Parallel HW4 Theory Problems
Andrew McCarthy

Problems: 108, 109, 121, 124, 125, 127, 129, 132, 159, 160, 185, 186, 188, 192

108)

There are three situations in which threads could overlap:

- add() add(): Two threads want to add a value at the same time into the list. If they both want to add something after the same element, then one waits. If the list contains a, b and c in this order and thread A wants to add something between a and b and thread B wants to add something between b and c this is fine because thread A does not modify b, but only sets a pointer to it.
- add() remove(): Thread A wants to add something and thread B wants to remove something from a list a, b and c. If thread A wanted to add something between a and b, and had already acquired the lock for pred, then b can't delete a or b because either the lock pred or curr would belong to thread A.
- add() - contains(): Same as with add() - remove().

109)

No matter if the method returns true or false, the node contains() checks must have been accessible from the head at some point. This point can then be chosen for the linearization.

121)

Array List

Items[]
Int head = 0;
Int tail = 0;

Lock lockHead = new lock()
Lock lockTail = new lock()

```
Condition notEmpty = lockHead.newCondition()
Condition notFull = lockTail.newCondition()
BoundedQueue(int capacity) {
        Items = (T[]) new object[capacity]
}

enq(T item) {
        lockTail.lock();
        While (tail - head = items.length) {
                notFull.await
        }
        Items[tail % items.length] = item;
        Tail++;
        If (tail-head == 1) {
                notEmpty.signal;
        }
        lockTail.unlock()
}

deq() {
        lockHead.lock()
        While (tail-head == 0) {
                notEmpty.await()
        }
        T x = items[head % items.length];
        Head++;
        If (head-tail == items.length - 1) {
                NotFull.signal;
        }
        Return x;
        lockhead.unlock();
}
```

Lock Free

```
T[] items;

head = new AtomicInteger;
```

```
tail = new AtomicInteger;
tailCommit = new AtomicInteger;
AtomicInteger latentCapacity;

LockFreeBoundedQueue(int capacity) {
      items = (T[]) new Object[capacity];
      latentCapacity = new AtomicInteger(capacity);
}

enq(T item) {
      int lc = latentCapacity.get();
      while (lc <= 0 || !latentCapacity.compareAndSet(lc, lc - 1))
         lc = latentCapacity.get();

      int t = tail.getAndIncrement();
      items[t % items.length] = item;

      while (tailCommit.compareAndSet(t, t + 1)) { };
   }

T deq() {
      int h = head.getAndIncrement();
      while (h >= tailCommit.get()) { };
      T item = items[h % items.length];
      latentCapacity.incrementAndGet();

      return item;
   }
}
```

One problem is defining where it is valid to read or write the indices. I had to add two more variables: the first representing how many elements can still be inserted, the second representing the valid indices for reading.

124)

1. Yes we can. The natural linearization point is the compareAndSet operation that removes the item from the queue. The order of reads that precede each compareAndSet call is the

same as the order of those compareAndSet calls. Each read falls within the interval of the call

2. No we cannot. The natural linearization point is the compareAndSet operation that appends the new node to the end of the list. Although the order of the compare and Set calls that update the tail field is the same as the order of the appending compareAndSey calls, these updating calls may not occur during the original enq() operation

125)

1. The enqueue function is wait free because there are no loops or conditionals
2. If the queue is empty, deq() is not lock-free because it will run forever. If the queue is non-empty when deq() is called, then deq() is lock -free but not wait free as it has to wait for an element.

1. The linearization point for enq() is when it stores its items in the array.
2. The linearization point for deq() is when it calls a getAndSet() that returns a non-null value.

127)

```
E[] arr = null;
int CAP;
int top = -1;
int size = 0;
Int lock;

public Stack(int cap) {
        this.CAP = cap;
        this.arr = (E[]) new Object[cap];
        lock=reentrantLock()
}

 pop() {
        lock.lock()
        if(this.size == 0){
                return null;
        }

        this.size--;
```

```
        E result = this.arr[top];
        this.arr[top] = null;
        This.top--;
        lock.unlock();
        return result;
}
push(e) {
        lock.lock()
        if (isFull())
                return false;

        this.size++;
        this.arr[++top] = e;

        lock.unlock();
        return true;
}
```

1. The issue with a lock free approach is decrementing the top and removing the item atomically. Removing an item and decrementing the top runs into problems because other threads can modify the stack between those two steps.

129)

For the LockFreeStack,we should use the Backoff object for pushes and pops because they compete with one another. For the EliminationBackoffStack, a backoff object is overkill and we should use that time in the elimination array.

132)

If push() runs concurrently with a pop() it will create problems. if push() increments top, and then a concurrent pop() decrements it, the pop() method will remove a meaningless value from the stack. To fix this problem, we can use room synchronization. When a thread calls a stack method, it enters a room associated with said method. Each room can hold an arbitrary number of threads, but only one room can be occupied at a time. For example, any number of threads might enter the push room and each will execute the push() method, but no thread can enter the pop room while the push room is occupied.

159)

If given a scenario where node a has predecessor node p and successor node s, and a is also an entry in the bucket array, an add() call intending to insert an entry b between a and s, can find a in two ways: either a's bucket entry, or by scanning directly through the list. To remove a, we need to change p's next field and the bucket entry to s. If we first change p's next field, a concurrent add(b) call that finds the bucket entry for a will link b between a and s. This list is incorrect because neither a nor b are reachable from the start of the list. If we instead redirect the bucket entry from a to s, a concurrent add(b) call that finds a by scanning through the list will link b between a and s. The resulting list is incorrect because the bucket entry should refer to b instead of to b's successor

160)

Grow the table to size 2^i while adding keys with least significant bit 0. If we add a key consisting of all 1s, we will need to initialize i dummy nodes of form 1...10, where the length of the contiguous sequence of 1s ranges from 1 to i − 1.

Although the total complexity in this case is logarithmic, it still works out to be constant. given a uniform distribution of items the chances of the above scenario happening are low, and the expected length of such bad parent initializations is constant.

185)

Assuming that merge is $\Theta(n)$, we have $T1(n) = 2*T1(n/2) + \Theta(n)$. Therefore, $T1(n) = \Theta(n \log n)$. $T(\text{infinity})(n) = T1(n/2) + \Theta(n) = \Theta(n)$. The parallelism is $\Theta(\log n)$.

186)

Using the formulas:

$T512 = T1 / P + T(\text{infinity}) = 2048 / 512 + 1 = 5s$
$T'512 = T'1/ \quad P + T'(\text{infinity}) = 1028 / 512 + 8 = 10s$

We conclude that the unoptimized version runs faster on a machine with more processors. The larger critical path length of the second implementation greatly hinders its performance on machines with many processors.

188)

(written on paper on the last page)

192)

The queues could change sizes between reading the first and second queue sizes. As a result, two threads could lock the same two queues in different orders, resulting in a deadlock.

188) upper bound for T_1:

$$T_4 \geq \frac{T_1}{4}$$

$$80 \geq \frac{T_1}{4}$$

$$320 \geq T_1$$

T_{64} yields a worse bound

$$T_{64} \geq \frac{T_1}{64}$$

$$10 \geq \frac{T_1}{64}$$

$$640 \geq T_1$$

$$T_{64} \geq T_\infty$$

$$10 \geq T_\infty$$

$$T_{10} \leq \frac{(T_1 - T_\infty)}{10} + T_\infty$$

$$\leq \frac{T_1}{10} + \frac{9 \cdot T_\infty}{10}$$

$$\leq \frac{320}{10} + \frac{9 \cdot T_\infty}{10}$$

$$\leq \frac{320}{10} + \frac{9 \cdot 10}{10}$$

$$\leq 32 + 9$$

$$\boxed{\leq 41}$$