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

Lecture 1: September 20

Lecturer: Vijay Garg *Scribe: Shuang Song*

1.1 Introduction

In this lecture, we first go over the GetAndSet operations and how do use them to achieve mutual exclusion. Additionally, we introduce the Queue locks and discuss the pros and cons of each queue lock. Outline is shown as follows:

1. Building locks using GetAndSet
2. Building locks using GetAndGetAndSet
3. Using backoff
4. Anderson's lock
5. CLH queue lock
6. MCS queue lock

1.2 Locks with Get-and-Set Operation

Locks for mutual exclusion can be built using simple rw instructions, which requires n memory locations for n threads. It is easier to use instructions with higher atomicity, such as the GetAndSet operation. The example lock is shown in Algorithm 1.

Algorithm 1 Building Locks Using GetAndSet.

```
1:  import java.util.concurrent.atomic.*;
2:  public class GetAndSet implements MyLock {
3:      AtomicBoolean isOccupied = new AtomicBoolean(false);
4:      public void lock() {
5:          while (isOccupied.getAndSet(true)){          ▷ isoccupied is true, then busy wait
6:              Thread.yield();                                                    ▷ skip();
7:          }          ▷ isoccupied is false, means CS is empty, thread goes to CS and set isoccupied back to true
8:      }
9:      public void unlock() {
10:         isOccupied.set(false);
11:     }
12: }
```

The problem of this approach can be summarized into three points, which are busy-wait, starvation freedom, and expensive getAndSet() operation. Keep checking by using getAndSet operation can cause the high contention for the

Algorithm 2 Building Locks Using GetAndGetAndSet.

```
1:  import java.util.concurrent.atomic.*;
2:  public class GetAndGetAndSet implements MyLock {
3:      AtomicBoolean isOccupied = new AtomicBoolean(false);
4:      public void lock() {
5:          while (true){
6:              while (isOccupied.get()) {
7:              }
8:              if (! isOccupied.getAndSet(true)) return;
9:          }
10:     }
11:     public void unlock() {
12:         isOccupied.set(false);
13:     }
14: }
```

shared bus. The goal here is to optimize the algorithm and alleviate the bus contention. The algorithm to achieve this goal is shown in Algorithm 2.

As can be seen, *isOccupied.get()* is a read instruction, which is served in cache mostly. Checking *isOccupied.get()* before using getAndSet reduces contention of the bus. Therefore, compared to Algorithm 1, the second implementation results in faster accesses to the critical session.

However, if all these threads now get that *isOccupied.get()* is *false* and try to set it to true by using getAndSet, only one of them can succeeds but all of them ending up causing high bus contention again. *Backoff*, illustrated in Algorithm 3, is an useful idea to solve this issue. The thread fails to set getAndSet, it backs off for a certain period of time.

Algorithm 3 Building Locks Using GetAndGetAndSet.

```
1:  import java.util.concurrent.atomic.*;
2:  public class GetAndGetAndSet implements MyLock {
3:      AtomicBoolean isOccupied = new AtomicBoolean(false);
4:      public void lock() {
5:          while (true){
6:              while (isOccupied.get()) {
7:              }
8:              if (! isOccupied.getAndSet(true)) return;
9:              else {
10:                 int timeToSleep = calculateDuration();
11:                 Thread.sleep(timeToSleep);
12:             }
13:         }
14:     }
15:     public void unlock() {
16:         isOccupied.set(false);
17:     }
18: }
```

1.3 Queue Locks

Previous approaches require threads to spin on the shared memory location to achieve mutual exclusion. To avoid spinnings, we discuss three methods here, which are Anderson's lock (fixed size array), CLH lock (implicit linked list) and MCS lock (explicit linked list).

Algorithm 4 pseudo code for Anderson Lock

```
1:  public class AndersonLock {
2:      AtomicInteger tailSlot = new AtomicInteger (0);
3:      boolean [ ] Avaliable ;
4:      ThreadLocal <Integer>mySlot;                          ▷ initialize to 0

5:      public AndersonLock (int n) {                         ▷ constructor
6:      }                                                     ▷ all Available false except Available[0]
7:      public void lock() {
8:          mySlot.set(tailSlot.getAndIncrement() % n);
9:          spinUntil (Available[mySlot]);
10:     }
11:     public void unlock() {
12:         Available[mySlot.get()] = false;
13:         Available[(mySlot.get()+1) % n] = true;
14:     }
15: }
```

1.3.1 Anderson's Lock

Anderson's lock uses a circular array *Available* of size n (shown in Figure 1.1), which is as big as the number of threads. Different threads waiting for the critical section spin on their own slots in the array, so it avoids the problem of multiple threads spinning on the same variable. The *TailSlot* is an atomic integer that is pointing to the next available slot in the array. Any thread acquires the lock will read the value of *tailSlot* in its local variable *mySlot* and calculates the next available slot by using atomic operation *getAndIncrement()*. It then spins on the *Available[mySlot]* until the slot becomes available. In this algorithm, only one thread is affected when a thread leaves the critical section, and all the spinnings result in local cache accesses. Anderson's lock has no starvation problem. It is shown in Algorithm 4.

The problem with Anderson lock is that it requires a separate array for each lock. Like, a system that uses m locks shared among n threads will use up to O(mn) space.

1.3.2 CLH Queue Lock

To reduce the space consumption in Anderson's lock, CLH uses a dynamic linked list instead of 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

```
1:  class Node {
2:      boolean locked;
3:  }
4:  AtomicReference<Node>tailNode;
5:  ThreadLocal<Node>myNode;
6:  ThreadLocal<Node>pred;
7:  lock() {
8:      myNode.locked = true;
9:      pred = tailNode.getAndSet(myNode);
10:     while(pred.locked) { noops; }
11: }
12: unlock() {
13:     myNode.locked = false;
14:     myNode = pred;                          ▷ reuse pred node for future
15: }
```

Algorithm 6 pseudo code for MCS Queue Lock

```
1:  class QNode {
2:      boolean locked;                         ▷ init true
3:      QNode next;                             ▷ init null
4:  }
5:  lock() {
6:      QNode pred = tailNode.getAndSet(myNode);
7:      pred.next = myNode;
8:      while(myNode.locked) { noops; }
9:  }
10: unlock() {
11:     if (myNode.next == null) {
12:         if ( tailNode.compareAndSet(myNode, null)) return;
13:         while (myNode.next == null) { noops; }
14:     }
15:     myNode.next.locked = false;
16:     myNode.next = null;
17: }
```

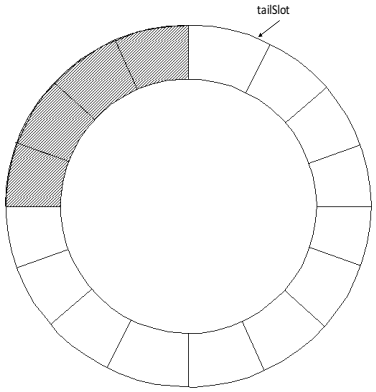

Figure 1.1: Circular Array

1.3.3 MCS Queue Lock

It is known that access to the core's local memory is faster than accessing to the local memory of another core. CLH algorithm results in thread spinning on remote locations in such architecture. 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.

References

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