COS 226 - Semester Test 2 Notes

Regan Koopmans

October 22, 2016

Contents

1 Chapter 7 - Spin Locks and Contension

1.1 Test-and-Set Locks

The testAndSet() method, that has a consensus number of 2, is a means by which we can construct a method of creating wait-free synchronization locks.

Suppose that a number of threads share a boolean field, indicating whether a lock is free or not. We can create a lock method, that maintains mutual exclusion by doing the following.

```
public void lock()
{
  while (state.getAndSet(true)) {};     // if this returns false we know
                                        // acquisition was successful
}
```

A thread that exits the critical section can simply set this field to false, indicating that the critical section is once again available. Unfortunately, for every thread, for every iteration, is writing to this shared value. This means that there are many unnecessary updates to this value, and hence much unneeded traffic.

1.2 Test-Test-and-Set Lock

Unfortunately, the test-and-set lock performs rather poorly. This is result of the fact that a spinning thread encounters a cache miss almost every time. The thread must then use the shared bus simply to update to the value it had previously. All of this cache coherence traffic, which is unnecessary in our system. The TTAS lock method appears as follows:

By doing this we save most of our overwrites of the shared variable, and therefore reduces a massive cost that our system would have to incur otherwise.

1.3 Exponential Backoff

Exponential Backoff is a technique that tries to minimize contention in a lock system, by enforcing that threads wait an exponentially increasing amount of time while contension is high.

If a thread finds that the lock is free for acquisition, and tries to obtain it, and find that another thread acquired it first, the initial thread considers this to be a environmnent of high contension. The thread will then choose a a

1.3.1 Why the random number?

Threads choose and initial, random waiting time for which they then raise to powers later on. This is done because if two or more threads always have equal waiting times, they will continue to collide and this will force waiting times to become unreasonably large. This problem is known as **lock-step**.

1.3.2 Disadvantages

A system using exponential backoff suffers from two major issues:

- Critical Section Underutilization
- Cache-coherence traffic

1.4 Queue Locks

1.4.1 Array Based Locks

In the Array Lock (or "ALock"), an array is defined, symbolizing the maximum capacity of the system. Each thread has a *thread-local* variable called `mySlot` which says which slot in the array they belong to. A global variable tail is maintained, which increments and constantly wraps around the array as time progresses.

The array is a boolean array, and the first element is always true, the rest being false. A thread will request the lock, increment tail, and then wait for their slot to be made true, indicating that those behind them have successfully exited their critical sections.

Contension may still occur, however, because of a phenomenon called *false sharing*

1.4.2 The CLH Queue Lock

The ALock is not space efficient. It requires a known bound n on the maximum amount of threads, an allocates an array of that size. Thus even if only two threads are competing for the lock at any given time, the ALock must make provision for all of them.

The CLH Queue Lock records each thread's status in a Qnode object, which has a Boolean locked field. If this field is true, then the corresponding thread has either acquired the lock, or is waiting to acquire the lock.

Each thread will wait on their predecessor's locked value. Threads do this by keeping a reference to their predecessor, and this creates a *virtual* (implicit) linked list.

1.4.3 The MCS Queue Lock

This queue is similar to CLH, but differs in that its list is explicit. Each thread depends on the previous thread to set their field to true, such that they can enter the critical section.

1.4.4 Queue Locks With Timeouts

The problem with the queue locks we have seen so far is that if one of the threads in the chain were to terminate, or be blocked for some reason, our entire system has no way of progressing. This is why we need a timeout, so to ensure that the system can accomodate failing threads.

2 Chapter 8 - Monitors and Blocking Conditions

2.1 Monitor Locks and Conditions

Monitors are objects that allow blocking in a system.

2.1.1 Conditions

A Condition variable is a shared register that allows threads to sleep until a certain property of the system becomes true. Condition variables typically have an await() method (which allows a thread to block on that condition), and a signal/signalAll(). Alternatively we can use the awaitUntil() to add a time constraint. Conditions are associated with locks in Java.

Threads that are waking up from waiting need to test critical values once again.

Conditions awaitings are typicall in a while loop, because we cannot assume that by the time a thread has woken up that requirements of the enviroment still hold.

2.1.2 The Lost Wake-Up Problem

Just as locks are inherently vulnerable to deadlock, Condition objects can be vulnerable to lost wakeups. This occurs when threads are blocking while the condition they are blocking for has become true, but they are not aware of it yet. This means that they are blocking but do not need to be.

```
public void enq(T x)
{
 lock.lock();
 try
 {
   while (count == items.length)
      isFull.await();
   items[tail] = x;
   ++count;
   if (count == 1)
     isEmpty.signalAll();         // If this was only signal, we may have lost wakeups
 }
 finally
 {
  lock.unlock();
 }
}
```

To work around this problem, just signal everyone. Also, if the requirements are restrictive enough, we can add the waitUntil time specification.

2.2 Readers-Writers Locks

In a readers-writers environment, we have an interesting dichotemy that divides threads. Reads do not need to synchronize with one another, but writers definitely do.

2.2.1 Simple Readers-Writers Lock

This is an example of a simple readers-writers lock.

```
public class SimpleReadWriteLock implements ReadWriteLock
{
 int readers;               // Number of readers
```

```
 boolean writer;            // Is there a writer?
 Lock lock;                 // Limites access to reader and writer fields
 Condition condition;
 Lock readLock,writeLock;

 public SimpleReadWriteLock()
 {
   writer = false;
   readers = 0;
   lock = new ReentrantLock();
   readLock = new ReadLock();
   writeLock = new WriteLock();
   condition = lock.newCondition();
 }

 public Lock readLock()
 {
  return readLock;
 }

 public Lock writeLock()
 {
  return writeLock;
 }
}

class ReadLock implements Lock
{
  public void lock()
  {
   lock.lock();
   try
   {
     while (writer)
     {
       condition.await();
     }
     readers++;
   }
   finally
```

```
   {
     lock.unlock();
   }
  }

  public void unlock()
  {
    lock.lock();
    try
    {
      readers--;
      if (readers == 0)
        condition.signalAll();
    }
    finally
    {
      lock.unlock();
    }
  }
}

class WriteLock implements Lock
{
 public void lock()
 {
   lock.lock();
   try
   {
     while (readers > 0 || writer)
       condition.await():

     writer = true;
   }
   finally
   {
     lock.unlock();
   }
 }
```

```
 public void unlock()
 {
   lock.lock();
   try
   {
     writer = false;
     condition.signalAll();
   }
   finally
   {
     lock.unlock();
   }
 }
}
```

2.2.2 Fair Readers-Writers Lock

The lock that was defined previously is not fair. As long as there are readers, the writer has to block. And therefore, if there is a reasonable stream of frequent readers, the writer may never get the opportunity to actualize their writing. Thus we do not have fairness.

To make this fair we only need to make a small adjustment.

2.3 Semaphores

Semaphores are a synchronisation primitive. Semaphores are a generalization of mutual exclusion locks. Semaphores have a capacity, which decreases when a thread successfully acquires. If a thread acquires the semaphore, and its value is less than 0, it will block, and will wait to be signalled from processing threads.

3 Chapter 9 - Linked Lists : The Role of Locking

In chapter 7 we saw to build scalable spin locks that provide mutual exclusion efficiently, even when they are heavily used. We might think that it is now a simple matter to construct scalable concurrent data structures : take a sequentiall implementation of the class, add a scalable lock field, and ensure that each method call acquired and releases the lock. We refer to this as **coarse grained synchronization**, because we are synchronizing on the extremely macro level.

3.1 List-Based Sets

The entirety of chapter 9 is based on a single, foundational concept: A set is implemented as a linked list of nodes.

The list has two kinds of nodes. In addition to **regular nodes** that hold items in the set, we use two **sentinel nodes**, namely head and tail for the first and last elements.

3.2 Concurrent Reasoning

Reasoning about concurrent data structures may seem impossibly difficult, but it is a skill that can be learned. Often the to understanding a concurrent data structure is to understand its *invariants*: **properties that should always hold**. We can show that a property is invariant by showing that:

1. The propery holds when the object is created.
2. Once the property holds, there is no action that threads can make to remove it from that state.

When reasoning about concurrent object implementations, it is important to understand the destinction between an **object's abstract value** (here, a set of items), and its **concrete representation** (here, a list of nodes).

3.3 Coarse-Grained Synchronization

3.4 Fine-Grained Synchronization

In this form of synchronization, instead of using a single lock to synchronize every access to the object (as in coarse graineed synchronization), we split the split the object into **independently synchronized subcomponents**. This ensures that method calls interfere only when trying to access the same component at the same time.

3.5 Optimistic Synchronization

In this form of synchronization

Although fine-grained locking is an improvement over single, coarse-grained lock, it still imposes a potentially long sequence of lock acquisitions and and releases.

3.6 Lazy Synchronization

The OptimisticList implementation works best if the cost of traversing the list twice without locking is significantly less than the cost of traversing the list once with locking.

3.7 Non-Blocking Synchonization

In this synchronization we remove locks entirely, relying rather on built in atomic operations.