C++ Concurrency IN ACTION

Practical Multithreading

Anthony Williams

MANNING

Table of Contents

7 Designing lock-free concurrent data structures

This chapter covers

- Implementations of data structures designed for concurrency without using locks
- Techniques for managing memory in lock-free data structures
- Simple guidelines to aid in the writing of lock-free data structures

In the last chapter we looked at general aspects of designing data structures for concurrency, with guidelines for thinking about the design to ensure they're safe. We then examined several common data structures and looked at example implementations that used mutexes and locks to protect the shared data. The first couple of examples used one mutex to protect the entire data structure, but later ones used more than one to protect various smaller parts of the data structure and allow greater levels of concurrency in accesses to the data structure.

Mutexes are powerful mechanisms for ensuring that multiple threads can safely access a data structure without encountering race conditions or broken invariants. It's also relatively straightforward to reason about the behavior of code that uses them: either the code has the lock on the mutex protecting the data or it doesn't. However, it's not all a bed of roses; you saw in chapter 3 how the incorrect use of locks can lead

to deadlock, and you've just seen with the lock-based queue and lookup table examples how the granularity of locking can affect the potential for true concurrency. If you can write data structures that are safe for concurrent access without locks, there's the potential to avoid these problems. Such a data structure is called a *lock-free* data structure.

In this chapter we'll look at how the memory-ordering properties of the atomic operations introduced in chapter 5 can be used to build lock-free data structures. You need to take extreme care when designing such data structures, because they're hard to get right, and the conditions that cause the design to fail may occur very rarely. We'll start by looking at what it means for data structures to be lock-free; then we'll move on to the reasons for using them before working through some examples and drawing out some general guidelines.

7.1 Definitions and consequences

Algorithms and data structures that use mutexes, condition variables, and futures to synchronize the data are called *blocking* data structures and algorithms. The application calls library functions that will suspend the execution of a thread until another thread performs an action. Such library calls are termed *blocking* calls because the thread can't progress past this point until the block is removed. Typically, the OS will suspend a blocked thread completely (and allocate its time slices to another thread) until it's *unblocked* by the appropriate action of another thread, whether that's unlocking a mutex, notifying a condition variable, or making a future *ready*.

Data structures and algorithms that don't use blocking library functions are said to be *nonblocking*. Not all such data structures are *lock-free*, though, so let's look at the various types of nonblocking data structures.

7.1.1 Types of nonblocking data structures

Back in chapter 5, we implemented a basic mutex using `std::atomic_flag` as a spin lock. The code is reproduced in the following listing.

Listing 7.1 Implementation of a spin-lock mutex using `std::atomic_flag`

```
class spinlock_mutex
{
    std::atomic_flag flag;
public:
    spinlock_mutex():
        flag(ATOMIC_FLAG_INIT)
    {}
    void lock()
    {
        while(flag.test_and_set(std::memory_order_acquire));
    }
    void unlock()
    {
        flag.clear(std::memory_order_release);
    }
};
```

This code doesn't call any blocking functions; `lock()` just keeps looping until the call to `test_and_set()` returns `false`. This is why it gets the name *spin lock*—the code "spins" around the loop. Anyway, there are no blocking calls, so any code that uses this mutex to protect shared data is consequently *nonblocking*. It's not *lock-free,* though. It's still a mutex and can still be locked by only one thread at a time. Let's look at the definition of *lock-free* so you can see what kinds of data structures *are* covered.

7.1.2 Lock-free data structures

For a data structure to qualify as lock-free, more than one thread must be able to access the data structure concurrently. They don't have to be able to do the same operations; a lock-free queue might allow one thread to push and one to pop but break if two threads try to push new items at the same time. Not only that, but if one of the threads accessing the data structure is suspended by the scheduler midway through its operation, the other threads must still be able to complete their operations without waiting for the suspended thread.

Algorithms that use compare/exchange operations on the data structure often have loops in them. The reason for using a compare/exchange operation is that another thread might have modified the data in the meantime, in which case the code will need to redo part of its operation before trying the compare/exchange again. Such code can still be lock-free if the compare/exchange would eventually succeed if the other threads were suspended. If it wouldn't, you'd essentially have a spin lock, which is nonblocking but not lock-free.

Lock-free algorithms with such loops can result in one thread being subject to *starvation*. If another thread performs operations with the "wrong" timing, the other thread might make progress while the first thread continually has to retry its operation. Data structures that avoid this problem are wait-free as well as lock-free.

7.1.3 Wait-free data structures

A wait-free data structure is a lock-free data structure with the additional property that every thread accessing the data structure can complete its operation within a bounded number of steps, regardless of the behavior of other threads. Algorithms that can involve an unbounded number of retries because of clashes with other threads are thus not wait-free.

Writing wait-free data structures correctly is extremely hard. In order to ensure that every thread can complete its operations within a bounded number of steps, you have to ensure that each operation can be performed in a single pass and that the steps performed by one thread don't cause an operation on another thread to fail. This can make the overall algorithms for the various operations considerably more complex.

Given how hard it is to get a lock-free or wait-free data structure right, you need some pretty good reasons to write one; you need to be sure that the benefit outweighs the cost. Let's therefore examine the points that affect the balance.

7.1.4 The pros and cons of lock-free data structures

When it comes down to it, the primary reason for using lock-free data structures is to enable maximum concurrency. With lock-based containers, there's always the potential for one thread to have to block and wait for another to complete its operation before the first thread can proceed; preventing concurrency through mutual exclusion is the entire purpose of a mutex lock. With a lock-free data structure, *some* thread makes progress with every step. With a wait-free data structure, every thread can make forward progress, regardless of what the other threads are doing; there's no need for waiting. This is a desirable property to have but hard to achieve. It's all too easy to end up writing what's essentially a spin lock.

A second reason to use lock-free data structures is robustness. If a thread dies while holding a lock, that data structure is broken forever. But if a thread dies partway through an operation on a lock-free data structure, nothing is lost except that thread's data; other threads can proceed normally.

The flip side here is that if you can't exclude threads from accessing the data structure, then you must be careful to ensure that the invariants are upheld or choose alternative invariants that can be upheld. Also, you must pay attention to the ordering constraints you impose on the operations. To avoid the undefined behavior associated with a data race, you must use atomic operations for the modifications. But that alone isn't enough; you must ensure that changes become visible to other threads in the correct order. All this means that writing thread-safe data structures without using locks is considerably harder than writing them with locks.

Because there aren't any locks, deadlocks are impossible with lock-free data structures, although there is the possibility of live locks instead. A *live lock* occurs when two threads each try to change the data structure, but for each thread the changes made by the other require the operation to be restarted, so both threads loop and try again. Imagine two people trying to go through a narrow gap. If they both go at once, they get stuck, so they have to come out and try again. Unless someone gets there first (either by agreement, by being quicker, or by sheer luck), the cycle will repeat. As in this simple example, live locks are typically short lived because they depend on the exact scheduling of threads. They therefore sap performance rather than cause long-term problems, but they're still something to watch out for. By definition, wait-free code can't suffer from live lock because there's always an upper limit on the number of steps needed to perform an operation. The flip side here is that the algorithm is likely more complex than the alternative and may require more steps even when no other thread is accessing the data structure.

This brings us to another downside of lock-free and wait-free code: although it can increase the potential for concurrency of operations on a data structure and reduce the time an individual thread spends waiting, it may well *decrease* overall performance. First, the atomic operations used for lock-free code can be much slower than nonatomic operations, and there'll likely be more of them in a lock-free data structure than in the mutex locking code for a lock-based data structure. Not only that, but the

hardware must synchronize data between threads that access the same atomic variables. As you'll see in chapter 8, the cache ping-pong associated with multiple threads accessing the same atomic variables can be a significant performance drain. As with everything, it's important to check the relevant performance aspects (whether that's worst-case wait time, average wait time, overall execution time, or something else) both with a lock-based data structure and a lock-free one before committing either way.

Now let's look at some examples.

7.2 Examples of lock-free data structures

In order to demonstrate some of the techniques used in designing lock-free data structures, we'll look at the lock-free implementation of a series of simple data structures. Not only will each example describe the implementation of a useful data structure, but I'll use the examples to highlight particular aspects of lock-free data structure design.

As already mentioned, lock-free data structures rely on the use of atomic operations and the associated memory-ordering guarantees in order to ensure that data becomes visible to other threads in the correct order. Initially, we'll use the default `memory_order_seq_cst` memory ordering for all atomic operations, because that's the easiest to reason about (remember that all `memory_order_seq_cst` operations form a total order). But for later examples we'll look at reducing some of the ordering constraints to `memory_order_acquire`, `memory_order_release`, or even `memory_order_relaxed`. Although none of these examples use mutex locks directly, it's worth bearing in mind that only `std::atomic_flag` is guaranteed not to use locks in the implementation. On some platforms what appears to be lock-free code might actually be using locks internal to the C++ Standard Library implementation (see chapter 5 for more details). On these platforms, a simple lock-based data structure might actually be more appropriate, but there's more to it than that; before choosing an implementation, you must identify your requirements and profile the various options that meet those requirements.

So, back to the beginning with the simplest of data structures: a stack.

7.2.1 Writing a thread-safe stack without locks

The basic premise of a stack is relatively simple: nodes are retrieved in the reverse order to which they were added—last in, first out (LIFO). It's therefore important to ensure that once a value is added to the stack, it can safely be retrieved immediately by another thread, and it's also important to ensure that only one thread returns a given value. The simplest stack is just a linked list; the `head` pointer identifies the first node (which will be the next to retrieve), and each node then points to the next node in turn.

Under such a scheme, adding a node is relatively simple:

1. Create a new node.
2. Set its `next` pointer to the current `head` node.
3. Set the `head` node to point to it.

This works fine in a single-threaded context, but if other threads are also modifying the stack, it's not enough. Crucially, if two threads are adding nodes, there's a race condition between steps 2 and 3: a second thread could modify the value of `head` between when your thread reads it in step 2 and you update it in step 3. This would then result in the changes made by that other thread being discarded or even worse consequences. Before we look at addressing this race condition, it's also important to note that once `head` has been updated to point to your new node, another thread could read that node. It's therefore vital that your new node is thoroughly prepared *before* `head` is set to point to it; you can't modify the node afterward.

OK, so what can you do about this nasty race condition? The answer is to use an atomic compare/exchange operation at step 3 to ensure that `head` hasn't been modified since you read it in step 2. If it has, you can loop and try again. The following listing shows how you can implement a thread-safe `push()` without locks.

Listing 7.2 Implementing `push()` without locks

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        T data;
        node* next;

        node(T const& data_):    <-1
            data(data_)
        {}
    };

    std::atomic<node*> head;
public:
    void push(T const& data)
    {
        node* const new_node=new node(data);    <-2      3
        new_node->next=head.load();                  <-|
        while(!head.compare_exchange_weak(new_node->next,new_node));    <-4
    }
};
```

This code neatly matches the three-point plan from above: create a new node ❷, set the node's `next` pointer to the current `head` ❸, and set the `head` pointer to the new node ❹. By populating the data in the `node` structure itself from the `node` constructor ❶, you've ensured that the node is ready to roll as soon as it's constructed, so that's the easy problem down. Then you use `compare_exchange_weak()` to ensure that the `head` pointer still has the same value as you stored in `new_node->next` ❸, and you set it to `new_node` if so. This bit of code also uses a nifty part of the compare/exchange functionality: if it returns `false` to indicate that the comparison failed (for example, because `head` was modified by another thread), the value supplied as the first parameter

(`new_node->next`) is updated to the current value of `head`. You therefore don't have to reload `head` each time through the loop, because the compiler does that for you. Also, because you're just looping directly on failure, you can use `compare_exchange_weak`, which can result in more optimal code than `compare_exchange_strong` on some architectures (see chapter 5).

So, you might not have a `pop()` operation yet, but you can quickly check `push()` against the guidelines. The only place that can throw an exception is the construction of the new `node` ❶, but this will clean up after itself, and the list hasn't been modified yet, so that's perfectly safe. Because you build the data to be stored as part of the `node`, and you use `compare_exchange_weak()` to update the `head` pointer, there are no problematic race conditions here. Once the compare/exchange succeeds, the node is on the list and ready for the taking. There are no locks, so there's no possibility of deadlock, and your `push()` function passes with flying colors.

Of course, now that you have a means of adding data to the stack, you need a way of getting it off again. On the face of it, this is quite simple:

1. Read the current value of `head`.
2. Read `head->next`.
3. Set `head` to `head->next`.
4. Return the `data` from the retrieved `node`.
5. Delete the retrieved node.

However, in the presence of multiple threads, this isn't so simple. If there are two threads removing items from the stack, they both might read the same value of `head` at step 1. If one thread then proceeds all the way through to step 5 before the other gets to step 2, the second thread will be dereferencing a dangling pointer. This is one of the biggest issues in writing lock-free code, so for now you'll just leave out step 5 and leak the nodes.

This doesn't resolve all the problems, though. There's another problem: if two threads read the same value of `head`, they'll return the same node. This violates the intent of the stack data structure, so you need to avoid this. You can resolve this the same way you resolved the race in `push()`: use compare/exchange to update `head`. If the compare/exchange fails, either a new node has been pushed on or another thread just popped the node you were trying to pop. Either way, you need to return to step 1 (although the compare/exchange call rereads `head` for you).

Once the compare/exchange call succeeds, you know you're the only thread that's popping the given node off the stack, so you can safely execute step 4. Here's a first cut at `pop()`:

```
template<typename T>
class lock_free_stack
{
public:
    void pop(T& result)
    {
```

```
        node* old_head=head.load();
        while(!head.compare_exchange_weak(old_head,old_head->next));
        result=old_head->data;
    }
};
```

Although this is nice and succinct, there are still a couple of problems aside from the leaking node. First, it doesn't work on an empty list: if `head` is a null pointer, it will cause undefined behavior as it tries to read the `next` pointer. This is easily fixed by checking for `nullptr` in the `while` loop and either throwing an exception on an empty stack or returning a `bool` to indicate success or failure.

The second problem is an exception-safety issue. When we first introduced the thread-safe stack back in chapter 3, you saw how just returning the object by value left you with an exception safety issue: if an exception is thrown when copying the return value, the value is lost. In that case, passing in a reference to the result was an acceptable solution because you could ensure that the stack was left unchanged if an exception was thrown. Unfortunately, here you don't have that luxury; you can only safely copy the data once you know you're the only thread returning the node, *which means the node has already been removed from the queue*. Consequently, passing in the target for the return value by reference is no longer an advantage: you might as well just return by value. If you want to return the value safely, you have to use the other option from chapter 3: return a (smart) pointer to the data value.

If you return a smart pointer, you can just return `nullptr` to indicate that there's no value to return, but this requires that the data be allocated on the heap. If you do the heap allocation as part of the `pop()`,you're *still* no better off, because the heap allocation might throw an exception. Instead, you can allocate the memory when you `push()` the data onto the stack—you have to allocate memory for the `node` anyway. Returning a `std::shared_ptr<>` won't throw an exception, so `pop()` is now safe. Putting all this together gives the following listing.

Listing 7.3 A lock-free stack that leaks nodes

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        node* next;

        node(T const& data_):
            data(std::make_shared<T>(data_))
        {}
    };

    std::atomic<node*> head;
public:
    void push(T const& data)
```

1 **Data is now held by pointer**

2 **Create std::shared_ptr for newly allocated T**

```
    {
        node* const new_node=new node(data);
        new_node->next=head.load();
        while(!head.compare_exchange_weak(new_node->next,new_node));
    }
    std::shared_ptr<T> pop()
    {
        node* old_head=head.load();
        while(old_head &&
            !head.compare_exchange_weak(old_head,old_head->next));
        return old_head ? old_head->data : std::shared_ptr<T>();
    }
};
```

❸ **Check old_head is not a null pointer before you dereference it**

❹

The data is held by the pointer now ❶, so you have to allocate the data on the heap in the node constructor ❷. You also have to check for a null pointer before you dereference `old_head` in the `compare_exchange_weak()` loop ❸. Finally, you either return the data associated with your node, if there is one, or a null pointer if not ❹. Note that although this is *lock-free*, it's not *wait-free*, because the `while` loops in both `push()` and `pop()` could in theory loop forever if the `compare_exchange_weak()` keeps failing.

If you have a garbage collector picking up after you (like in managed languages such as C# or Java), you're finished; the old node will be collected and recycled once it's no longer being accessed by any threads. However, not many C++ compilers ship with a garbage collector, so you generally have to tidy up after yourself.

7.2.2 Stopping those pesky leaks: managing memory in lock-free data structures

When we first looked at `pop()`, we opted to leak nodes in order to avoid the race condition where one thread deletes a node while another thread still holds a pointer to it that it's just about to dereference. However, leaking memory isn't acceptable in any sensible C++ program, so we have to do something about that. Now it's time to look at the problem and work out a solution.

The basic problem is that you want to free a node, but you can't do so until you're sure there are no other threads that still hold pointers to it. If only one thread ever calls `pop()` on a particular stack instance, you're home free. `push()` doesn't touch the node once it's been added to the stack, so the thread that called `pop()` must be the only thread that can touch the node, and it can safely delete it.

On the other hand, if you need to handle multiple threads calling `pop()` on the same stack instance, you need some way to track when it's safe to delete a node. This essentially means you need to write a special-purpose garbage collector just for `nodes`. Now, this might sound scary, but although it's certainly tricky, it's not *that* bad: you're only checking for `nodes`, and you're only checking for nodes accessed from `pop()`. You're not worried about nodes in `push()`, because they're only accessible from one thread until they're on the stack, whereas multiple threads might be accessing the same node in `pop()`.

If there are no threads calling `pop()`,it's perfectly safe to delete all the nodes currently awaiting deletion. Therefore, if you add the nodes to a "to be deleted" list when

you've extracted the data, then you can delete them all when there are no threads calling `pop()`. How do you know there aren't any threads calling `pop()`? Simple—count them. If you increment a counter on entry and decrement that counter on exit, it's safe to delete the nodes from the "to be deleted" list when the counter is zero. Of course, it will have to be an atomic counter so it can safely be accessed from multiple threads. The following listing shows the amended `pop()` function, and listing 7.5 shows the supporting functions for such an implementation.

Listing 7.4 Reclaiming nodes when no threads are in `pop()`

```
template<typename T>
class lock_free_stack
{
private:
    std::atomic<unsigned> threads_in_pop;
    void try_reclaim(node* old_head);
public:
    std::shared_ptr<T> pop()
    {
        ++threads_in_pop;
        node* old_head=head.load();
        while(old_head &&
              !head.compare_exchange_weak(old_head,old_head->next));
        std::shared_ptr<T> res;
        if(old_head)
        {
            res.swap(old_head->data);
        }
        try_reclaim(old_head);
        return res;
    }
};
```

❶ Atomic variable

❷ Increase counter before doing anything else

❸ Reclaim deleted nodes if you can

❹ Extract data from node rather than copying pointer

The atomic variable `threads_in_pop` ❶ is used to count the threads currently trying to pop an item off the stack. It's incremented at the start of `pop()` ❷ and decremented inside `try_reclaim()`, which is called once the node has been removed ❹. Because you're going to potentially delay the deletion of the node itself, you can use `swap()` to remove the data from the node ❸ rather than just copying the pointer, so that the data will be deleted automatically when you no longer need it rather than it being kept alive because there's still a reference in a not-yet-deleted node. The next listing shows what goes into `try_reclaim()`.

Listing 7.5 The reference-counted reclamation machinery

```
template<typename T>
class lock_free_stack
{
private:
    std::atomic<node*> to_be_deleted;

    static void delete_nodes(node* nodes)
    {
```

```
        while(nodes)
        {
            node* next=nodes->next;
            delete nodes;
            nodes=next;
         }
    }
    void try_reclaim(node* old_head)
    {
        if(threads_in_pop==1)                                    ❶
        {
            node* nodes_to_delete=to_be_deleted.exchange(nullptr);
            if(!--threads_in_pop)                                ❸
            {
                delete_nodes(nodes_to_delete);                   ❹
            }
            else if(nodes_to_delete)                             ❺
            {
                chain_pending_nodes(nodes_to_delete);            ❻
            }
            delete old_head;                                     ❼
        }
        else
        {
            chain_pending_node(old_head);                        ❽
            --threads_in_pop;
        }
    }
    void chain_pending_nodes(node* nodes)
    {
        node* last=nodes;
        while(node* const next=last->next)                       ❾
        {
            last=next;
        }
        chain_pending_nodes(nodes,last);
    }

    void chain_pending_nodes(node* first,node* last)
    {
        last->next=to_be_deleted;                                ❿
        while(!to_be_deleted.compare_exchange_weak(              ⓫
                  last->next,first));
    }

    void chain_pending_node(node* n)
    {
        chain_pending_nodes(n,n);                                ⓬
    }
};
```

Claim list of to-be-deleted nodes ❷

Are you the only thread in pop()?

Follow the next pointer chain to the end

Loop to guarantee that last->next is correct

If the count of `threads_in_pop` is 1 when you're trying to reclaim the node ❶, you're the only thread currently in `pop()`, which means it's safe to delete the node you just removed ❼, and it *may* also be safe to delete the pending nodes. If the count is *not* 1, it's not safe to delete any nodes, so you have to add the node to the pending list ❽.

Assume for a moment that `threads_in_pop` is 1. You now need to try to reclaim the pending nodes; if you don't, they'll stay pending until you destroy the stack. To do this, you first claim the list for yourself with an atomic `exchange` operation ❷ and then decrement the count of `threads_in_pop` ❸. If the count is zero after the decrement, you know that no other thread can be accessing this list of pending nodes. There may be new pending nodes, but you're not bothered about them for now, as long as it's safe to reclaim your list. You can then just call `delete_nodes` to iterate down the list and delete them ❹.

If the count is *not* zero after the decrement, it's not safe to reclaim the nodes, so if there are any ❺, you must chain them back onto the list of nodes pending deletion ❻. This can happen if there are multiple threads accessing the data structure concurrently. Other threads might have called `pop()` in between the first test of `threads_in_pop` ❶ and the "claiming" of the list ❷, potentially adding new nodes to the list that are still being accessed by one or more of those other threads. In figure 7.1, thread C adds node Y to the `to_be_deleted` list, even though thread B is still referencing it as `old_head`, and will thus try and read its `next` pointer. Thread A can't therefore delete the nodes without potentially causing undefined behavior for thread B.

To chain the nodes that are pending deletion onto the pending list, you reuse the `next` pointer from the nodes to link them together. In the case of relinking an existing chain back onto the list, you traverse the chain to find the end ❾, replace the `next` pointer from the last node with the current `to_be_deleted` pointer ❿, and store the first node in the chain as the new `to_be_deleted` pointer ⓫. You have to use `compare_exchange_weak` in a loop here in order to ensure that you don't leak any nodes that have been added by another thread. This has the benefit of updating the `next` pointer from the end of the chain if it has been changed. Adding a single node onto the list is a special case where the first node in the chain to be added is the same as the last one ⓬.

This works reasonably well in low-load situations, where there are suitable *quiescent* points at which no threads are in `pop()`. However, this is potentially a transient situation, which is why you need to test that the `threads_in_pop` count decrements to zero ❸ before doing the reclaim and why this test occurs *before* you delete the just-removed node ❼. Deleting a node is potentially a time-consuming operation, and you want the window in which other threads can modify the list to be as small as possible. The longer the time between when the thread first finds `threads_in_pop` to be equal to 1 and the attempt to delete the nodes, the more chance there is that another thread has called `pop()`, and that `threads_in_pop` is no longer equal to 1, thus preventing the nodes from actually being deleted.

In high-load situations, there may *never* be such a quiescent state, because other threads have entered `pop()` before all the threads initially in `pop()` have left. Under such a scenario, the `to_be_deleted` list would grow without bounds, and you'd be essentially leaking memory again. If there aren't going to be any quiescent periods, you need to find an alternative mechanism for reclaiming the nodes. The key is to identify when no more threads are accessing a particular node so that it can be reclaimed. By far the easiest such mechanism to reason about is the use of *hazard pointers*.

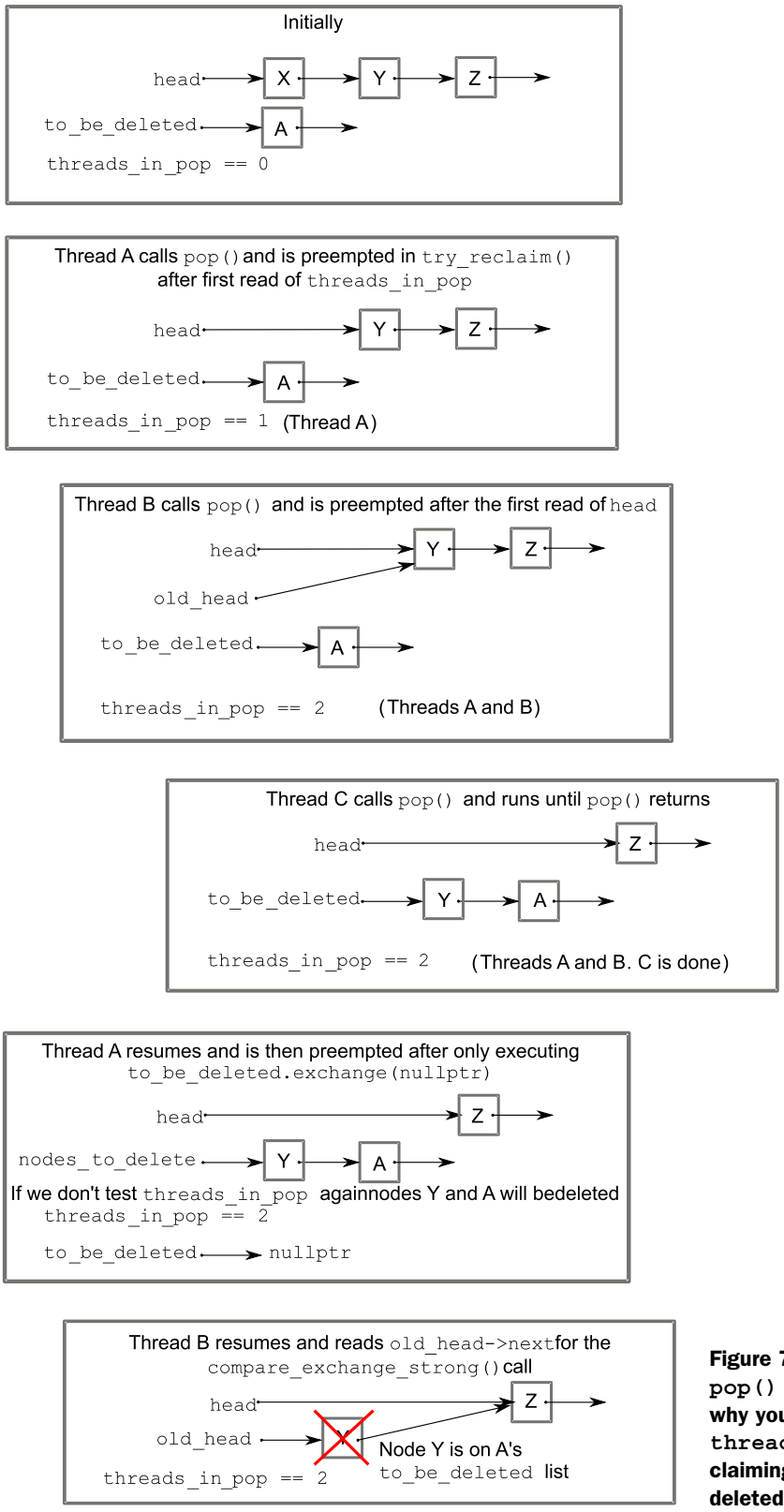

Figure 7.1 Three threads call `pop()` concurrently, showing why you must check `threads_in_pop` after claiming the nodes to be deleted in `try_reclaim()`.

7.2.3 Detecting nodes that can't be reclaimed using hazard pointers

The term *hazard pointers* is a reference to a technique discovered by Maged Michael.[1] They are so called because deleting a node that might still be referenced by other threads is hazardous. If other threads do indeed hold references to that node and proceed to access the node through that reference, you have undefined behavior. The basic idea is that if a thread is going to access an object that another thread might want to delete, it first sets a hazard pointer to reference the object, thus informing the other thread that deleting the object would indeed be hazardous. Once the object is no longer needed, the hazard pointer is cleared. If you've ever watched the Oxford/Cambridge boat race, you've seen a similar mechanism used when starting the race: the cox of either boat can raise their hand to indicate that they aren't ready. While either cox has their hand raised, the umpire may not start the race. If both coxes have their hands down, the race may start, but a cox may raise their hand again if the race hasn't started and they feel the situation has changed.

When a thread wishes to delete an object, it must first check the hazard pointers belonging to the other threads in the system. If none of the hazard pointers reference the object, it can safely be deleted. Otherwise, it must be left until later. Periodically, the list of objects that have been left until later is checked to see if any of them can now be deleted.

Described at such a high level, it sounds relatively straightforward, so how do you do this in C++?

Well, first off you need a location in which to store the pointer to the object you're accessing, the *hazard pointer* itself. This location must be visible to all threads, and you need one of these for each thread that might access the data structure. Allocating them correctly and efficiently can be a challenge, so you'll leave that for later and assume you have a function `get_hazard_pointer_for_current_thread()` that returns a reference to your hazard pointer. You then need to set it when you read a pointer that you intend to dereference—in this case the `head` value from the list:

```
std::shared_ptr<T> pop()
{
    std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
    node* old_head=head.load();    <-1
    node* temp;                    (1)
    do
    {
        temp=old_head;
        hp.store(old_head);        <-2
        old_head=head.load();
    } while(old_head!=temp);       <-3
    // ...
}
```

[1] "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes," Maged M. Michael, in *PODC '02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (2002), ISBN 1-58113-485-1.

You have to do this in a `while` loop to ensure that the `node` hasn't been deleted between the reading of the old `head` pointer ❶ and the setting of the hazard pointer ❷. During this window no other thread knows you're accessing this particular node. Fortunately, if the old `head` node is going to be deleted, `head` itself must have changed, so you can check this and keep looping until you know that the `head` pointer still has the same value you set your hazard pointer to ❸. Using hazard pointers like this relies on the fact that it's safe to use the value of a pointer after the object it references has been deleted. This is technically undefined behavior if you are using the default implementation of `new` and `delete`, so either you need to ensure that your implementation permits it, or you need to use a custom allocator that permits such usage.

Now that you've set your hazard pointer, you can proceed with the rest of `pop()`, safe in the knowledge that no other thread will delete the nodes from under you. Well, almost: every time you reload `old_head`, you need to update the hazard pointer before you dereference the freshly read pointer value. Once you've extracted a node from the list, you can clear your hazard pointer. If there are no other hazard pointers referencing your node, you can safely delete it; otherwise, you have to add it to a list of nodes to be deleted later. The following listing shows a full implementation of `pop()` using such a scheme.

Listing 7.6 An implementation of `pop()` using hazard pointers

```
std::shared_ptr<T> pop()
{
    std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
    node* old_head=head.load();
    do
    {
        node* temp;
        do                                          ❶ Loop until you've set the
        {                                             hazard pointer to head
            temp=old_head;
            hp.store(old_head);
            old_head=head.load();
        } while(old_head!=temp);
    }
    while(old_head &&
          !head.compare_exchange_strong(old_head,old_head->next));
    hp.store(nullptr);                              ❷ Clear hazard pointer
    std::shared_ptr<T> res;                           once you're finished
    if(old_head)
    {
        res.swap(old_head->data);
        if(outstanding_hazard_pointers_for(old_head))   ❸ Check for hazard
        {                                                 pointers referencing
            reclaim_later(old_head);                      a node before you
        }                                                 delete it
        else
        {
            delete old_head;                            ❺
        }
        delete_nodes_with_no_hazards();                 ❻
    }                                                   ❹
    return res;
}
```

```
    }
    return res;
}
```

First off, you've moved the loop that sets the hazard pointer *inside* the outer loop for reloading `old_head` if the compare/exchange fails ❶. You're using `compare_exchange_strong()` here because you're actually doing work inside the `while` loop: a spurious failure on `compare_exchange_weak()` would result in resetting the hazard pointer unnecessarily. This ensures that the hazard pointer is correctly set before you dereference `old_head`. Once you've claimed the node as yours, you can clear your hazard pointer ❷. If you did get a node, you need to check the hazard pointers belonging to other threads to see if they reference it ❸. If so, you can't delete it just yet, so you must put it on a list to be reclaimed later ❹; otherwise, you can delete it right away ❺. Finally, you put in a call to check for any nodes for which you had to call `reclaim_later()`. If there are no longer any hazard pointers referencing those nodes, you can safely delete them ❻. Any nodes for which there are still outstanding hazard pointers will be left for the next thread that calls `pop()`.

Of course, there's still a lot of detail hidden in these new functions—`get_hazard_pointer_for_current_thread()`, `reclaim_later()`, `outstanding_hazard_pointers_for()`, and `delete_nodes_with_no_hazards()`—so let's draw back the curtain and look at how they work.

The exact scheme for allocating hazard pointer instances to threads used by `get_hazard_pointer_for_current_thread()` doesn't really matter for the program logic (although it can affect the efficiency, as you'll see later). So for now you'll go with a simple structure: a fixed-size array of pairs of thread IDs and pointers. `get_hazard_pointer_for_current_thread()` then searches through the array to find the first free slot and sets the ID entry of that slot to the ID of the current thread. When the thread exits, the slot is freed by resetting the ID entry to a default-constructed `std::thread::id()`. This is shown in the following listing.

Listing 7.7 A simple implementation of `get_hazard_pointer_for_current_thread()`

```
unsigned const max_hazard_pointers=100;
struct hazard_pointer
{
    std::atomic<std::thread::id> id;
    std::atomic<void*> pointer;
};
hazard_pointer hazard_pointers[max_hazard_pointers];

class hp_owner
{
    hazard_pointer* hp;

public:
    hp_owner(hp_owner const&)=delete;
    hp_owner operator=(hp_owner const&)=delete;
```

```
    hp_owner():
        hp(nullptr)
    {
        for(unsigned i=0;i<max_hazard_pointers;++i)
        {
            std::thread::id old_id;
            if(hazard_pointers[i].id.compare_exchange_strong(
                old_id,std::this_thread::get_id()))
            {
                hp=&hazard_pointers[i];
                break;
            }
        }
        if(!hp)     <-- 1
        {
            throw std::runtime_error("No hazard pointers available");
        }
    }

    std::atomic<void*>& get_pointer()
    {
        return hp->pointer;
    }

    ~hp_owner()     <-- 2
    {
        hp->pointer.store(nullptr);
        hp->id.store(std::thread::id());
    }
};

std::atomic<void*>& get_hazard_pointer_for_current_thread()     <-- 3
{
    thread_local static hp_owner hazard;
    return hazard.get_pointer();
}
```

Try to claim ownership of a hazard pointer

Each thread has its own hazard pointer (4)

(5)

The actual implementation of `get_hazard_pointer_for_current_thread()` itself is deceptively simple (3): it has a `thread_local` variable of type `hp_owner` (4) that stores the hazard pointer for the current thread. It then just returns the pointer from that object (5). This works as follows: The first time *each thread* calls this function, a new instance of `hp_owner` is created. The constructor for this new instance (1) then searches through the table of owner/pointer pairs looking for an entry without an owner. It uses `compare_exchange_strong()` to check for an entry without an owner and claim it in one go (2). If the `compare_exchange_strong()` fails, another thread owns that entry, so you move on to the next. If the exchange succeeds, you've successfully claimed the entry for the current thread, so you store it and stop the search (3). If you get to the end of the list without finding a free entry (4), there are too many threads using hazard pointers, so you throw an exception.

Once the `hp_owner` instance has been created for a given thread, further accesses are much faster because the pointer is cached, so the table doesn't have to be scanned again.

When each thread exits, if an instance of `hp_owner` was created for that thread, then it's destroyed. The destructor then resets the actual pointer to `nullptr` before setting the owner ID to `std::thread::id()`, allowing another thread to reuse the entry later ❺.

With this implementation of `get_hazard_pointer_for_current_thread()`, the implementation of `outstanding_hazard_pointers_for()` is really simple: just scan through the hazard pointer table looking for entries:

```
bool outstanding_hazard_pointers_for(void* p)
{
    for(unsigned i=0;i<max_hazard_pointers;++i)
    {
        if(hazard_pointers[i].pointer.load()==p)
        {
            return true;
        }
    }
    return false;
}
```

It's not even worth checking whether each entry has an owner: unowned entries will have a null pointer, so the comparison will return `false` anyway, and it simplifies the code.

`reclaim_later()` and `delete_nodes_with_no_hazards()` can then work on a simple linked list; `reclaim_later()` just adds nodes to the list, and `delete_nodes_with_no_hazards()` scans through the list, deleting entries with no outstanding hazards. The next listing shows just such an implementation.

Listing 7.8 A simple implementation of the reclaim functions

```
template<typename T>
void do_delete(void* p)
{
    delete static_cast<T*>(p);
}

struct data_to_reclaim
{
    void* data;
    std::function<void(void*)> deleter;
    data_to_reclaim* next;

    template<typename T>
    data_to_reclaim(T* p):      <-- 1
        data(p),
        deleter(&do_delete<T>),
        next(0)
    {}

    ~data_to_reclaim()
    {
        deleter(data);      <-- 2
    }
};

std::atomic<data_to_reclaim*> nodes_to_reclaim;
```

```
void add_to_reclaim_list(data_to_reclaim* node)     <-- 3
{
    node->next=nodes_to_reclaim.load();
    while(!nodes_to_reclaim.compare_exchange_weak(node->next,node));
}

template<typename T>
void reclaim_later(T* data)     <-- 4
{
    add_to_reclaim_list(new data_to_reclaim(data));     <-- 5
}

void delete_nodes_with_no_hazards()
{
    data_to_reclaim* current=nodes_to_reclaim.exchange(nullptr);     <-- 6
    while(current)
    {
        data_to_reclaim* const next=current->next;
        if(!outstanding_hazard_pointers_for(current->data))     <-- 7
        {
            delete current;     <-- 8
        }
        else
        {
            add_to_reclaim_list(current);     <-- 9
        }
        current=next;
    }
}
```

First off, I expect you've spotted that `reclaim_later()` is a function template rather than a plain function ❹. This is because hazard pointers are a general-purpose utility, so you don't want to tie yourselves to stack nodes. You've been using `std::atomic<void*>` for storing the pointers already. You therefore need to handle any pointer type, but you can't use `void*` because you want to delete the data items when you can, and `delete` requires the real type of the pointer. The constructor of `data_to_reclaim` handles that nicely, as you'll see in a minute: `reclaim_later()` just creates a new instance of `data_to_reclaim` for your pointer and adds it to the reclaim list ❺. `add_to_reclaim_list()` itself ❸ is just a simple `compare_exchange_weak()` loop on the list head like you've seen before.

So, back to the constructor of `data_to_reclaim` ❶: the constructor is also a template. It stores the data to be deleted as a `void*` in the `data` member and then stores a pointer to the appropriate instantiation of `do_delete()`—a simple function that casts the supplied `void*` to the chosen pointer type and then deletes the pointed-to object. `std::function<>` wraps this function pointer safely, so that the destructor of `data_to_reclaim` can then delete the data just by invoking the stored function ❷.

The destructor of `data_to_reclaim` isn't called when you're adding nodes to the list; it's called when there are no more hazard pointers to that node. This is the responsibility of `delete_nodes_with_no_hazards()`.

`delete_nodes_with_no_hazards()` first claims the entire list of nodes to be reclaimed for itself with a simple `exchange()` ❻. This simple but crucial step ensures

that this is the only thread trying to reclaim this particular set of nodes. Other threads are now free to add further nodes to the list or even try to reclaim them without impacting the operation of this thread.

Then, as long as there are still nodes left in the list, you check each node in turn to see if there are any outstanding hazard pointers ❼. If there aren't, you can safely delete the entry (and thus clean up the stored data) ❽. Otherwise, you just add the item back on the list for reclaiming later ❾.

Although this simple implementation does indeed safely reclaim the deleted nodes, it adds quite a bit of overhead to the process. Scanning the hazard pointer array requires checking `max_hazard_pointers` atomic variables, and this is done for every `pop()` call. Atomic operations are inherently slow—often 100 times slower than an equivalent non-atomic operation on desktop CPUs—so this makes `pop()` an expensive operation. Not only do you scan the hazard pointer list for the node you're about to remove, but you also scan it for each node in the waiting list. Clearly this is a bad idea. There may well be `max_hazard_pointers` nodes in the list, and you're checking all of them against `max_hazard_pointers` stored hazard pointers. Ouch! There has to be a better way.

Better reclamation strategies using hazard pointers

Of course, there *is* a better way. What I've shown here is a simple and naïve implementation of hazard pointers to help explain the technique. The first thing you can do is trade memory for performance. Rather than checking every node on the reclamation list every time you call `pop()`, you don't try to reclaim any nodes at all unless there are more than `max_hazard_pointers` nodes on the list. That way you're guaranteed to be able to reclaim at least one node. If you just wait until there are `max_hazard_pointers+1` nodes on the list, you're not much better off. Once you get to `max_hazard_pointers` nodes, you'll be trying to reclaim nodes for most calls to `pop()`, so you're not doing much better. But if you wait until there are `2*max_hazard_pointers` nodes on the list, you're guaranteed to be able to reclaim at least `max_hazard_pointers` nodes, and it will then be at least `max_hazard_pointers` calls to `pop()` before you try to reclaim any nodes again. This is much better. Rather than checking around `max_hazard_pointers` nodes every call to `push()` (and not necessarily reclaiming any), you're checking `2*max_hazard_pointers` nodes every `max_hazard_pointers` calls to `pop()` and reclaiming at least `max_hazard_pointers` nodes. That's effectively two nodes checked for every `pop()`, one of which is reclaimed.

Even this has a downside (beyond the increased memory usage): you now have to count the nodes on the reclamation list, which means using an atomic count, and you still have multiple threads competing to access the reclamation list itself. If you have memory to spare, you can trade increased memory usage for an even better reclamation scheme: each thread keeps its own reclamation list in a thread-local variable. There's thus no need for atomic variables for the count or the list access. Instead, you have `max_hazard_pointers*max_hazard_pointers` nodes allocated. If a thread exits before all its nodes have been reclaimed, they can be stored in the global list as before and added to the local list of the next thread doing a reclamation process.

Another downside of hazard pointers is that they're covered by a patent application submitted by IBM.[2] If you write software for use in a country where the patents are valid, you need to make sure you have a suitable licensing arrangement in place. This is something common to many of the lock-free memory reclamation techniques; this is an active research area, so large companies are taking out patents where they can. You may well be asking why I've devoted so many pages to a technique that many people will be unable to use, and that's a fair question. First, it may be possible to use the technique without paying for a license. For example, if you're developing free software licensed under the GPL,[3] your software may be covered by IBM's statement of non-assertion.[4] Second, and most important, the explanation of the techniques shows some of the things that are important to think about when writing lock-free code, such as the costs of atomic operations.

So, are there any unpatented memory reclamation techniques that can be used with lock-free code? Luckily, there are. One such mechanism is reference counting.

7.2.4 Detecting nodes in use with reference counting

Back in section 7.2.2, you saw that the problem with deleting nodes is detecting which nodes are still being accessed by reader threads. If you could safely identify precisely which nodes were being referenced and when no threads were accessing these nodes, you could delete them. Hazard pointers tackle the problem by storing a list of the nodes in use. Reference counting tackles the problem by storing a count of the number of threads accessing each node.

This may seem nice and straightforward, but it's quite hard to manage in practice. At first, you might think that something like `std::shared_ptr<>` would be up to the task; after all, it's a reference-counted pointer. Unfortunately, although some operations on `std::shared_ptr<>` are atomic, they aren't guaranteed to be lock-free. Although by itself this is no different than any of the operations on the atomic types, `std::shared_ptr<>` is intended for use in many contexts, and making the atomic operations lock-free would likely impose an overhead on all uses of the class. If your platform supplies an implementation for which `std::atomic_is_lock_free(&some_shared_ptr)` returns `true`, the whole memory reclamation issue goes away. Just use `std::shared_ptr<node>` for the list, as in the following listing.

Listing 7.9 A lock-free stack using a lock-free `std::shared_ptr<>` implementation

```
template<typename T>
class lock_free_stack
{
private:
```

[2] Maged M. Michael, U.S. Patent and Trademark Office application number 20040107227, "Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation."

[3] GNU General Public License http://www.gnu.org/licenses/gpl.html.

[4] IBM Statement of Non-Assertion of Named Patents Against OSS, http://www.ibm.com/ibm/licensing/patents/pledgedpatents.pdf.

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

        node(T const& data_):
            data(std::make_shared<T>(data_))
        {}
    };

    std::shared_ptr<node> head;
public:
    void push(T const& data)
    {
        std::shared_ptr<node> const new_node=std::make_shared<node>(data);
        new_node->next=head.load();
        while(!std::atomic_compare_exchange_weak(&head,
                  &new_node->next,new_node));
    }
    std::shared_ptr<T> pop()
    {
        std::shared_ptr<node> old_head=std::atomic_load(&head);
        while(old_head && !std::atomic_compare_exchange_weak(&head,
                  &old_head,old_head->next));
        return old_head ? old_head->data : std::shared_ptr<T>();
    }
};
```

In the probable case that your `std::shared_ptr<>` implementation isn't lock-free, you need to manage the reference counting manually.

One possible technique involves the use of not one but two reference counts for each node: an internal count and an external count. The sum of these values is the total number of references to the node. The external count is kept alongside the pointer to the node and is increased every time the pointer is read. When the reader is finished with the node, it decreases the *internal* count. A simple operation that reads the pointer will thus leave the external count increased by one and the internal count decreased by one when it's finished.

When the external count/pointer pairing is no longer required (that is, the node is no longer accessible from a location accessible to multiple threads), the internal count is increased by the value of the external count minus one and the external counter is discarded. Once the internal count is equal to zero, there are no outstanding references to the node and it can be safely deleted. It's still important to use atomic operations for updates of shared data. Let's now look at an implementation of a lock-free stack that uses this technique to ensure that the nodes are reclaimed only when it's safe to do so.

The following listing shows the internal data structure and the implementation of `push()`, which is nice and straightforward.

Listing 7.10 Pushing a node on a lock-free stack using split reference counts

```
template<typename T>
class lock_free_stack
```

```
{
private:
    struct node;

    struct counted_node_ptr     <-- 1
    {
        int external_count;
        node* ptr;
    };

    struct node
    {
        std::shared_ptr<T> data;                <-- 2
        std::atomic<int> internal_count;
        counted_node_ptr next;          <-- 3

        node(T const& data_):
            data(std::make_shared<T>(data_)),
            internal_count(0)
        {}
    };

    std::atomic<counted_node_ptr> head;     <-- 4

public:
    ~lock_free_stack()
    {
        while(pop());
    }

    void push(T const& data)     <-- 5
    {
        counted_node_ptr new_node;
        new_node.ptr=new node(data);
        new_node.external_count=1;
        new_node.ptr->next=head.load();
        while(!head.compare_exchange_weak(new_node.ptr->next,new_node));
    }
};
```

First, the external count is wrapped together with the node pointer in the `counted_node_ptr` structure ❶. This can then be used for the `next` pointer in the `node` structure ❸ alongside the internal count ❷. Because `counted_node_ptr` is just a simple `struct`, you can use it with the `std::atomic<>` template for the `head` of the list ❹.

On those platforms that support a double-word-compare-and-swap operation, this structure will be small enough for `std::atomic<counted_node_ptr>` to be lock-free. If it isn't on your platform, you might be better off using the `std::shared_ptr<>` version from listing 7.9, because `std::atomic<>` will use a mutex to guarantee atomicity when the type is too large for the platform's atomic instructions (thus rendering your "lock-free" algorithm lock-based after all). Alternatively, if you're willing to limit the size of the counter, and you know that your platform has spare bits in a pointer (for example, because the address space is only 48 bits but a pointer is 64 bits), you can store the count inside the spare bits of the pointer to fit it all back in a single machine word. Such tricks require platform-specific knowledge and are thus outside the scope of this book.

`push()` is relatively simple ❺. You construct a `counted_node_ptr` that refers to a freshly allocated `node` with associated data and set the `next` value of the `node` to the current value of `head`. You can then use `compare_exchange_weak()` to set the value of `head`, just as in the previous listings. The counts are set up so the `internal_count` is zero, and the `external_count` is one. Because this is a new node, there's currently only one external reference to the node (the `head` pointer itself).

As usual, the complexities come to light in the implementation of `pop()`, which is shown in the following listing.

Listing 7.11 Popping a node from a lock-free stack using split reference counts

```
template<typename T>
class lock_free_stack
{
private:
    void increase_head_count(counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;

        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!head.compare_exchange_strong(old_counter,new_counter));  <-1

        old_counter.external_count=new_counter.external_count;
    }

public:
    std::shared_ptr<T> pop()#
    {
        counted_node_ptr old_head=head.load();
        for(;;)
        {
            increase_head_count(old_head);
            node* const ptr=old_head.ptr;     <-2
            if(!ptr)
            {
                return std::shared_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next))     <-3
            {
                std::shared_ptr<T> res;
                res.swap(ptr->data);      <-4

                int const count_increase=old_head.external_count-2;     <-5

                if(ptr->internal_count.fetch_add(count_increase)==     <-6
                   -count_increase)
                {
                    delete ptr;
                }

                return res;     <-7
```

```
            }
            else if(ptr->internal_count.fetch_sub(1)==1)
            {
                delete ptr;     <-- 8
            }
        }
    }
};
```

This time, once you've loaded the value of `head`, you must first increase the count of external references to the `head` node to indicate that you're referencing it and to ensure that it's safe to dereference it. If you dereference the pointer *before* increasing the reference count, another thread could free the node before you access it, thus leaving you with a dangling pointer. *This is the primary reason for using the split reference count:* by incrementing the external reference count, you ensure that the pointer remains valid for the duration of your access. The increment is done with a `compare_exchange_strong()` loop ❶ that compares and sets the whole structure to ensure that the pointer hasn't been changed by another thread in the meantime.

Once the count has been increased, you can safely dereference the `ptr` field of the value loaded from `head` in order to access the pointed-to node ❷. If the pointer is a null pointer, you're at the end of the list: no more entries. If the pointer isn't a null pointer, you can try to remove the node by a `compare_exchange_strong()` call on `head` ❸.

If the `compare_exchange_strong()` succeeds, you've taken ownership of the node and can swap out the `data` in preparation for returning it ❹. This ensures that the data isn't kept alive just because other threads accessing the stack happen to still have pointers to its node. Then you can add the external count to the internal count on the node with an atomic `fetch_add` ❻. If the reference count is now zero, the *previous* value (which is what `fetch_add` returns) was the negative of what you just added, in which case you can delete the node. It's important to note that the value you add is actually *two less* than the external count ❺; you've removed the node from the list, so you drop one off the count for that, and you're no longer accessing the node from this thread, so you drop another off the count for that. Whether or not you deleted the node, you've finished, so you can return the data ❼.

If the compare/exchange ❸ *fails*, another thread removed your node before you did, or another thread added a new node to the stack. Either way, you need to start again with the fresh value of `head` returned by the compare/exchange call. But first you must decrease the reference count on the node you were trying to remove. This thread won't access it anymore. If you're the last thread to hold a reference (because another thread removed it from the stack), the internal reference count will be 1, so subtracting 1 will set the count to zero. In this case, you can delete the node here before you loop ❽.

So far, you've been using the default `std::memory_order_seq_cst` memory ordering for all your atomic operations. On most systems these are more expensive in terms of execution time and synchronization overhead than the other memory orderings, and on some systems considerably so. Now that you have the logic of your data structure

right, you can think about relaxing some of these memory-ordering requirements; you don't want to impose any unnecessary overhead on the users of the stack. So, before leaving your stack behind and moving on to the design of a lock-free queue, let's examine the stack operations and ask ourselves, can we use more relaxed memory orderings for some operations and still get the same level of safety?

7.2.5 Applying the memory model to the lock-free stack

Before you go about changing the memory orderings, you need to examine the operations and identify the required relationships between them. You can then go back and find the minimum memory orderings that provide these required relationships. In order to do this, you'll have to look at the situation from the point of view of threads in several different scenarios. The simplest possible scenario has to be where one thread pushes a data item onto the stack and another thread then pops that data item off the stack some time later, so we'll start from there.

In this simple case, three important pieces of data are involved. First is the `counted_node_ptr` used for transferring the data: `head`. Second is the `node` structure that `head` refers to, and third is the data item pointed to by that node.

The thread doing the `push()` first constructs the data item and the `node` and then sets `head`. The thread doing the `pop()` first loads the value of `head`, then does a compare/exchange loop on `head` to increase the reference count, and then reads the `node` structure to obtain the `next` value. Right here you can see a required relationship; the `next` value is a plain nonatomic object, so in order to read this safely, there must be a happens-before relationship between the store (by the pushing thread) and the load (by the popping thread). Because the only atomic operation in the `push()` is the `compare_exchange_weak()`, and you need a *release* operation to get a happens-before relationship between threads, the `compare_exchange_weak()` must be `std::memory_order_release` or stronger. If the `compare_exchange_weak()` call fails, nothing has changed and you keep looping, so you need only `std::memory_order_relaxed` in that case:

```
void push(T const& data)
{
    counted_node_ptr new_node;
    new_node.ptr=new node(data);
    new_node.external_count=1;
    new_node.ptr->next=head.load(std::memory_order_relaxed)
    while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
        std::memory_order_release,std::memory_order_relaxed));
}
```

What about the `pop()` code? In order to get the happens-before relationship you need, you must have an operation that's `std::memory_order_acquire` or stronger before the access to `next`. The pointer you dereference to access the `next` field is the old value read by the `compare_exchange_strong()` in `increase_head_count()`, so you need the ordering on that if it succeeds. As with the call in `push()`, if the exchange fails, you just loop again, so you can use relaxed ordering on failure:

```
void increase_head_count(counted_node_ptr& old_counter)
{
    counted_node_ptr new_counter;

    do
    {
        new_counter=old_counter;
        ++new_counter.external_count;
    }
    while(!head.compare_exchange_strong(old_counter,new_counter,
        std::memory_order_acquire,std::memory_order_relaxed));

    old_counter.external_count=new_counter.external_count;
}
```

If the `compare_exchange_strong()` call succeeds, you know that the value read had the `ptr` field set to what's now stored in `old_counter`. Because the store in `push()` was a release operation, and this `compare_exchange_strong()` is an acquire operation, the store synchronizes with the load and you have a happens-before relationship. Consequently, the store to the `ptr` field in the `push()` happens before the `ptr->next` access in `pop()`, and you're safe.

Note that the memory ordering on the initial `head.load()` didn't matter to this analysis, so you can safely use `std::memory_order_relaxed` for that.

Next up, the `compare_exchange_strong()` to set `head` to `old_head.ptr->next`. Do you need anything from this operation to guarantee the data integrity of this thread? If the exchange succeeds, you access `ptr->data`, so you need to ensure that the store to `ptr->data` in the `push()` thread happens before the load. However, you already have that guarantee: the acquire operation in `increase_head_count()` ensures that there's a synchronizes-with relationship between the store in the `push()` thread and that compare/exchange. Because the store to `data` in the `push()` thread is sequenced before the store to `head` and the call to `increase_head_count()` is sequenced before the load of `ptr->data`, there's a happens-before relationship, and all is well even if this compare/exchange in `pop()` uses `std::memory_order_relaxed`. The only other place where `ptr->data` is changed is the very call to `swap()` that you're looking at, and no other thread can be operating on the same node; that's the whole point of the compare/exchange.

If the `compare_exchange_strong()` fails, the new value of `old_head` isn't touched until next time around the loop, and you already decided that the `std::memory_order_acquire` in `increase_head_count()` was enough, so `std::memory_order_relaxed` is enough there also.

What about other threads? Do you need anything stronger here to ensure other threads are still safe? The answer is, no, because `head` is only ever modified by compare/exchange operations. Because these are read-modify-write operations, they form part of the release sequence headed by the compare/exchange in `push()`. Therefore, the `compare_exchange_weak()` in `push()` synchronizes with a call to `compare_exchange_strong()` in `increase_head_count()`, which reads the value stored, even if many other threads modify `head` in the meantime.

So you've nearly finished: the only remaining operations to deal with are the `fetch_add()` operations for modifying the reference count. The thread that got to return the data from this node can proceed, safe in the knowledge that no other thread can have modified the node data. However, any thread that did *not* successfully retrieve the data knows that another thread *did* modify the node data; it used `swap()` to extract the referenced data item. Therefore you need to ensure that the `swap()` happens-before the `delete` in order to avoid a data race. The easy way to do this is to make the `fetch_add()` in the successful-return branch use `std::memory_order_release` and the `fetch_add()` in the loop-again branch use `std::memory_order_acquire`. However, this is still overkill: only one thread does the `delete` (the one that sets the count to zero), so only that thread needs to do an acquire operation. Thankfully, because `fetch_add()` is a read-modify-write operation, it forms part of the release sequence, so you can do that with an additional `load()`. If the loop-again branch decreases the reference count to zero, it can reload the reference count with `std::memory_order_acquire` in order to ensure the required synchronizes-with relationship, and the `fetch_add()` itself can use `std::memory_order_relaxed`. The final stack implementation with the new version of `pop()` is shown here.

Listing 7.12 A lock-free stack with reference counting and relaxed atomic operations

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

    struct counted_node_ptr
    {
        int external_count;
        node* ptr;
    };

    struct node
    {
        std::shared_ptr<T> data;
        std::atomic<int> internal_count;
        counted_node_ptr next;

        node(T const& data_):
            data(std::make_shared<T>(data_)),
            internal_count(0)
        {}
    };

    std::atomic<counted_node_ptr> head;

    void increase_head_count(counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;

        do
        {
```

```
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!head.compare_exchange_strong(old_counter,new_counter,
                                            std::memory_order_acquire,
                                            std::memory_order_relaxed));

        old_counter.external_count=new_counter.external_count;
    }

public:
    ~lock_free_stack()
    {
        while(pop());
    }

    void push(T const& data)
    {
        counted_node_ptr new_node;
        new_node.ptr=new node(data);
        new_node.external_count=1;
        new_node.ptr->next=head.load(std::memory_order_relaxed)
        while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
                                          std::memory_order_release,
                                          std::memory_order_relaxed));
    }
    std::shared_ptr<T> pop()
    {
        counted_node_ptr old_head=
            head.load(std::memory_order_relaxed);
        for(;;)
        {
            increase_head_count(old_head);
            node* const ptr=old_head.ptr;
            if(!ptr)
            {
                return std::shared_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next,
                                            std::memory_order_relaxed))
            {
                std::shared_ptr<T> res;
                res.swap(ptr->data);

                int const count_increase=old_head.external_count-2;

                if(ptr->internal_count.fetch_add(count_increase,
                       std::memory_order_release)==-count_increase)
                {
                    delete ptr;
                }

                return res;
            }
            else if(ptr->internal_count.fetch_add(-1,
                        std::memory_order_relaxed)==1)
            {
```

```
                ptr->internal_count.load(std::memory_order_acquire);
                delete ptr;
            }
        }
    }
};
```

That was quite a workout, but you got there in the end, and the stack is better for it. By using more relaxed operations in a carefully thought-through manner, the performance is improved without impacting the correctness. As you can see, the implementation of `pop()` is now 37 lines rather than the 8 lines of the equivalent `pop()` in the lock-based stack of listing 6.1 and the 7 lines of the basic lock-free stack without memory management in listing 7.2. As we move on to look at writing a lock-free queue, you'll see a similar pattern: lots of the complexity in lock-free code comes from managing memory.

7.2.6 Writing a thread-safe queue without locks

A queue offers a slightly different challenge to a stack, because the `push()` and `pop()` operations access different parts of the data structure in a queue, whereas they both access the same head node for a stack. Consequently, the synchronization needs are different. You need to ensure that changes made to one end are correctly visible to accesses at the other. However, the structure of `try_pop()` for the queue in listing 6.6 isn't actually that far off that of `pop()` for the simple lock-free stack in listing 7.2, so you can reasonably assume that the lock-free code won't be that dissimilar. Let's see how.

If you take listing 6.6 as a basis, you need two `node` pointers: one for the `head` of the list and one for the `tail`. You're going to be accessing these from multiple threads, so they'd better be atomic in order to allow you to do away with the corresponding mutexes. Let's start by making that small change and see where it gets you. The following listing shows the result.

Listing 7.13 A single-producer, single-consumer lock-free queue

```
template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        node* next;

        node():
            next(nullptr)
        {}
    };

    std::atomic<node*> head;
    std::atomic<node*> tail;
```

```
    node* pop_head()
    {
        node* const old_head=head.load();
        if(old_head==tail.load())          <-- 1
        {
            return nullptr;
        }
        head.store(old_head->next);
        return old_head;
    }
public:
    lock_free_queue():
        head(new node),tail(head.load())
    {}

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

    ~lock_free_queue()
    {
        while(node* const old_head=head.load())
        {
            head.store(old_head->next);
            delete old_head;
        }
    }
    std::shared_ptr<T> pop()
    {
        node* old_head=pop_head();
        if(!old_head)
        {
            return std::shared_ptr<T>();
        }

        std::shared_ptr<T> const res(old_head->data);     <-- 2
        delete old_head;
        return res;
    }

    void push(T new_value)
    {
        std::shared_ptr<T> new_data(std::make_shared<T>(new_value));
        node* p=new node;                                  <-- 3
        node* const old_tail=tail.load();    <-- 4
        old_tail->data.swap(new_data);                <-- 5
        old_tail->next=p;              <-- 6
        tail.store(p);      <-- 7
    }
};
```

At first glance, this doesn't seem too bad, and if there's only one thread calling `push()` at a time, and only one thread calling `pop()`, then this is actually perfectly fine. The important thing in that case is the happens-before relationship between the `push()` and the `pop()` to ensure that it's safe to retrieve the `data`. The store to `tail` ❼ synchronizes with the load from `tail` ❶; the store to the preceding node's `data` pointer ❺

is sequenced before the store to `tail`; and the load from `tail` is sequenced before the load from the `data` pointer ❷, so the store to `data` happens before the load, and everything is OK. This is therefore a perfectly serviceable *single-producer, single-consumer (SPSC)* queue.

The problems come when multiple threads call `push()` concurrently or multiple threads call `pop()` concurrently. Let's look at `push()` first. If you have two threads calling `push()` concurrently, they both allocate new nodes to be the new dummy node ❸, both read the *same* value for `tail` ❹, and consequently both update the data members of the same node when setting the `data` and `next` pointers ❺, ❻. This is a data race!

There are similar problems in `pop_head()`. If two threads call concurrently, they will both read the same value of `head`, and both then overwrite the old value with the same `next` pointer. Both threads will now think they've retrieved the same node—a recipe for disaster. Not only do you have to ensure that only one thread `pop()`s a given item, but you also need to ensure that other threads can safely access the `next` member of the node they read from `head`. This is exactly the problem you saw with `pop()` for your lock-free stack, so any of the solutions for that could be used here.

So if `pop()` is a "solved problem," what about `push()`? The problem here is that in order to get the required happens-before relationship between `push()` and `pop()`, you need to set the data items on the dummy node before you update `tail`. But this then means that concurrent calls to `push()` are racing over those very same data items, because they've read the same `tail` pointer.

Handling multiple threads in push()

One option is to add a dummy node between the real nodes. This way, the only part of the current `tail` node that needs updating is the `next` pointer, which could therefore be made atomic. If a thread manages to successfully change the `next` pointer from `nullptr` to its new node, then it has successfully added the pointer; otherwise, it would have to start again and reread the `tail`. This would then require a minor change to `pop()` in order to discard nodes with a null data pointer and loop again. The downside here is that every `pop()` call will typically have to remove two nodes, and there are twice as many memory allocations.

A second option is to make the `data` pointer atomic and set that with a call to compare/exchange. If the call succeeds, this is your tail node, and you can safely set the `next` pointer to your new node and then update `tail`. If the compare/exchange fails because another thread has stored the data, you loop around, reread `tail`, and start again. If the atomic operations on `std::shared_ptr<>` are lock-free, you're home free. If not, you need an alternative. One possibility is to have `pop()` return a `std::unique_ptr<>` (after all, it's the only reference to the object) and store the data as a plain pointer in the queue. This would allow you to store it as a `std::atomic<T*>`, which would then support the necessary `compare_exchange_strong()` call. If you're using the reference-counting scheme from listing 7.11 to handle multiple threads in `pop()`,`push()` now looks like this.

Listing 7.14 A (broken) first attempt at revising `push()`

```
void push(T new_value)
{
    std::unique_ptr<T> new_data(new T(new_value));
    counted_node_ptr new_next;
    new_next.ptr=new node;
    new_next.external_count=1;
    for(;;)
    {
        node* const old_tail=tail.load();     <-1
        T* old_data=nullptr;
        if(old_tail->data.compare_exchange_strong(
            old_data,new_data.get()))              <-2
        {
            old_tail->next=new_next;
            tail.store(new_next.ptr);     <-3
            new_data.release();
            break;
        }
    }
}
```

Using the reference-counting scheme avoids this particular race, but it's not the only race in `push()`. If you look at the revised version of `push()` in listing 7.14, you'll see a pattern you saw in the stack: load an atomic pointer ❶ and dereference that pointer ❷. In the meantime, another thread could update the pointer ❸, eventually leading to the node being deallocated (in `pop()`). If the node is deallocated before you dereference the pointer, you have undefined behavior. Ouch! It's tempting to add an external count in `tail` the same as you did for `head`, but each node already has an external count in the `next` pointer of the previous node in the queue. Having two external counts for the same node requires a modification to the reference-counting scheme to avoid deleting the node too early. You can address this by also counting the number of external counters inside the `node` structure and decreasing this number when each external counter is destroyed (as well as adding the corresponding external count to the internal count). If the internal count is zero and there are no external counters, you know the node can safely be deleted. This is a technique I first encountered through Joe Seigh's Atomic Ptr Plus Project.[5] The following listing shows how `push()` looks under this scheme.

Listing 7.15 Implementing `push()` for a lock-free queue with a reference-counted `tail`

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

    struct counted_node_ptr
    {
```

[5] Atomic Ptr Plus Project, http://atomic-ptr-plus.sourceforge.net/.

```
        int external_count;
        node* ptr;
    };

    std::atomic<counted_node_ptr> head;
    std::atomic<counted_node_ptr> tail;     <-1

    struct node_counter
    {
        unsigned internal_count:30;
        unsigned external_counters:2;       <-2
    };

    struct node
    {
        std::atomic<T*> data;
        std::atomic<node_counter> count;    <-3
        counted_node_ptr next;

        node()
        {
            node_counter new_count;
            new_count.internal_count=0;
            new_count.external_counters=2;    <-4
            count.store(new_count);

            next.ptr=nullptr;
            next.external_count=0;
        }
    };

public:
    void push(T new_value)
    {
        std::unique_ptr<T> new_data(new T(new_value));
        counted_node_ptr new_next;
        new_next.ptr=new node;
        new_next.external_count=1;
        counted_node_ptr old_tail=tail.load();

        for(;;)
        {
            increase_external_count(tail,old_tail);     <-5

            T* old_data=nullptr;
            if(old_tail.ptr->data.compare_exchange_strong(    <-6
               old_data,new_data.get()))
            {
                old_tail.ptr->next=new_next;
                old_tail=tail.exchange(new_next);
                free_external_counter(old_tail);     <-7
                new_data.release();
                break;
            }
            old_tail.ptr->release_ref();
        }
    }
};
```

In listing 7.15, `tail` is now an `atomic<counted_node_ptr>` the same as `head` ❶, and the `node` structure has a `count` member to replace the `internal_count` from before ❸. This `count` is a structure containing the `internal_count` and an additional `external_counters` member ❷. Note that you need only 2 bits for the `external_counters` because there are at most two such counters. By using a bit field for this and specifying `internal_count` as a 30-bit value, you keep the total counter size to 32 bits. This gives you plenty of scope for large internal count values while ensuring that the whole structure fits inside a machine word on 32-bit and 64-bit machines. It's important to update these counts together as a single entity in order to avoid race conditions, as you'll see shortly. Keeping the structure within a machine word makes it more likely that the atomic operations can be lock-free on many platforms.

The `node` is initialized with the `internal_count` set to zero and the `external_counters` set to 2 ❹ because every new node starts out referenced from `tail` and from the `next` pointer of the previous node once you've actually added it to the queue. `push()` itself is similar to listing 7.14, except that before you dereference the value loaded from `tail` in order to call to `compare_exchange_strong()` on the `data` member of the node ❻, you call a new function `increase_external_count()` to increase the count ❺, and then afterward you call `free_external_counter()` on the old tail value ❼.

With the `push()` side dealt with, let's take a look at `pop()`. This is shown in the following listing and blends the reference-counting logic from the `pop()` implementation in listing 7.11 with the queue-pop logic from listing 7.13.

Listing 7.16 Popping a node from a lock-free queue with a reference-counted tail

```
template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        void release_ref();
    };
public:
    std::unique_ptr<T> pop()
    {
        counted_node_ptr old_head=head.load(std::memory_order_relaxed);  ❶
        for(;;)
        {
            increase_external_count(head,old_head);     ❷
            node* const ptr=old_head.ptr;
            if(ptr==tail.load().ptr)
            {
                ptr->release_ref();          ❸
                return std::unique_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next))     ❹
            {
```

```
                T* const res=ptr->data.exchange(nullptr);
                free_external_counter(old_head);          <-- 5
                return std::unique_ptr<T>(res);
            }
            ptr->release_ref();     <-- 6
        }
    }
};
```

You prime the pump by loading the `old_head` value before you enter the loop ❶ and before you increase the external count on the loaded value ❷. If the `head` node is the same as the `tail` node, you can release the reference ❸ and return a null pointer because there's no data in the queue. If there is data, you want to try to claim it for yourself, and you do this with the call to `compare_exchange_strong()` ❹. As with the stack in listing 7.11, this compares the external count and pointer as a single entity; if either changes, you need to loop again, after releasing the reference ❻. If the exchange succeeded, you've claimed the data in the node as yours, so you can return that to the caller after you've released the external counter to the popped node ❺. Once both the external reference counts have been freed and the internal count has dropped to zero, the node itself can be deleted. The reference-counting functions that take care of all this are shown in listings 7.17, 7.18, and 7.19.

Listing 7.17 Releasing a node reference in a lock-free queue

```
template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        void release_ref()
        {
            node_counter old_counter=
                count.load(std::memory_order_relaxed);
            node_counter new_counter;
            do
            {
                new_counter=old_counter;
                --new_counter.internal_count;     <-- 1
            }
            while(!count.compare_exchange_strong(      <-- 2
                  old_counter,new_counter,
                  std::memory_order_acquire,std::memory_order_relaxed));

            if(!new_counter.internal_count &&
               !new_counter.external_counters)
            {
                delete this;     <-- 3
            }
        }
    };
};
```

The implementation of `node::release_ref()` is only slightly changed from the equivalent code in the implementation of `lock_free_stack::pop()` from listing 7.11. Whereas the code in listing 7.11 only has to handle a single external count, so you could just use a simple `fetch_sub`, the whole `count` structure now has to be updated atomically, even though you only want to modify the `internal_count` field ❶. This therefore requires a compare/exchange loop ❷. Once you've decremented the `internal_count`, if both the internal and external counts are now zero, this is the last reference, so you can delete the node ❸.

Listing 7.18 Obtaining a new reference to a node in a lock-free queue

```
template<typename T>
class lock_free_queue
{
private:
    static void increase_external_count(
        std::atomic<counted_node_ptr>& counter,
        counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;

        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!counter.compare_exchange_strong(
              old_counter,new_counter,
              std::memory_order_acquire,std::memory_order_relaxed));

        old_counter.external_count=new_counter.external_count;
    }
};
```

Listing 7.18 is the other side. This time, rather than releasing a reference, you're obtaining a fresh one and increasing the external count. `increase_external_count()` is similar to the `increase_head_count()` function from listing 7.12, except that it has been made into a `static` member function that takes the external counter to update as the first parameter rather than operating on a fixed counter.

Listing 7.19 Freeing an external counter to a node in a lock-free queue

```
template<typename T>
class lock_free_queue
{
private:
    static void free_external_counter(counted_node_ptr &old_node_ptr)
    {
        node* const ptr=old_node_ptr.ptr;
        int const count_increase=old_node_ptr.external_count-2;

        node_counter old_counter=
            ptr->count.load(std::memory_order_relaxed);
```

```
        node_counter new_counter;
        do
        {
            new_counter=old_counter;                          (1)
            --new_counter.external_counters;
            new_counter.internal_count+=count_increase;       (2)
        }
        while(!ptr->count.compare_exchange_strong(            (3)
              old_counter,new_counter,
              std::memory_order_acquire,std::memory_order_relaxed));

        if(!new_counter.internal_count &&
           !new_counter.external_counters)
        {
            delete ptr;                                       (4)
        }
    }
};
```

The counterpart to `increase_external_count()` is `free_external_counter()`. This is similar to the equivalent code from `lock_free_stack::pop()` in listing 7.11 but modified to handle the `external_counters` count. It updates the two counts using a single `compare_exchange_strong()` on the whole `count` structure (3), just as you did when decreasing the `internal_count` in `release_ref()`. The `internal_count` value is updated as in listing 7.11 (2), and the `external_counters` value is decreased by one (1). If *both* the values are now zero, there are no more references to the node, so it can be safely deleted (4). This has to be done as a single action (which therefore requires the compare/exchange loop) to avoid a race condition. If they're updated separately, two threads may both think they are the last one and thus both delete the node, resulting in undefined behavior.

Although this now works and is race-free, there's still a performance issue. Once one thread has started a `push()` operation by successfully completing the `compare_exchange_strong()` on `old_tail.ptr->data` ((5) from listing 7.15), no other thread can perform a `push()` operation. Any thread that tries will see the new value rather than `nullptr`, which will cause the `compare_exchange_strong()` call to fail and make that thread loop again. This is a busy wait, which consumes CPU cycles without achieving anything. Consequently, this is effectively a lock. The first `push()` call blocks other threads until it has completed, so *this code is no longer lock-free*. Not only that, but whereas the operating system can give priority to the thread that holds the lock on a mutex if there are blocked threads, it can't do so in this case, so the blocked threads will waste CPU cycles until the first thread is done. This calls for the next trick from the lock-free bag of tricks: the waiting thread can help the thread that's doing the `push()`.

MAKING THE QUEUE LOCK-FREE BY HELPING OUT ANOTHER THREAD

In order to restore the lock-free property of the code, you need to find a way for a waiting thread to make progress even if the thread doing the `push()` is stalled. One way to do this is to help the stalled thread by doing its work for it.

In this case, you know exactly what needs to be done: the `next` pointer on the tail node needs to be set to a new dummy node, and then the `tail` pointer itself must be updated. The thing about dummy nodes is that they're all equivalent, so it doesn't matter if you use the dummy node created by the thread that successfully pushed the data or the dummy node from one of the threads that's waiting to push. If you make the `next` pointer in a node atomic, you can then use `compare_exchange_strong()` to set the pointer. Once the `next` pointer is set, you can then use a `compare_exchange_weak()` loop to set the `tail` while ensuring that it's still referencing the same original node. If it isn't, someone else has updated it, and you can stop trying and loop again. This requires a minor change to `pop()` as well in order to load the `next` pointer; this is shown in the following listing.

Listing 7.20 `pop()` modified to allow helping on the `push()` side

```
template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        std::atomic<T*> data;
        std::atomic<node_counter> count;
        std::atomic<counted_node_ptr> next;     <-(1)
    };
public:
    std::unique_ptr<T> pop()
    {
        counted_node_ptr old_head=head.load(std::memory_order_relaxed);
        for(;;)
        {
            increase_external_count(head,old_head);
            node* const ptr=old_head.ptr;
            if(ptr==tail.load().ptr)
            {
                return std::unique_ptr<T>();
            }
            counted_node_ptr next=ptr->next.load();         <-(2)
            if(head.compare_exchange_strong(old_head,next))
            {
                T* const res=ptr->data.exchange(nullptr);
                free_external_counter(old_head);
                return std::unique_ptr<T>(res);
            }
            ptr->release_ref();
        }
    }
};
```

As I mentioned, the changes here are simple: the `next` pointer is now atomic (1), so the `load` at (2) is atomic. In this example, you're using the default `memory_order_seq_cst` ordering, so you could omit the explicit call to `load()` and rely on the load in the

implicit conversion to `counted_node_ptr`, but putting in the explicit call reminds you where to add the explicit memory ordering later.

The code for `push()` is more involved and is shown here.

Listing 7.21 A sample `push()` with helping for a lock-free queue

```
template<typename T>
class lock_free_queue
{
private:
    void set_new_tail(counted_node_ptr &old_tail,       <-1
                      counted_node_ptr const &new_tail)
    {
        node* const current_tail_ptr=old_tail.ptr;
        while(!tail.compare_exchange_weak(old_tail,new_tail) &&     <-2
              old_tail.ptr==current_tail_ptr);
        if(old_tail.ptr==current_tail_ptr)       <-3
            free_external_counter(old_tail);          <-4
        else
            current_tail_ptr->release_ref();     <-5
    }
public:
    void push(T new_value)
    {
        std::unique_ptr<T> new_data(new T(new_value));
        counted_node_ptr new_next;
        new_next.ptr=new node;
        new_next.external_count=1;
        counted_node_ptr old_tail=tail.load();

        for(;;)
        {
            increase_external_count(tail,old_tail);

            T* old_data=nullptr;
            if(old_tail.ptr->data.compare_exchange_strong(     <-6
                   old_data,new_data.get()))
            {
                counted_node_ptr old_next={0};
                if(!old_tail.ptr->next.compare_exchange_strong(      <-7
                       old_next,new_next))
                {
                    delete new_next.ptr;         <-8
                    new_next=old_next;     <-9
                }
                set_new_tail(old_tail, new_next);
                new_data.release();
                break;
            }
            else         <-10
            {
                counted_node_ptr old_next={0};
                if(old_tail.ptr->next.compare_exchange_strong(     <-11
                       old_next,new_next))
```

```
                {
                    old_next=new_next;      <-- (12)
                    new_next.ptr=new node;          <-- (13)
                }
                set_new_tail(old_tail, old_next);    <-- (14)
            }
        }
    }
};
```

This is similar to the original `push()` from listing 7.15, but there are a few crucial differences. If you *do* set the `data` pointer ❻, you need to handle the case where another thread has helped you, and there's now an `else` clause to do the helping ❿.

Having set the `data` pointer in the node ❻, this new version of `push()` updates the `next` pointer using `compare_exchange_strong()` ❼. You use `compare_exchange_strong()` to avoid looping. If the exchange fails, you know that another thread has already set the `next` pointer, so you don't need the new node you allocated at the beginning, and you can delete it ❽. You also want to use the `next` value that the other thread set for updating `tail` ❾.

The actual update of the `tail` pointer has been extracted into `set_new_tail()` ❶. This uses a `compare_exchange_weak()` loop ❷ to update the `tail`, because if other threads are trying to `push()` a new node, the `external_count` part may have changed, and you don't want to lose it. However, you also need to take care that you don't replace the value if another thread has successfully changed it already; otherwise, you may end up with loops in the queue, which would be a rather bad idea. Consequently, you need to ensure that the `ptr` part of the loaded value is the same if the compare/exchange fails. If the `ptr` is the same once the loop has exited ❸, then you must have successfully set the `tail`, so you need to free the old external counter ❹. If the `ptr` value is different, then another thread will have freed the counter, so you just need to release the single reference held by this thread ❺.

If the thread calling `push()` failed to set the `data` pointer this time through the loop, it can help the successful thread to complete the update. First off, you try to update the `next` pointer to the new node allocated on this thread ⓫. If this succeeds, you want to use the node you allocated as the new `tail` node ⓬, and you need to allocate another new node in anticipation of actually managing to push an item on the queue ⓭. You can then try to set the `tail` node by calling `set_new_tail` before looping around again ⓮.

You may have noticed that there are rather a lot of `new` and `delete` calls for such a small piece of code, because new nodes are allocated on `push()` and destroyed in `pop()`. The efficiency of the memory allocator therefore has a considerable impact on the performance of this code; a poor allocator can completely destroy the scalability properties of a lock-free container such as this. The selection and implementation of such allocators is beyond the scope of this book, but it's important to bear in mind that the only way to know that an allocator is better is to try it and measure the performance of the code before and after. Common techniques for optimizing memory

allocation include having a separate memory allocator on each thread and using free lists to recycle nodes rather than returning them to the allocator.

That's enough examples for now; instead, let's look at extracting some guidelines for writing lock-free data structures from the examples.

7.3 Guidelines for writing lock-free data structures

If you've followed through all the examples in this chapter, you'll appreciate the complexities involved in getting lock-free code right. If you're going to design your own data structures, it helps to have some guidelines to focus on. The general guidelines regarding concurrent data structures from the beginning of chapter 6 still apply, but you need more than that. I've pulled a few useful guidelines out from the examples, which you can then refer to when designing your own lock-free data structures.

7.3.1 Guideline: use std::memory_order_seq_cst for prototyping

`std::memory_order_seq_cst` is much easier to reason about than any other memory ordering because all such operations form a total order. In all the examples in this chapter, you've started with `std::memory_order_seq_cst` and only relaxed the memory-ordering constraints once the basic operations were working. In this sense, using other memory orderings is an *optimization*, and as such you need to avoid doing it prematurely. In general, you can only determine which operations can be relaxed when you can see the full set of code that can operate on the guts of the data structure. Attempting to do otherwise just makes your life harder. This is complicated by the fact that the code may work when tested but isn't guaranteed. Unless you have an algorithm checker that can systematically test all possible combinations of thread visibilities that are consistent with the specified ordering guarantees (and such things do exist), just running the code isn't enough.

7.3.2 Guideline: use a lock-free memory reclamation scheme

One of the biggest difficulties with lock-free code is managing memory. It's essential to avoid deleting objects when other threads might still have references to them, but you still want to delete the object as soon as possible in order to avoid excessive memory consumption. In this chapter you've seen three techniques for ensuring that memory can safely be reclaimed:

- Waiting until no threads are accessing the data structure and deleting all objects that are pending deletion
- Using hazard pointers to identify that a thread is accessing a particular object
- Reference counting the objects so that they aren't deleted until there are no outstanding references

In all cases the key idea is to use some method to keep track of how many threads are accessing a particular object and only delete each object when it's no longer referenced from anywhere. There are many other ways of reclaiming memory in lock-free

data structures. For example, this is the ideal scenario for using a garbage collector. It's much easier to write the algorithms if you know that the garbage collector will free the nodes when they're no longer used, but not before.

Another alternative is to recycle nodes and only free them completely when the data structure is destroyed. Because the nodes are reused, the memory never becomes invalid, so some of the difficulties in avoiding undefined behavior go away. The downside here is that another problem becomes more prevalent. This is the so-called *ABA problem*.

7.3.3 Guideline: watch out for the ABA problem

The ABA problem is something to be wary of in any compare/exchange–based algorithm. It goes like this:

1. Thread 1 reads an atomic variable `x` and finds it has value `A`.
2. Thread 1 performs some operation based on this value, such as dereferencing it (if it's a pointer) or doing a lookup or something.
3. Thread 1 is stalled by the operating system.
4. Another thread performs some operations on `x` that changes its value to `B`.
5. A thread then changes the data associated with the value `A` such that the value held by thread 1 is no longer valid. This may be as drastic as freeing the pointed-to memory or just changing an associated value.
6. A thread then changes `x` back to `A` based on this new data. If this is a pointer, it may be a new object that just happens to share the same address as the old one.
7. Thread 1 resumes and performs a compare/exchange on `x`, comparing against `A`. The compare/exchange succeeds (because the value is indeed `A`), but this is *the wrong* `A` *value*. The data originally read at step 2 is no longer valid, but thread 1 has no way of telling and will thus corrupt the data structure.

None of the algorithms presented here suffer from this problem, but it's easy to write lock-free algorithms that do. The most common way to avoid the problem is to include an ABA counter alongside the variable `x`. The compare/exchange operation is then done on the combined structure of `x` plus the counter as a single unit. Every time the value is replaced, the counter is incremented, so even if `x` has the same value, the compare/exchange will fail if another thread has modified `x`.

The ABA problem is particularly prevalent in algorithms that use free lists or otherwise recycle nodes rather than returning them to the allocator.

7.3.4 Guideline: identify busy-wait loops and help the other thread

In the final queue example you saw how a thread performing a push operation had to wait for another thread also performing a push to complete its operation before it could proceed. Left alone, this would have been a busy-wait loop, with the waiting thread wasting CPU time while failing to proceed. If you end up with a busy-wait loop, you effectively have a blocking operation and might as well use mutexes and locks. By

modifying the algorithm so that the waiting thread performs the incomplete steps if it's scheduled to run before the original thread completes the operation, you can remove the busy-wait and the operation is no longer blocking. In the queue example this required changing a data member to be an atomic variable rather than a nonatomic variable and using compare/exchange operations to set it, but in more complex data structures it might require more extensive changes.

7.4 Summary

Following from the lock-based data structures of chapter 6, this chapter has described simple implementations of various lock-free data structures, starting with a stack and a queue, as before. You saw how you must take care with the memory ordering on your atomic operations to ensure that there are no data races and that each thread sees a coherent view of the data structure. You also saw how memory management becomes much harder for lock-free data structures than lock-based ones and examined a couple of mechanisms for handling it. You also saw how to avoid creating wait loops by helping the thread you're waiting for to complete its operation.

Designing lock-free data structures is a difficult task, and it's easy to make mistakes, but such data structures have scalability properties that are important in some situations. Hopefully, by following through the examples in this chapter and reading the guidelines, you'll be better equipped to design your own lock-free data structure, implement one from a research paper, or find the bug in the one your former colleague wrote just before he left the company.

Wherever data is shared between threads, you need to think about the data structures used and how the data is synchronized between threads. By designing data structures for concurrency, you can encapsulate that responsibility in the data structure itself, so the rest of the code can focus on the task it's trying to perform *with* the data rather than the data synchronization. You'll see this in action in chapter 8 as we move on from concurrent data structures to concurrent code in general. Parallel algorithms use multiple threads to improve their performance, and the choice of concurrent data structure is crucial where the algorithms need their worker threads to share data.