MANCHESTER 1824

The University of Manchester

Chip Multiprocessors COMP35112

Lecture 8 - Lock-Free Data Structures

Graham Riley

Today's Lecture

- Last lecture told you about instructions introduced into the instruction set of shared-memory multiprocessors to make it possible to implement locks efficiently
- So can we use these instructions **instead** of using locks?
- Yes – we can. But it is far from easy!
- "Lock-free" essentially means not having multiple instructions executed atomically.
  - This helps to make sure algorithms can keep making progress
- I will illustrate for an unbounded queue

Unbounded Queue - Locking

- First we will do it using (two) locks
- The advantage in using separate locks for calls to enQ() and deQ() is that (in general) they don't need protection from each other …
- Use a linked list of Nodes
- Have a dummy node, even in the empty queue – it avoids making a special case of taking the last item or adding the first

Class Node

```
class Node {
private Object value ;
private Node next ;
public Node(Object newValue) {
    value = newValue ;
    next = null ;
}
```

Unbounded Queue

```
class UbQueue { // implemented using locks
private Node head, tail ;
private ReentrantLock enLock, deLock ;
public UbQueue() {
   head = tail = new Node(null) ; // sentinel
   enLock = new ReentrantLock() ;
   deLock = new ReentrantLock() ;
}

public void enQ(Object item) {
// slide 6
}
public Object deQ() throws EmptyException {
// slide 7
}  }
```

Method enQ()

```
public enQ(Object item) {
    enLock.lock() ;
    try {
        Node n = new Node(item) ;
        tail.next = n ;
        tail = n ;
     } finally {
         enLock.unlock() ;
}}
```

Method deQ()

```
public Object deQ() throws EmptyException {
    Object result ;
    deLock.lock() ;
    try {
        if (head.next == null)
                throw new EmptyException() ;
        head = head.next ;
        result = head.value ;
    } finally { deLock.unlock() ; }
    return result ;
}
```

How to do this without locks

- We need to use some special features:
  - `java.util.concurrent.atomic.AtomicReference<V>`
  - `boolean compareAndSet(V expected, V update)`
- The class allows the creation of references which can be updated atomically – not necessarily easy if a reference is (say) 64 bits in a 32-bit ISA
- Compare_and_set is one of those RMW instructions we encountered in the last lecture that help to implement locks. If the value is the same as expected, the update is written atomically and **true** is returned. Otherwise nothing is written and **false** is returned.

Modified Class Node

```
class Node {
private Object value ;
private AtomicReference<Node> next ;
public Node(Object newValue) {
    value = newValue ;
    next =
     new AtomicReference<Node>(null) ;
    }
}
```

Lock-free Unbounded Queue

```
class UbQueue {
private AtomicReference<Node> head, tail ;
public UbQueue() {
   head.set(new Node(null)) ; // sentinel
   tail.set(head) ;
}

public void enQ(Object item) {
// slide 11/12
}
public Object deQ() throws EmptyException {
// slide 13/14
}  }
```

Method enQ()

```
public enQ(Object item) {
  Node n = new Node(item) ;
  while (true) {
    Node last = tail.get() ;
    Node afterLast = last.next.get();
    if (last == tail.get()) {
     if (afterLast == null) {
       if (last.next.compareAndSet(null, n){
          tail.compareAndSet(last, n) ;
          return ;
       }
```

Method enQ() continued

```
      } else { // afterLast wasn't null!
        tail.compareAndSet(last, afterLast);
      }// end (if (afterlast == null))
     } // end (if (last == tail))- no else
   } // end of while forever loop
} // end of enQ
```

Method deQ()

```
public Object deQ() throws EmptyException {
  while(true) {
    Node first = head.get() ;
    Node last = tail.get() ;
    Node second = first.next.get() ;
    if (first == head.get()) {
      if (first == last){
        if (second == null) {
          throw new EmptyException() ;
        }
```

Method deQ() continued

```
        tail.compareAndSet(last, second);
      } else {//(if (first == last))
        Object ans = second.value ;
        if (head.compareAndSet(first,second))
          return value;
      }// end (if (first == last))
    }// end (if (first == head))
  }// end while forever loop
}// end of deQ
```

Understanding this code – is hard!

- In the absence of locks, need to realise that enQ() needs to update **two** references:
  - One from the last element in the chain
    - Update “tail.next” to point to new node (n)
  - One from tail
    - Update “tail” to point to new node (n)
  - deQ only updates a single reference
    - Update “head” to point next node in chain (and return value from new “head”

Understanding this code

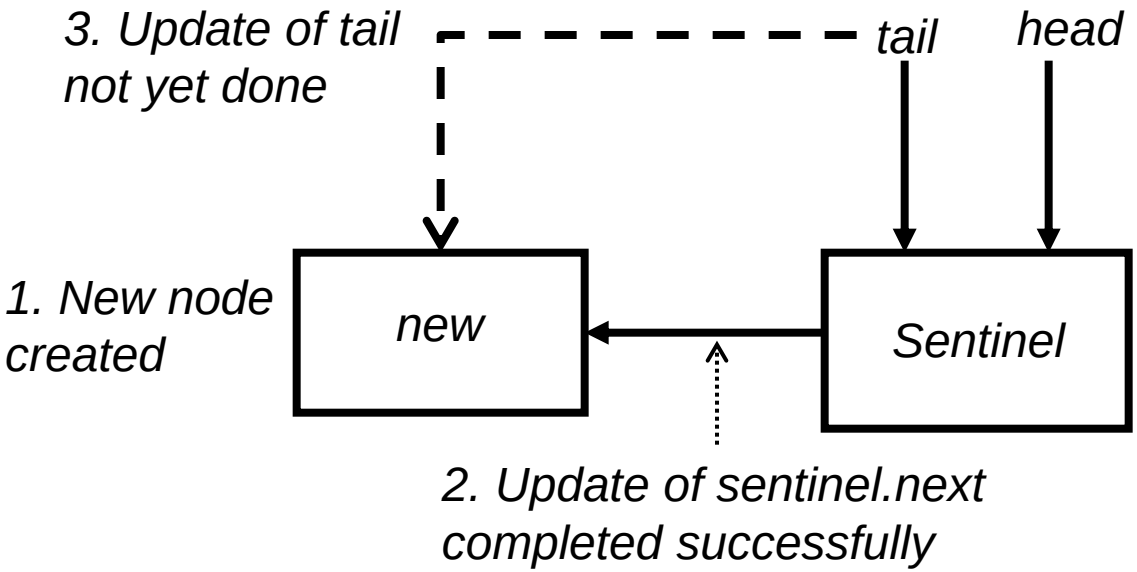
- The two reference updates of enQ are NOT atomic as a pair! (the definition of "lock-free")
  - Thus, interleaving occurs
  - Other threads (calls to enQ or deQ) have to be prepared to handle an "incomplete" enQ
- So they have to be prepared to deal with the first update succeeding and the second (to "tail") failing – i.e. we can have a queue with tail not at the end.
  - On finding this situation, both enQ() and deQ() help to fix it.

Understanding this code

- Two main cases: enQ and deQ on an empty Q; multiple enQs
  - Multiple deQs may also interleave.
- Philosophy of "lock-free" algorithms
  - On entry, a method saves the state of the queue and as the method proceeds, responds appropriately depending on whether the state has changed or not
  - This is similar to the approach taken by LDL/STC

Why deQ() cares about tail

Before moving head on, want to make sure that tail isn't left behind pointing at the previous sentinel.



This is the state if a new node is enQ'd on an empty queue, but tail is not updated before a deQ occurs.

Next Lecture

- OpenMP and MPI are two standards (originating from the world of scientific supercomputing) for writing parallel programs. I will describe them both, and discuss their relevance to programming multicore processors.