EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 8: September 20

Lecturer: Vijay Garg *Scribe: Liheng Ding*

8.1 Introduction

This lecture covers following topics:

1. Locks with Get-and-Set Operation
2. Queue Locks

8.2 Locks with Get-and-Set Operation

Although locks for mutual exclusion can be built using simple read and write instructions, any such algorithm requires as many memory locations as the number of threads. By using instructions with higher atomicity, it is much easier to build locks. For example, the *getAndSet* operation (also called *testAndSet*) allows us to build a lock as shown in Alogrithm 1.

Algorithm 1 Building Locks Using GetAndSet.

```
import java.util.concurrent.atomic.*;
public class GetAndSet implements MyLock {
    AtomicBoolean isOccupied = new AtomicBoolean(false);
    public void lock() {
        while (isOccupied.getAndSet(true))
            Thread.yield();
            // skip();
        }
    }
    public void unlock() {
        isOccupied.set(false);
    }
}
```

This algorithm satisfies the mutual exclusion and progress property. However, it does not satisfy starvation freedom. Besides, keeping invoking *getAndSet* is not efficient enough since this method will use a shared bus to get access to *isOccupied*. So an alternative implementation is shown in Algorithm 2.

In this implementation, a thread first checks if the lock is available using the *get* operation. It calls the *getAndSet* operation only when it finds the critical section available. If it succeeds in *getAndSet*, then it enters the critical section; otherwise, it goes back to spinning on the *get* operation.

However, high bus contention still happens when all threads get that *isOccupied.get()* is *false* and try to set it to *true* by using *getAndSet()*. So a better implementation using *Backoff* is shown in Algorithm 3. The thread fails to set *getAndSet()* will back off for a certain period of time.

Algorithm 2 Building Locks Using GetAndGetAndSet.

```
import java.util.concurrent.atomic.*;
public class GetAndGetAndSet implements MyLock {
        AtomicBoolean isOccupied = new AtomicBoolean(false);
        public void lock() {
                while (true){
                        while (isOccupied.get()) {
                        }
                        if (! isOccupied.getAndSet(true)) return;
                }
        }
        public void unlock() {
                isOccupied.set(false);
        }
}
```

Algorithm 3 Building Locks Using Backoff.

```
import java.util.concurrent.atomic.*;
public class GetAndGetAndSet implements MyLock {
        AtomicBoolean isOccupied = new AtomicBoolean(false);
        public void lock() {
                while (true){
                        while (isOccupied.get()) {
                        }
                        if (! isOccupied.getAndSet(true)) return;
                        else {
                                int timeToSleep = calculateDuration();
                                Thread.sleep(timeToSleep);
                        }
                }
        }
        public void unlock() {
                isOccupied.set(false);
        }
}
```

8.3 Queue Locks

Previous approaches to solve mutual exclusion require threads to spin on the shared memory location. In this section, we present alternate methods to avoid spinning on the same memory location. All methods maintain a queue of threads waiting to enter the critical section. Anderson's lock uses a fixed size array, CLH lock uses an implicit linked list and MCS lock uses an explicit linked list for the queue. One of the key challenges in designing these algorithms is that we cannot use locks to update the queue.

8.3.1 Anderson's Lock

Anderson's lock uses a circular array **Available** of size n which is at least as big as the number of threads that may be contending for the critical section. The array is circular so that the index i in the array is always a value in the range $0..n-1$ and is incremented modulo n. Different threads waiting for the critical section spin on the different slots in this array thus avoiding the problem of multiple threads spinning on the same variable.

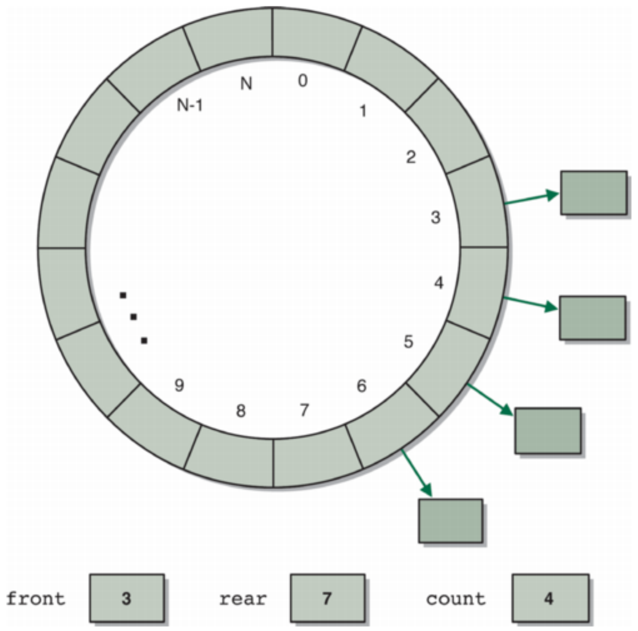

Figure 8.1: Circular Array

Note that the above description assumes that each slot is big enough so that adjacent slots do not share a cache line. Hence even though we just need a single bit to store **Available[i]**, it is important to keep it big enough by padding to avoid the problem of *false sharing*. Also note that since Anderson's lock assigns slots to threads in the FCFS manner, it guarantees fairness and therefore freedom from starvation.

A problem with Anderson lock is that it requires a separate array of size n for each lock. Hence, a system that uses m locks shared among n threads will use up $O(nm)$ space.

Algorithm 4 pseudo code for Anderson Lock

```
public class AndersonLock {
    AtomicInteger tailSlot = new AtomicInteger (0);
    boolean [ ] Avaliable ;
    ThreadLocal <Integer>mySlot; // initialize to 0

    public AndersonLock (int n) { // constructor
    } // all Available false except Available[0]
    public void lock() {
        mySlot.set(tailSlot.getAndIncrement() % n);
        spinUntil (Available[mySlot]);
    }
    public void unlock() {
        Available[mySlot.get()] = false;
        Available[(mySlot.get()+1) % n] = true;
    }
}
```

8.3.2 CLH Queue Lock

We can reduce the memory consumption in Anderson's algorithm by using a dynamic linked list instead of a fixed size array. Any thread that wants to get lock inserts a node in the linked list. The insertion in the linked list must be atomic. The sub-algorithm is shown in Algorithm 5 and full algorithm can be found at [1]. It is important to note that if the thread exists the CS wants to enter the CS again, it cannot reuse its own node because the next thread may still spinning on that node's field. Therefore, it uses the predecessor in the *unlock* function.

The linked list is virtual. The thread that is spinning has the variable *pred* that points to the predecessor node in the linked list.

Algorithm 5 pseudo code for CLH Queue Lock

```
class Node {
    boolean locked;
}
AtomicReference<Node>tailNode;
ThreadLocal<Node>myNode;
ThreadLocal<Node>pred;
lock() {
    myNode.locked = true;
    pred = tailNode.getAndSet(myNode);
    while(pred.locked) { noops; }
}
unlock() {
    myNode.locked = false;
    myNode = pred; // reuse pred node for future
}
```

8.3.3 MCS Queue Lock

In many architectures, each core may have local memory such that access to its local memory is fast but access to the local memory of another core is slower. In such architectures, CLH algorithm results in threads spinning on remote locations. MCS avoids the remote spinning. Same as CLH, MCS maintain a queue. However, it is an explicit linked list.

The tricky part of MCS is in the *unlock()* function. When a thread *p* exits the CS, it checks if there is any node linked next to its node. If there exits such node, then it makes the *locked* false and removes its node from the linked list. When it finds no node linked next to it, there can be two cases for this. First, no thread has been inserted yet. Therefore, we can find that *tailNode* still points to *myNode*. It can be simply reset to null. In the second case, thread *p* waits until the next is not null. Then it can set the locked field for that node to be false as usual. This is shown in Algorithm 6.

Algorithm 6 pseudo code for MCS Queue Lock

```
class QNode {
    boolean locked; // init true
    QNode next; // init null
}
lock() {
    QNode pred = tailNode.getAndSet(myNode);
    pred.next = myNode;
    while(myNode.locked) { noops; }
}
unlock() {
    if (myNode.next == null) {
        if ( tailNode.compareAndSet(myNode, null)) return;
        while (myNode.next == null) { noops; }
    }
    myNode.next.locked = false;
    myNode.next = null;
}
```

References

[1] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore