to specialize `std::hash<>` for their key type or provide a separate hash function.

So, let's look at some code. What might the implementation of a thread-safe lookup table look like? One possibility is shown here.

Listing 6.11 A thread-safe lookup table

```
template<typename Key,typename Value,typename Hash=std::hash<Key> >
class threadsafe_lookup_table
{
private:
    class bucket_type
    {
    private:
        typedef std::pair<Key,Value> bucket_value;
```

```
        typedef std::list<bucket_value> bucket_data;
        typedef typename bucket_data::iterator bucket_iterator;

        bucket_data data;
        mutable boost::shared_mutex mutex;     <-(1)

        bucket_iterator find_entry_for(Key const& key) const    <-(2)
        {
            return std::find_if(data.begin(),data.end(),
                                [&](bucket_value const& item)
                                {return item.first==key;});
        }

    public:
        Value value_for(Key const& key,Value const& default_value) const
        {
            boost::shared_lock<boost::shared_mutex> lock(mutex);     <-(3)
            bucket_iterator const found_entry=find_entry_for(key);
            return (found_entry==data.end())?
                default_value:found_entry->second;
        }

        void add_or_update_mapping(Key const& key,Value const& value)
        {
            std::unique_lock<boost::shared_mutex> lock(mutex);       <-(4)
            bucket_iterator const found_entry=find_entry_for(key);
            if(found_entry==data.end())
            {
                data.push_back(bucket_value(key,value));
            }
            else
            {
                found_entry->second=value;
            }
        }

        void remove_mapping(Key const& key)
        {
            std::unique_lock<boost::shared_mutex> lock(mutex);       <-(5)
            bucket_iterator const found_entry=find_entry_for(key);
            if(found_entry!=data.end())
            {
                data.erase(found_entry);
            }
        }
    };

    std::vector<std::unique_ptr<bucket_type> > buckets;     <-(6)
    Hash hasher;

    bucket_type& get_bucket(Key const& key) const     <-(7)
    {
        std::size_t const bucket_index=hasher(key)%buckets.size();
        return *buckets[bucket_index];
    }

public:
    typedef Key key_type;
```

```
    typedef Value mapped_type;
    typedef Hash hash_type;

    threadsafe_lookup_table(
        unsigned num_buckets=19,Hash const& hasher_=Hash()):
        buckets(num_buckets),hasher(hasher_)
    {
        for(unsigned i=0;i<num_buckets;++i)
        {
            buckets[i].reset(new bucket_type);
        }
    }

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

    Value value_for(Key const& key,
                    Value const& default_value=Value()) const
    {
        return get_bucket(key).value_for(key,default_value);     <-- 8
    }

    void add_or_update_mapping(Key const& key,Value const& value)
    {
        get_bucket(key).add_or_update_mapping(key,value);     <-- 9
    }

    void remove_mapping(Key const& key)
    {
        get_bucket(key).remove_mapping(key);     <-- 10
    }
};
```

This implementation uses a `std::vector<std::unique_ptr<bucket_type>>` ❻ to hold the buckets, which allows the number of buckets to be specified in the constructor. The default is 19, which is an arbitrary prime number; hash tables work best with a prime number of buckets. Each bucket is protected with an instance of `boost::shared_mutex` ❶ to allow many concurrent reads or a single call to either of the modification functions *per bucket*.

Because the number of buckets is fixed, the `get_bucket()` function ❼ can be called without any locking ❽, ❾, ❿, and then the bucket mutex can be locked either for shared (read-only) ownership ❸ or unique (read/write) ownership ❹, ❺ as appropriate for each function.

All three functions make use of the `find_entry_for()` member function ❷ on the bucket to determine whether the entry is in the bucket. Each bucket contains just a `std::list<>` of key/value pairs, so adding and removing entries is easy.

I've already covered the concurrency angle, and everything is suitably protected with mutex locks, so what about exception safety? `value_for` doesn't modify anything, so that's fine; if it throws an exception, it won't affect the data structure. `remove_mapping` modifies the list with the call to `erase`, but this is guaranteed not to throw, so that's safe. This leaves `add_or_update_mapping`, which might throw in either

of the two branches of the `if`. `push_back` is exception-safe and will leave the list in the original state if it throws, so that branch is fine. The only problem is with the assignment in the case where you're replacing an existing value; if the assignment throws, you're relying on it leaving the original unchanged. However, this doesn't affect the data structure as a whole and is entirely a property of the user-supplied type, so you can safely leave it up to the user to handle this.

At the beginning of this section, I mentioned that one nice-to-have feature of such a lookup table would be the option of retrieving a snapshot of the current state into, for example, a `std::map<>`. This would require locking the entire container in order to ensure that a consistent copy of the state is retrieved, which requires locking all the buckets. Because the "normal" operations on the lookup table require a lock on only one bucket at a time, this would be the only operation that requires a lock on all the buckets. Therefore, provided you lock them in the same order every time (for example, increasing bucket index), there'll be no opportunity for deadlock. Such an implementation is shown in the following listing.

Listing 6.12 Obtaining contents of a `threadsafe_lookup_table` as a `std::map<>`

```
std::map<Key,Value> threadsafe_lookup_table::get_map() const
{
    std::vector<std::unique_lock<boost::shared_mutex> > locks;
    for(unsigned i=0;i<buckets.size();++i)
    {
        locks.push_back(
            std::unique_lock<boost::shared_mutex>(buckets[i].mutex));
    }
    std::map<Key,Value> res;
    for(unsigned i=0;i<buckets.size();++i)
    {
        for(bucket_iterator it=buckets[i].data.begin();
            it!=buckets[i].data.end();
            ++it)
        {
            res.insert(*it);
        }
    }
    return res;
}
```

The lookup table implementation from listing 6.11 increases the opportunity for concurrency of the lookup table as a whole by locking each bucket separately and by using a `boost::shared_mutex` to allow reader concurrency on each bucket. But what if you could increase the potential for concurrency on a bucket by even finer-grained locking? In the next section, you'll do just that by using a thread-safe list container with iterator support.

6.3.2 Writing a thread-safe list using locks

A list is one of the most basic data structures, so it should be straightforward to write a thread-safe one, shouldn't it? Well, that depends on what facilities you're after, and you need one that offers iterator support, something I shied away from adding to your map on the basis that it was too complicated. The basic issue with STL-style iterator support is that the iterator must hold some kind of reference into the internal data structure of the container. If the container can be modified from another thread, this reference must somehow remain valid, which essentially requires that the iterator hold a lock on some part of the structure. Given that the lifetime of an STL-style iterator is completely outside the control of the container, this is a bad idea.

The alternative is to provide iteration functions such as `for_each` as part of the container itself. This puts the container squarely in charge of the iteration and locking, but it does fall foul of the deadlock avoidance guidelines from chapter 3. In order for `for_each` to do anything useful, it must call user-supplied code while holding the internal lock. Not only that, but it must also pass a reference to each item to this user-supplied code in order for the user-supplied code to work on this item. You could avoid this by passing a copy of each item to the user-supplied code, but that would be expensive if the data items were large.

So, for now you'll leave it up to the user to ensure that they don't cause deadlock by acquiring locks in the user-supplied operations and don't cause data races by storing the references for access outside the locks. In the case of the list being used by the lookup table, this is perfectly safe, because you know you're not going to do anything naughty.

That leaves you with the question of which operations to supply for your list. If you cast your eyes back on listings 6.11 and 6.12, you can see the sorts of operations you require:

- Add an item to the list.
- Remove an item from the list if it meets a certain condition.
- Find an item in the list that meets a certain condition.
- Update an item that meets a certain condition.
- Copy each item in the list to another container.

For this to be a good general-purpose list container, it would be helpful to add further operations such as a positional insert, but this is unnecessary for your lookup table, so I'll leave it as an exercise for the reader.

The basic idea with fine-grained locking for a linked list is to have one mutex per node. If the list gets big, that's a lot of mutexes! The benefit here is that operations on separate parts of the list are truly concurrent: each operation holds only the locks on the nodes it's actually interested in and unlocks each node as it moves on to the next. The next listing shows an implementation of just such a list.

Listing 6.13 A thread-safe list with iteration support

```
template<typename T>
class threadsafe_list
{
    struct node                          <-- 1
    {
        std::mutex m;
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;

        node():                          <-- 2
            next()
        {}

        node(T const& value):            <-- 3
            data(std::make_shared<T>(value))
        {}
    };

    node head;

public:
    threadsafe_list()
    {}

    ~threadsafe_list()
    {
        remove_if([](node const&){return true;});
    }

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

    void push_front(T const& value)
    {
        std::unique_ptr<node> new_node(new node(value));    <-- 4
        std::lock_guard<std::mutex> lk(head.m);
        new_node->next=std::move(head.next);                <-- 5
        head.next=std::move(new_node);                      <-- 6
    }

    template<typename Function>
    void for_each(Function f)                               <-- 7
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);            <-- 8
        while(node* const next=current->next.get())         <-- 9
        {
            std::unique_lock<std::mutex> next_lk(next->m);  <-- 10
            lk.unlock();                                    <-- 11
            f(*next->data);                                 <-- 12
            current=next;
            lk=std::move(next_lk);                          <-- 13
        }
    }

    template<typename Predicate>
    std::shared_ptr<T> find_first_if(Predicate p)           <-- 14
```

```
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);
        while(node* const next=current->next.get())
        {
            std::unique_lock<std::mutex> next_lk(next->m);
            lk.unlock();
            if(p(*next->data))     <-(15)
            {
                return next->data;     <-(16)
            }
            current=next;
            lk=std::move(next_lk);
        }
        return std::shared_ptr<T>();
    }

    template<typename Predicate>
    void remove_if(Predicate p)       <-(17)
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);
        while(node* const next=current->next.get())
        {
            std::unique_lock<std::mutex> next_lk(next->m);
            if(p(*next->data))                                  <-(18)
            {
                std::unique_ptr<node> old_next=std::move(current->next);
                current->next=std::move(next->next);                    <-(19)
                next_lk.unlock();
            }                   <-(20)
            else
            {
                lk.unlock();      <-(21)
                current=next;
                lk=std::move(next_lk);
            }
        }
    }
};
```

The `threadsafe_list<>` from listing 6.13 is a singly linked list, where each entry is a `node` structure ❶. A default-constructed `node` is used for the `head` of the list, which starts with a `NULL` `next` pointer ❷. New nodes are added with the `push_front()` function; first a new node is constructed ❹, which allocates the stored data on the heap ❸, while leaving the `next` pointer as `NULL`. You then need to acquire the lock on the mutex for the `head` node in order to get the appropriate `next` value ❺ and insert the node at the front of the list by setting `head.next` to point to your new node ❻. So far, so good: you only need to lock one mutex in order to add a new item to the list, so there's no risk of deadlock. Also, the slow memory allocation happens outside the lock, so the lock is only protecting the update of a couple of pointer values that can't fail. On to the iterative functions.

First up, let's look at `for_each()` ❼. This operation takes a `Function` of some type to apply to each element in the list; in common with most standard library algorithms, it takes this function by value and will work with either a genuine function or an object of a type with a function call operator. In this case, the function must accept a value of type `T` as the sole parameter. Here's where you do the hand-over-hand locking. To start with, you lock the mutex on the `head` node ❽. It's then safe to obtain the pointer to the `next` node (using `get()` because you're not taking ownership of the pointer). If that pointer isn't `NULL` ❾, you lock the mutex on that node ❿ in order to process the data. Once you have the lock on that node, you can release the lock on the previous node ⓫ and call the specified function ⓬. Once the function completes, you can update the `current` pointer to the node you just processed and `move` the ownership of the lock from `next_lk` out to `lk` ⓭. Because `for_each` passes each data item directly to the supplied `Function`, you can use this to update the items if necessary or copy them into another container, or whatever. This is entirely safe if the function is well behaved, because the mutex for the node holding the data item is held across the call.

`find_first_if()` ⓮ is similar to `for_each()`; the crucial difference is that the supplied `Predicate` must return `true` to indicate a match or `false` to indicate no match ⓯. Once you have a match, you just return the found data ⓰ rather than continuing to search. You could do this with `for_each()`, but it would needlessly continue processing the rest of the list even once a match had been found.

`remove_if()` ⓱ is slightly different, because this function has to actually update the list; you can't use `for_each()` for this. If the `Predicate` returns `true` ⓲, you remove the node from the list by updating `current->next` ⓳. Once you've done that, you can release the lock held on the mutex for the `next` node. The node is deleted when the `std::unique_ptr<node>` you moved it into goes out of scope ⓴. In this case, you don't update `current` because you need to check the new `next` node. If the `Predicate` returns `false`, you just want to move on as before ㉑.

So, are there any deadlocks or race conditions with all these mutexes? The answer here is quite definitely *no*, provided that the supplied predicates and functions are well behaved. The iteration is always one way, always starting from the `head` node, and always locking the next mutex before releasing the current one, so there's no possibility of different lock orders in different threads. The only potential candidate for a race condition is the deletion of the removed node in `remove_if()` ⓴ because you do this after you've unlocked the mutex (it's undefined behavior to destroy a locked mutex). However, a few moments' thought reveals that this is indeed safe, because you still hold the mutex on the previous node (`current`), so no new thread can try to acquire the lock on the node you're deleting.

What about opportunities for concurrency? The whole point of such fine-grained locking was to improve the possibilities for concurrency over a single mutex, so have you achieved that? Yes, you have: different threads can be working on different nodes in the list at the same time, whether they're just processing each item with `for_each()`, searching with `find_first_if()`, or removing items with `remove_if()`. But because

the mutex for each node must be locked in turn, the threads can't pass each other. If one thread is spending a long time processing a particular node, other threads will have to wait when they reach that particular node.

6.4 Summary

This chapter started by looking at what it means to design a data structure for concurrency and providing some guidelines for doing so. We then worked through several common data structures (stack, queue, hash map, and linked list), looking at how to apply those guidelines to implement them in a way designed for concurrent access, using locks to protect the data and prevent data races. You should now be able to look at the design of your own data structures to see where the opportunities for concurrency lie and where there's potential for race conditions.

In chapter 7 we'll look at ways of avoiding locks entirely, using the low-level atomic operations to provide the necessary ordering constraints, while sticking to the same set of guidelines.