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

Lecture 13: October 11

Lecturer: Vijay Garg *Scribes: Rohan Tripathi*

13.1 Concurrent Data Structures

There are 2 choices for building concurrent data structures :-

1. Lock Based
2. Lock Free

The **Lock Based** algorithms can be further divided into:

1. Coarse Grained
2. Fine Grained

The **Lock Free** algorithms can be further divided into:

1. Lock Free
2. Wait Free

Below is an example of a wait free structure:-

```
import java.util.concurrent.atomic.*;
class MyAtomicInteger extends  AtomicInteger {
    public MyAtomicInteger(int val) { super(val);}
    public int myAddAndGet(int delta) {
        for (;;) {
            int current = get();
            int next = current + delta;
            if (compareAndSet(current, next))
                return next;
        }
    }
    public static void main(String[] args) throws Exception {
        MyAtomicInteger x = new MyAtomicInteger(10);
        System.out.println(x.myAddAndGet(5));
    }
}
```

13.1.1 Stack

Stack is one of the simplest concurrent data strucutres in terms of implementation. Stack can be:

- Lock Based
- Lock Free

The 2 major operations associated with stack are *push*(*value*) and *pop*().

The below code represents a lock based Stack :

```
public class Stack<T> {
    Node<T> top = null;

    public synchronized void push(T value) {
        Node<T> node = new Node(value);
        node.next = top;
        top = node;
    }

    public synchronized T pop() throws NoSuchElementException {
        if(top == null) {
            throw new NoSuchElementException();
        } else {
            Node<T> oldTop = top;
            top = top.next;
            return oldTop.value;
        }
    }
}
```

To implement a lock free version of the stack we must make sure that the operations on the data structure are being perfomed atomically even though locks are not being used. A lock free implementation uses *AtomicReference* to achieve this.

```
public class LockFreeStack<T> {
    AtomicReference<Node<T>> top = new AtomicReference<Node<T>>(null);

    public void push(T value) {
        Node<T> node = new Node<T>(value);
                while(true) {
                   Node<T> oldTop = top.get();
                   node.next = oldTop;
                   if (top.compareAndSet(oldTop, node)) return;
                    else Thread.yield();
                }
    }

    public T pop() throws NoSuchElementException {
                while(true) {
                    Node<T> oldTop = top.get();
```

```
                    if(oldTop == null) throw new NoSuchElementException();
                    T val = oldTop.value;
                    Node<T> newTop = oldTop.next;
                    if(top.compareAndSet(oldTop, newTop)) return val;
                    else Thread.yield();
                }
    }
}
```

Speeding up Lock Free stack

The lock free implementation of stack can be made to run faster by adding some design changes. One of them can be $Cancellation - of - Operations$. For example, if a thread is executing a $push(30)$ operation and before the operation can terminate we get a request for $push(70)$ and $pop()$. The last 2 operations can cancel each other out since the effect of physically performing them would have been the same.

13.1.2 Queue

There are 2 types of queues :-

1. Bounded
2. Unbounded

The **Bounded** and **Unbounded** queues can be further divided into:

- Blocking
- Non Blocking

Below is an example of a Bounded Blocking Queue:

```
import java.util.concurrent.locks.*;

class MBoundedBufferMonitor {
   final int size = 10;
   final ReentrantLock monitorLock = new ReentrantLock();
   final Condition notFull  = monitorLock.newCondition();
   final Condition notEmpty = monitorLock.newCondition();

   final Object[] buffer = new Object[size];
   int inBuf=0, outBuf=0, count=0;

   public void put(Object x) throws InterruptedException {
     monitorLock.lock();
     try {
       while (count == buffer.length)
         notFull.await();
       buffer[inBuf] = x;
       inBuf = (inBuf + 1) % size;
       count++;
```

```
      notEmpty.signal();
    } finally {
      monitorLock.unlock();
    }
  }

  public Object take() throws InterruptedException {
    monitorLock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      Object x = buffer[outBuf];
      outBuf = (outBuf + 1) % size;
      count--;
      notFull.signal();
      return x;
    } finally {
      monitorLock.unlock();
    }
  }
}
```

Speeding up Queues

The speedup can be accomplished by having a separate lock for enqueue and dequeue operations.

```
import java.util.concurrent.locks.ReentrantLock;
public class UnboundedQueue<T> {

  ReentrantLock enqLock, deqLock;
  Node<T> head;
  Node<T> tail;
  int size;
  public UnboundedQueue() {
    head = new Node<T>(null);
    tail = head;
    enqLock = new ReentrantLock();
    deqLock = new ReentrantLock();
  }
  public T deq() throws EmptyException {
    T result;
    deqLock.lock();
    try {
      if (head.next == null) {
        throw new EmptyException();
      }
      result = head.next.value;
      head = head.next;
    } finally {
      deqLock.unlock();
    }
    return result;
```

```
  }
  public void enq(T x) {
    if (x == null) throw new NullPointerException();
    enqLock.lock();
    try {
      Node<T> e = new Node<T>(x);
      tail.next = e;
      tail = e;
    } finally {
      enqLock.unlock();
    }
  }
}
```

A special case of queue

The case of single producer and single consumer is faster since there is no need for synchronization and no need for CAS operattion. The enque operation only looks at the tail and dequeue only looks at the head.

```
public class SingleQueue { // Single Producer Single Consumer
  int head = 0;    // slot for get
  int tail = 0;    // empty slot  for put
  Object [] items;
  public SingleQueue(int size) {
    head = 0; tail = 0;
    items = new Object[size];
  }
  public void put(Object x) {
    while (tail - head == items.length) {}; //busywait
    items[tail % items.length] = x;
    tail++;
  }
  public Object get() {
    while (tail - head == 0) {}; // busywait
    Object x = items[head % items.length];
    head++;
    return x;
  }
}
```

It is not possible to write a code for multiple consumers without using a CAS operation.

13.1.3 Michael and Scotts Lock-Free Queue

1. ABA Problem
This problem occurs in Lock free structures due to the ABA problem when we are using memory from the heap. Assume reference X is pointing to object A. A thread T1 reads the value of the reference X and makes a local copy of an updated version. Before it can install the new version using compareAndSet, another thread T2 updates X to B. Furthermore, it deallocates the object A. It again acts on object A and updates X to A. Now if T1 executes compareAndSet,it succeeds because X is still pointing to A. This operation ideally should have failed since the object itself is completely dierent.

2. Helping
Sometimes it is required to perform multiple actions on a lock free implementation. If a thread finds that the remaining actions are idempotent then instead of waiting for the earlier thread to nish its operation, the current thread can simply perform the action before continuing to do its own operation.

13.1.4 Linked List

Linked List has 3major operations $add(value), remove(value)$ and $contains(value)$.

1. Fine Grained Linked List
Fine grained locking uses the principle of implementing multiple locks. In this process a lock is implemented for each node. There is the idea of *LazyDeletion* that can be implemented for hand-over-hand locking in Fine Grained list. Every node has a *isDeleted* bit associated with it. We separate the deletion in 2 parts, the logical deletion and the physical deletion. The logical deletion is followed by the actual physical deletion. When a remove operation is called the isDeleted bit is set to *true*.

For insertion operation, the previous and successive node should be locked. Then we check that both the nodes have not been deleted by checking the *isDeleted* bit. Another thing we have to check is that previous should point to successive node. If there conditions are satisfied the insertion operation occurs otherwise the thread retries.

2. Lock Free Linked List
This can be implemented using principles similar to Free Grained Locking.

References

[1] V. K. Garg, *Introduction to Multicore Computing*

[2] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter6-concurrent-data-structures