Introduction

Before going beyond locks, let's look at some common data structures. Adding locks to a data structure makes it thread-safe. Of course, how these locks are added affects the data structure's accuracy and speed. So here's our task:

How do we add locks to a data structure to make it work properly? How do we apply locks to the data structure so that it is fast and can be accessed by several threads simultaneously?

The topic has been investigated for years, with thousands of research articles published on it. So we hope to provide you a good overview of the essential thinking.

Concurrent Counters

One of the most basic data structures is the counter. The structure is well-known, and the user interface is straightforward. A simple non-concurrent counter is shown in Figure 29.1.

Simple But Not Scalable

As you can see, the non-synchronized counter is a simple data structure to build, requiring only a few lines of code. So now we have a new problem to solve: how can we make this function thread-safe? Figure 29.2 depicts how we accomplish this.
This concurrent counter is straightforward and accurate. It merely adds a single lock acquired when calling a method that manipulates the data structure and is released upon returning from the call, a design pattern common to the simplest and most fundamental concurrent data structures. For example, it's akin to a data structure built with monitors in this way, where locks are automatically acquired and released when you call and return from object methods.

You now have a concurrent data structure that works. The issue you may be facing is one of performance. If your data structure is too slow, you'll need to do more than just add a single lock; as a result, the rest of the chapter will focus on such optimizations. On the other hand, if the data structure isn't too slow, you're done! If something basic will do, then there is no need to go overboard.
We construct a test in which each thread updates a single shared counter a fixed number of times; we then adjust the number of threads to determine the performance costs of the basic technique. Figure 29.5 displays the total time taken; each thread updates the counter one million times.
The performance of the synchronized counter scales poorly, as seen by the top line in the picture (labeled 'Precise'). This is because the million counter updates can be completed in a fraction of a second (approximately 0.03 seconds), but updating the counter one million times simultaneously results in a significant delay (nearly 5 seconds). It only gets worse with more threads.

Ideally, the threads should complete just as quickly on several processors as they do on a single processor. The process of achieving this goal is known as perfect scaling; even while more work is done, it is done in parallel, reducing the time it takes to accomplish the task.

Scalable Counting

Researchers have been researching how to develop more scalable counters for years. Even more impressive is the fact that scalable counters matter, as demonstrated by recent work in operating system performance research; without scalable counting, several Linux workloads suffer from substantial scalability issues on multicore platforms.

This issue has been tackled in a variety of ways. We'll go through one method called an approximation counter.
The approximation counter works by representing a single logical counter with a single global counter and several local physical counters, one for each CPU core. There are four local counters and one global counter on a computer with four CPUs. There are two more counters in addition to these: one for each local counter and one for the global counter.

The following is the core concept of approximation counting. When a thread on a specific core wants to increment the counter, it uses its local counter; access to this local counter is synchronized via the local lock. Threads across CPUs can update local counters without contention since each CPU has its own local counter, making counter updates scalable.

However, because the local counter must be updated on a regular basis (for example, to prevent a thread from accessing its value), it is incremented by gaining the global lock and incrementing it based on the value of the local counter. After then, the local counter is reset to zero.
A threshold S determines how often this local-to-global transmission occurs. The smaller S, the more the counter acts like the non-scalable counter above; the larger S, the more scalable the counter is, but the further off the global value from the real count the counter may be. To obtain an exact value, one could simply acquire all of the local locks as well as the global lock (in a specific order to avoid deadlock), but this is not scalable.

Let's look at an example to demonstrate this (Figure 29.3). The threshold S is set to 5 in this example, and threads on each of the four CPUs are updating their local counts L1... L4. With time decreasing, the global counter value (G) is also indicated in the trace. A local counter may be incremented at each time step; if the local value reaches the threshold S, it is transferred to the global counter, and the local counter is reset.
The performance of approximation counters with a threshold S of 1024 is shown in the lower line in Figure 29.5 (labeled 'Approximate,' on page 6). Again, the performance is excellent; updating the counter four million times on four processors takes about the same amount of time as updating it one million times on one CPU.

Figure 29.6 illustrates the significance of the threshold value S, with four threads on four CPUs, each incrementing the counter one million times. When S is low, performance is terrible (though the global count is always accurate); when S is high, performance is superb, but the global count lags (by at most the number of CPUs multiplied by S). Approximate counters permit this accuracy/performance trade-off.
Figure 29.4 shows a preliminary example of an estimated counter (page 5). To further understand how it works, read it or, better yet, perform some experiments yourself.

Concurrent Linked Lists

Next, we'll look at a more intricate structure called a linked list. Let's start from the beginning once more. We'll overlook some of the apparent routines that such a list would have for simplicity's sake and simply focus on concurrent insert; we'll leave it to the reader to consider lookup, delete, and so on. Figure 29.7 shows the code for this simple data structure.

As you can see in the code, entering the insert method simply obtains a lock and releases it when you exit. However, if malloc() fails (which is a rare occurrence), the code must also release the lock before failing the insert; in this instance, the code must also fail the insert.

This type of exceptional control flow is quite error-prone; a recent study of Linux kernel patches discovered that such rarely-taken code paths account for nearly 40% of all bugs (indeed, this observation sparked some of our own research, in which we removed all memory-failing paths from a Linux file system, resulting in a more robust system).

As a result, we face a challenge: How can we update the insert and lookup methods to keep them accurate when there are concurrent inserts without adding the unlock call if the insert fails?

In this scenario, the answer is yes. In particular, we may reorganize the code so that the lock and release only surround the critical region in the insert code, and the lookup function uses a common exit path. Because part of the insert does not need to be locked, the former works; provided malloc() is thread-safe, each thread can call it without fear of race situations or other concurrent issues. A lock is only required while changing the shared list. Figure 29.8 outlines these changes in more detail.

The lookup routine is a basic code change that allows you to exit the main search loop and return to a single path. Doing so once more reduces the number of lock acquire/release points in the code, lowering the risk of accidentally adding errors (such as forgetting to unlock before returning).

Scaling Linked Lists

Even though we have a basic concurrent linked list again, we are in a situation where it does not scale effectively. Hand-over-hand locking is one way that researchers have looked into to permit higher concurrency within a list (a.k.a. lock coupling).

The concept is straightforward. You add a lock to each node in the list instead of having a single lock for the entire list. The code then obtains the next node's lock before releasing the current node's lock as it traverses the list (which inspires the name hand-over-hand).

A hand-over-hand linked list makes sense conceptually; it allows for a high degree of parallelism in list operations. In practice, however, it's challenging to make such a structure faster than a basic single lock technique since the overheads of acquiring and releasing locks for each node of a list traversal are prohibitive. Furthermore, even with extensive lists and a large number of threads, permitting several concurrent traversals is unlikely to be faster than grabbing a single lock, performing an operation, and then releasing it. Something like a hybrid (where you obtain a fresh lock every so many nodes) might be worth investigating.

Concurrent Queues

As you may already know, there is always a convenient way to create a concurrent data structure: add a prominent lock. We'll forgo that approach for a queue, provided you can figure it out.

Instead, we'll take a look at Michael and Scott's significantly more concurrent queue. Figure 29.9 on the following page shows the data structures and code for this queue.

If you look closely at this code, you'll note two locks, one for the queue's head and the tail. The purpose of these two locks is to allow enqueue and dequeue activities to run concurrently. In most cases, the enqueue function will only access the tail lock, while the dequeue method will only access the head lock.

Michael and Scott utilize one approach to introduce a dummy node (allocated in the queue startup code) that separates head and tail operations. Then read the code, or even better, type it in, run it, and measure it to have a thorough understanding of how it works.

In multi-threaded applications, queues are frequently employed. However, the form of queue utilized here (with mere locks) frequently fails to match the requirements of such systems. The following chapter on condition variables will focus on a more fully developed bounded queue that allows a thread to wait if the queue is either empty or very filled. Keep an eye out for it!

Concurrent Hash Table

The hash table, a simple and extensively applicable concurrent data structure, brings our topic to a close. We'll concentrate on a simple hash table that doesn't resize; resizing requires a bit more work, which we'll leave as an exercise for the reader (sorry!).

This concurrent hash table (Figure 29.10) is simple to construct, using the concurrent lists we created earlier and performs admirably. Its high performance is because it uses a lock per hash bucket rather than a single lock for the entire structure (each represented by a list). This allows for multiple operations to run at the same time.

Figure 29.11 depicts the hash table's performance when many updates are made simultaneously (from 10,000 to 50,000 concurrent updates from each of four threads, on the same iMac with four CPUs). A linked list's performance is also displayed for comparison's purposes (with a single lock). This simple concurrent hash table scales beautifully, as shown in the graph; the linked list, on the other hand, does not.

Summary

We've gone over various concurrent data structures, from counters to lists and queues, and finally, the ever-popular hash table. Along the way, a few key lessons have been learned:

- Be cautious while acquiring and releasing locks during control flow changes.
- Increasing concurrency does not continuously improve performance.
- Performance issues should be addressed as soon as they appear.

This last point about avoiding premature optimization is crucial for any performance-minded developer; there's no use in making something faster if it doesn't help the program's overall performance.