Chapter 29: Lock-based Concurrent Data Structures

When given a particular data structure, how should we add locks to it, in order to make it work correctly?

**29.1 Concurrent Counters**

One of the simplest data structures is a counter. It is a structure that is commonly used and has a simple interface.

Graphical user interface, text, application

Description automatically generated

**Simple But Not Scalable**

How to make this code **thread safe**?

Text

Description automatically generated

We simply add a lock and acquire it when calling a routine that manipulates the data structure and release it when returning from call.

At this point, you have a working concurrent data structure. The problem you might have is performance. If your data structure is too slow, you’ll have to do more than just add a single lock. The performance of such code is shown as follows:

Chart, diagram, line chart

Description automatically generated

As we have more threads. It causes a massive slowdown. Ideally, we would like to see the threads complete just as quickly on multiple processors as the single thread does on one. This is **perfect scaling**.

**Scalable Counting**

**Approximate counter** works by presenting a single logical counter via numerous local physical counters, one per CPU core, as well as a global counter. Specifically, on a machine with four CPUs, there are four local counters and one global one. In addition to these counters, there are also locks: one for each local counter, and one for the global counter.

The idea is that when each thread wishes to increment the counter, it increases the local counter. Because each CPU has its own local counter, threads across CPUs can update local counters without contention, and thus updates to the counter are scalable.

To keep the global counter up to date, the local values are periodically transferred to the global counter by acquiring the global lock and incrementing it by the local counter’s value. The local counter then be set to 0.

Table

Description automatically generated

The performance is given as follows:

The performance is really good (figure 29.5 on the circle dots). With four threads incrementing the counter 1 million times, if S is low, the performance is poor. If S is high, the performance is excellent, but the global count lags.

A picture containing chart

Description automatically generated

This accuracy/performance tradeoff is what approximate counters enable.

**29.2 Concurrent Linked Lists**

If we malloc fail, then we must release the lock. The clean version of concurrent linked list is shown as follows:

Text

Description automatically generated

**Scaling Linked Lists**

Though we again have a basic concurrent linked list, once again we are in a situation where it does not scale particularly well. One technique that researchers have explored to enable more concurrency within a list is something called **hand-over-hand locking.**

The idea is pretty simple. Instead of having a single lock for the entire list, you instead add a lock per node of the list. When traversing the list, the code first grabs the next node’s lock and then releases the current node’s lock (which inspires the name hand-over-hand).

This approach makes sense, but it is way slower than the single lock approach because of overhead.

**29.3 Concurrent Queues**

Text, letter

Description automatically generated

In this approach, we have two locks, one for the head of the queue, one for the tail. The goal of these two locks is to ensure concurrency.

One trick used by Michael and Scott is to add a dummy node (allocated in the queue initialization code); this dummy enables the separation of head and tail operations.

**29.4 Concurrent Hash Table**

We’ll focus on a simple hash table that does not resize.

Text

Description automatically generated

This concurrent hash table (Figure 29.10) is straightforward, is built using the concurrent lists we developed earlier, and works incredibly well. The reason for its good performance is that instead of having a single lock for the entire structure, it uses a lock per hash bucket.

The performance of this approach is excellent:

Chart, line chart

Description automatically generated