1 Introduction

In this chapter we discuss concurrent implmentation of data structures that are useful in many computer science applications. For each of the data structures, we give the main ideas for a lock based as well as a lock free implementation. Generally speaking, lock based data structures are relatively easier to implement. However, lock free implementations are usually more scalable under heavy contention. Lock based implementations use ReentrantLocks discussed earlier whereas lock-free implementations use `compareAndSet` operations. To illustrate lock free constructions, we start with a simple integer object. Suppose that we can atomically read, write or `compareAndSet` the value of this object. We want to add a method that atomically adds `delta` and returns the new value. Furthermore, we do not want to use locks (or `synchronized`) to achieve this. Here is one solution.

```
1  import java.util.concurrent.atomic.*;
2  class MyAtomicInteger extends  AtomicInteger {
3      public MyAtomicInteger(int val) { super(val);}
4      public int myAddAndGet(int delta) {
5          for (;;) {
6              int current = get();
7              int next = current + delta;
8              if (compareAndSet(current, next))
9                  return next;
10         }
11     }
12     public static void main(String[] args) throws Exception {
13         MyAtomicInteger x = new MyAtomicInteger(10);
14         System.out.println(x.myAddAndGet(5));
15     }
16 }
```

The main ideas in this construction are as follows. Every thread first reads the value of the object (or copies the object). It then makes update to a local copy. In our example, every thread reads the value of the object in the local variable `current` uisng `get()`. It then adds `delta` to `current` to get a local copy of the updated version in the variable `next`. Next, it atomically changes the value of the object to `next` only if the current value of the object is still `current`. This atomic change is done via the operation `compareAndSet`. If `compareAndSet` succeeds, then the entire operation has been done atomically. Otherwise, some other thread has changed the value of the `object` in the duration between when it read the object with `get()` and when it called `compareAndSet`. In this case, this thread must retry its operation.

2 Stack

We now show lock based and lock-free constructions of stacks. The lock based construction is quite simple. It uses `synchronized` to ensure mutual exclusion. In this implementation, a `pop` on empty stack results in `NoSuchElementException`.

2.1 Lock Based Stack

```
1  import java.util.NoSuchElementException;
2
3  public class Stack<T> {
4      Node<T> top = null;
5
6      public synchronized void push(T value) {
7          Node<T> node = new Node(value);
8          node.next = top;
9          top = node;
10     }
11
12     public synchronized T pop() throws NoSuchElementException {
13                 if(top == null) {
14                     throw new NoSuchElementException();
15                 } else {
16                 Node<T> oldTop = top;
17                 top = top.next;
18                 return oldTop.value;
19                }
20
21     }
22
23 }
```

2.2 Lockfree Stack

For lock free construction of push(), we need to add a node at the head of the linked list atomically. We create a node such that its next pointer points to the current top. We then use `compareAndSet` to atomically change the reference `top` if it has not been changed since it was last read using `top.get()`.

```
1  import java.util.NoSuchElementException;
2  import java.util.concurrent.atomic.AtomicReference;
3
4  public class LockFreeStack<T> {
5      AtomicReference<Node<T>> top = new AtomicReference<Node<T>>(null);
6
7      public void push(T value) {
8          Node<T> node = new Node<T>(value);
9                  while(true) {
10                     Node<T> oldTop = top.get();
11                     node.next = oldTop;
12                     if (top.compareAndSet(oldTop, node)) return;
13                      else Thread.yield();
14                 }
15     }
16
17
18     public T pop() throws NoSuchElementException {
19                 while(true) {
20                      Node<T> oldTop = top.get();
21                      if(oldTop == null) throw new NoSuchElementException();
22                      T val = oldTop.value;
23                      Node<T> newTop = oldTop.next;
24                      if(top.compareAndSet(oldTop, newTop)) return val;
25                      else Thread.yield();
26                 }
27     }
28
29
30 }
```

3 Queues

In stacks, both push and pop operations (insert and delete) happen at the same end of the linked list. Queues, in contrast, have inserts at one end and the deletes at the other end. Hence, it is easier to exploit parallelism for queues.

3.1 Lock Based Queue

Fig. 1 shows a simple single lock based queue constructed with a circular array as discussed in Chapter 3. This implementation requires both `put` and `take` to be executed under the same lock.

We now show a version that uses two locks to increase concurrency. Fig. 2 shows a queue that uses seperate locks for `head` and `tail`. In addition, we use two techniques to simplify programming. First, we use a *sentinel* node which ensures that `head` and `tail` references are never null. If there is only one node in the queue, then the queue is empty. The first node in the queue is given by head.next. The second technique used in this program is that to dequeue a value from the queue, we get the value from the head.next node and then instead of deleting that node we delete the sentinel node. Now, the earlier head.next node serves as the sentinel node.

In this implementation `deq()` throws EmptyException when it is called on an empty queue. Building a queue that forces the thread to wait until an item is inserted is left as an exercise.

```
1  import java.util.concurrent.locks.*;
2
3  class MBoundedBufferMonitor {
4     final int size = 10;
5     final ReentrantLock monitorLock = new ReentrantLock();
6     final Condition notFull  = monitorLock.newCondition();
7     final Condition notEmpty = monitorLock.newCondition();
8
9     final Object[] buffer = new Object[size];
10    int inBuf=0, outBuf=0, count=0;
11
12    public void put(Object x) throws InterruptedException {
13      monitorLock.lock();
14      try {
15        while (count == buffer.length)
16          notFull.await();
17        buffer[inBuf] = x;
18        inBuf = (inBuf + 1) % size;
19        count++;
20        notEmpty.signal();
21      } finally {
22        monitorLock.unlock();
23      }
24    }
25
26    public Object take() throws InterruptedException {
27      monitorLock.lock();
28      try {
29        while (count == 0)
30          notEmpty.await();
31        Object x = buffer[outBuf];
32        outBuf = (outBuf + 1) % size;
33        count--;
34        notFull.signal();
35        return x;
36      } finally {
37        monitorLock.unlock();
38      }
39    }
40 }
```

Figure 1: Bounded Buffer Using ReentrantLocks and Conditions

```
1  import java.util.concurrent.locks.ReentrantLock;
2  public class UnboundedQueue<T> {
3
4    ReentrantLock enqLock, deqLock;
5    Node<T> head;
6    Node<T> tail;
7    int size;
8    public UnboundedQueue() {
9      head = new Node<T>(null);
10     tail = head;
11     enqLock = new ReentrantLock();
12     deqLock = new ReentrantLock();
13   }
14   public T deq() throws EmptyException {
15     T result;
16     deqLock.lock();
17     try {
18       if (head.next == null) {
19         throw new EmptyException();
20       }
21       result = head.next.value;
22       head = head.next;
23     } finally {
24       deqLock.unlock();
25     }
26     return result;
27   }
28   public void enq(T x) {
29     if (x == null) throw new NullPointerException();
30     enqLock.lock();
31     try {
32       Node<T> e = new Node<T>(x);
33       tail.next = e;
34       tail = e;
35     } finally {
36       enqLock.unlock();
37     }
38   }
39
40 }
```

Figure 2: Unbounded Queue Using Two ReentrantLocks

3.2 Lockfree Queue

We first start with a bounded queue with a singe producer and a single consumer. Such a queue can be built without using any locks and without any `compareAndSet` operations! Since there is only one producer and one consumer, we only need to ensure that there is no interference between concurrent activations of `put` and `get`. In this program, only the producer can update the variable `tail` and only the consumer can update the variable `head`. While one thread is updating a variable, if another thread reads the variable then it gets either the old value or the new value of the variable.

```
1  public class SingleQueue { // Single Producer Single Consumer
2    int head = 0;    // slot for get
3    int tail = 0;    // empty slot  for put
4    Object [] items;
5    public SingleQueue(int size) {
6      head = 0; tail = 0;
7      items = new Object[size];
8    }
9    public void put(Object x) {
10     while (tail - head == items.length) {}; //busywait
11     items[tail % items.length] = x;
12     tail++;
13   }
14   public Object get() {
15     while (tail - head == 0) {}; // busywait
16     Object x = items[head % items.length];
17     head++;
18     return x;
19   }
20 }
```

While the above construction works for single producer and single consumer, it cannot be generalized to multiple producers and multiple consumers.

3.3 Optional Material: Michael and Scott's Lock-Free Queue

We now show an algorithm shown in Fig. 3 due to Michael and Scott for lock-free construction of queues. This construction has the following ideas.

- **Avoiding ABA Problem**: Lockfree constructions built using `compareAndSet` may have the ABA problem when we use memory from the heap. Suppose that we have a reference X pointing to an object A. A thread T_1 reads the value of the reference X and makes a local copy of an updated version of the object. Before it can install the new version using `compareAndSet`, another thread T_2 updates X to B. Furthermore, it deallocates the object A. Now, another thread T_3 comes in. It recycles the object A and updates X to A. Now when T_1 executes `compareAndSet`, it succeeds because X is still pointing to A even though the object itself is completely different.

  In Fig. 3, this problem is addressed via accessing nodes using `pointer_t` that keeps a `pointer` as well as a `count`. The operation `compareAndSet` is performed on `pointer_t` which is changed even when the pointer points back to reclaimed node. The field `count` is updated on every `CAS`. Java provides `AtomicStampedReference` for this functionality.

- **Helping Other Operations**: In many lock-free constructions, an operation may require multiple actions to be done on a data structure. Since there are no locks, other threads may find the data structure in a state in which not all actions have been taken. If the remaining action is idempotent

(i.e. doing it multiple times is equivalent to doing it once), then instead of waiting for the earlier thread to finish its operation, the current thread can simply perform the action before continuing to do its own operation.

The `enq` operation requires two major actions: the new node must be added after the tail node and the tail pointer should be moved to the new node. Suppose that thread T_1 does an `enq` operation, but before it could do the second step another thread T_2 barges in. At this point, the tail is not pointing to the last node. Thread T_2 can now help T_1 by advancing the tail. Whenever T_1 comes back and tries to advance the tail, it will find that the tail has changed and its `CAS` operation will have no effect.

In the pseudo-code shown in Figure 3, lines E1-E3 create a node to be inserted. Lines E4-E16 try to add this node to the last node. A thread can exit from the loop only when its CAS at line E9 succeeds. Line E12 corresponds to helping a tail that was lagging behind. Line E17 corresponds to the second action in which the tail is pointed to the inserted node.

Lines D1 to D18 represent a loop for `dequeue` that can be exited either at D8 when the queue is empty or at D14 when D13 succeeds in deleting the node at head. Just as in two-lock version of queues, we are using a sentinel node for queue. The value at the sentinel node is irrelevant and the value to be returned is the value at the node pointed by the sentinel node. This is accomplished by line D12. Line D13 deletes the sentinel node and the next node now becomes a sentinel node.

4 Linked Lists

In this section we discuss methods to implement concurrent linked lists. The easiest implementation is based on protecting the entire linked list by a single lock. However, this implementation does not offer any parallelism. All operations on the list are serialized.

For concreteness, we will implement a sorted linked list. We discuss two implementations: fine-grained lock based and lock-free. We consider three operations: `insert`, `remove` and `search`. To avoid the case analysis, we keep two dummy nodes as `head` and `tail`. The actual nodes of the linked lists are between these two dummy nodes.

4.1 Fine-Grained Locking

Linked lists bring out many important principles when dealing with fine-grained locks for data structures. First, most data structures have much higher percentage of read operations than update operations. Therefore, the overall performance of these data structures can be improved by optimizing for read operations. In fact, every effort should be made that the read operations are completely wait-free. The second useful idea is that of *lazy deletion*. Since a data structure such as a linked list or a tree can result in physical change when a node is deleted and there may be threads that are traversing the data structure, it is helpful to do a lazy deletion. Every node maintains a bit: `isDeleted`. The actual delete operation may be complex but by marking that node as isDeleted, we have *logically* deleted the node. The logical deletion is followed by the actual physical deletion by that thread (or by another thread using the idea of helping another thread). The algorithm maintains the invariant that any node that is not deleted is reachable from the head node in spite of concurrent insertions and removals.

With these two ideas, we can implement search without locks quite simply. To look for an item with key k, we traverse the list till we find either key k or a key greater than k. If we find key k, then we have found the required node if `isDeleted` is false. Otherwise, we return false. If we hit a node with key greater than k, then again we return false.

```
structure pointer_t {ptr: pointer to node_t, count: unsigned integer}
structure node_t {value: data type, next: pointer_t}
structure queue_t {Head: pointer_t, Tail: pointer_t}

initialize(Q: pointer to queue_t)
   node = new_node() // Allocate a free node
   node->next.ptr = NULL // Make it the only node in the linked list
   Q->Head.ptr = Q->Tail.ptr = node // Both Head and Tail point to it

enqueue(Q: pointer to queue_t, value: data type)
 E1:   node = new_node() // Allocate a new node from the free list
 E2:   node->value = value // Copy enqueued value into node
 E3:   node->next.ptr = NULL // Set next pointer of node to NULL
 E4:   loop // Keep trying until Enqueue is done
 E5:      tail = Q->Tail // Read Tail.ptr and Tail.count together
 E6:      next = tail.ptr->next // Read next ptr and count fields together
 E7:      if tail == Q->Tail // Are tail and next consistent?
             // Was Tail pointing to the last node?
 E8:         if next.ptr == NULL
                // Try to link node at the end of the linked list
 E9:            if CAS(&tail.ptr->next, next, <node, next.count+1>)
E10:               break // Enqueue is done.  Exit loop
E11:            endif
E12:         else // Tail was not pointing to the last node
                // Try to swing Tail to the next node
E13:            CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
E14:         endif
E15:      endif
E16:   endloop
       // Enqueue is done.  Try to swing Tail to the inserted node
E17:   CAS(&Q->Tail, tail, <node, tail.count+1>)

dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
 D1:   loop      // Keep trying until Dequeue is done
 D2:      head = Q->Head      // Read Head
 D3:      tail = Q->Tail      // Read Tail
 D4:      next = head.ptr->next    // Read Head.ptr->next
 D5:      if head == Q->Head      // Are head, tail, and next consistent?
 D6:         if head.ptr == tail.ptr // Is queue empty or Tail falling behind?
 D7:            if next.ptr == NULL  // Is queue empty?
 D8:               return FALSE      // Queue is empty, couldn't dequeue
 D9:            endif
                // Tail is falling behind.  Try to advance it
D10:            CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
D11:         else      // No need to deal with Tail
                // Read value before CAS
                // Otherwise, another dequeue might free the next node
D12:            *pvalue = next.ptr->value
                // Try to swing Head to the next node
D13:            if CAS(&Q->Head, head, <next.ptr, head.count+1>)
D14:               break             // Dequeue is done.  Exit loop
D15:            endif
D16:         endif
D17:      endif
D18:   endloop
D19:   free(head.ptr)      // It is safe now to free the old node
D20:   return TRUE                   // Queue was not empty, dequeue succeeded
```

Figure 3: Michael and Scott's Lock Free Queue Algorithm: [Michael Scott's web page]

Let us now examine the operation `insert(k)`. An insertion of a node x with key k requires presence of two nodes: `prev` and `succ` such that the key stored at `prev` is less than k, and the key stored at `succ` is greater than k. At this point, we need to take care of concurrent updates. Hence, both the nodes `prev` and `succ` must be locked. Now we can check if both nodes satisfy all the required constraints for insertion of x. These are: (1) Neither `pred`, nor `succ` should have been deleted, and (2) The next pointer of `pred` should be pointing to `succ`. Since both nodes were locked before the check for these conditions, we are guaranteed that if both conditions hold, then they will continue to do so till these nodes are unlocked. If either of the condition fails, we can retry the *insert* operation. Otherwise, we can finish inserting x in the linked list. Care must be taken that $x.next$ points to `succ` before $prev.next$ is changed to x. This is because we are using lock free search and a concurrent search should never traverse a null pointer.

Finally, we consider the `remove` operation. Let the node that needs to be deleted be `curr`, the node that points to `curr` be `prev`. We will require both nodes `curr` and `prev` to be locked before any actual change. Moreover, `prev` and `curr` should satisfy properties similar for insertion. Both `prev` and `curr` must not have `isDeleted` flag set and $prev.next$ must point to $curr$. If these properties do not hold, we retry the remove operation. Otherwise, we first set `curr`'s `isDeleted` and then set $prev.next$ to $curr.next$. Note that if both `prev` and `curr` are locked, the node $curr.next$ cannot be removed concurrently and there cannot be any insertion at node `curr`.

4.2 Optional Material: Lock Free Linked List

We now discuss implementation of a lock free linked list. Many of the ideas are similar to those use in fine-grained linked list. We need a bit with every node to mark if the node has been deleted or not. We use `AtomicMarkableReference` in Java for this purpose. An AtomicMarkableReference has a Reference as well as a mark that can updated atomically. Similarly, one can `CAS` with expected (reference, mark) and update both reference and mark atomically if the current reference and the mark are as expected.

To insert a node x after `prev` and before `curr`, we first create a node x such that it points to `curr`. Now we need to check that `prev` is not deleted and $prev.next$ points to `curr`. If this check succeeds, then $prev.next$ must be set to x. The check and update of $prev.next$ can be done in one `compareAndSet` instruction.

To remove a node `curr`, two actions need to be taken. First, the mark of AtomicMarkableReference curr.next needs to be set true. This corresponds to setting `isDeleted` flag true in the lock based Linked List. We can use a CAS to perform this action. If the CAS does not succeed, then the remove operation can be retried. The second action requires a check that pred is not deleted, and pred.next points to curr. If the condition is true, then pred.next should be set to `curr`. The check and update can be done atomically in a CAS operation.