90 points total.

I. Theory [40 points]

1. [10 points] A proposed algorithm for mutual-exclusion works as follows:

   Let there be a multi-writer register `roll`, whose range is $\{-1, 0, \ldots, N-1\}$ and the initial value is -1. To enter the critical section, a process i spins on `roll`, until the value is -1. And when it finds the value to be -1, it sets the value of `roll` to i, within one time unit of the read. After that, the process waits for more than one time unit, and then reads `roll`. If it still has the value i, then it enters the critical section. Otherwise, it returns to spinning on `roll` until the value is -1. When a process exits its critical section, its sets the value of `roll` to -1.

   (a) Does the algorithm guarantee mutual exclusion?

   (b) Is it livelock-free?

   (c) What are the drawbacks of this algorithm (in terms of performance and requirement)?

2. [10 points] Consider a four-socket system comprising of four 10-core processors. These processors are connected with each other through a point-to-point interconnect. Cache coherency within a socket is maintained through snooping. Coherency across different sockets is implemented through a broadcast mechanism. Assume a high contention scenario where n threads simply try to acquire a single lock. n is varied from 2 to 40. Based on this, answer the following questions:

   (a) What will happen to the throughput for threads $n > 10$? Give reason.

   (b) Which lock(s) among TAS, TTAS, ALock, CLH and MCS you expect to be strong against contention as n varies from 2 to 40? Why?

3. [5 points] Figure 1 shows a modified implementation of ALock, with the only change being the two statements (Line 24 and Line 25) in the unlock procedure are switched. Prove/disprove that this implementation is correct. (Consider `size` $= 3$).

```
1  public class ALock implements Lock {
2
3   ThreadLocal<Integer> mySlotIndex = new ThreadLocal<
       Integer> (){
4     protected Integer initialValue() {
5       return 0;
6     }
7    };
8    AtomicInteger tail;
9    boolean[] flag;
10   int size;
11
12   public ALock(int capacity) {
13     size = capacity;
14     tail = new AtomicInteger(0);
15     flag = new boolean[capacity];
16     flag[0] = true;
17   }
18   public void lock() {
19     int slot = tail.getAndIncrement() % size;
20     mySlotIndex.set(slot);
21     while (! flag[mySlotIndex.get()]) {};
22   }
23   public void unlock() {
24     flag[(mySlotIndex.get() + 1) % size] = true;
25     flag[mySlotIndex.get()] = false;
26   }
27 }
```

Figure 1: Anderson's Lock

4. [15 points] MCS Lock (Figure 2)

   (a) Consider a modified implementation of MCS Lock when `getAndSet` is not atomic, i.e., `get()` and `set()` are separate. Prove/disprove that this implementation is correct.

   (b) Consider a modified implementation of MCS Lock when `compareAndSet` is not atomic, i.e., `compare()` and `set()` are separate. Prove/disprove that this implementation is correct.

   (c) Prove that the unlock procedure of the MCS lock is deadlock-free.

   (d) Consider a modified implementation of MCS Lock where Line 16 and Line 17 are swapped. Prove/disprove that this implementation is correct.

II. Implementation [50 points]

1. [10 points] Imagine n threads, each of which executes method `foo()` followed by method `bar()`. Suppose we want to make sure that no thread starts `bar()` until all threads have finished `foo()`. For this kind of synchronization, we place a barrier between `foo()` and `bar()`.

   Barrier implementation: We have an n-element array b, all 0. Thread zero sets $b[0]$ to 1. Every thread i, for $0 < i \leq n-1$, spins until $b[i-1]$ is 1, sets $b[i]$ to 1, and waits until $b[n-1]$ becomes 2, at which point it proceeds to leave the barrier. Thread $b[n-1]$, upon detecting that $b[n-1]$ is 1, sets $b[n-1]$ to 2 and leaves the barrier.

   Implement the barrier and measure the barrier time (i.e., time between `foo()` and `bar()`) as a function of number of threads (upto 16; Use Rlogin).

2. [20 points] Implement the four locks below (you may reuse textbook code) and measure the time to execute an empty critical section as a function of number of threads for all locks specified below, and plot the data (similar to Figure 7.4 in the textbook). Collect data for upto 16 threads (Use Rlogin).

   Note: You will actually measure throughput (in terms of the number of locks acquired) over a two second period. Then you can calculate the average waiting time, which you need to plot. Also, you *might*

```
1  public class MCSLock implements Lock {
2    AtomicReference<QNode> queue;
3    ThreadLocal<QNode> myNode;
4    public MCSLock() {
5      queue = new AtomicReference<QNode>(null);
6      myNode = new ThreadLocal<QNode>() {
7        protected QNode initialValue() {
8          return new QNode();
9        }
10     };
11   }
12   public void lock() {
13     QNode qnode = myNode.get();
14     QNode pred = queue.getAndSet(qnode);
15     if (pred != null) {
16       qnode.locked = true;
17       pred.next = qnode;
18       while (qnode.locked) {}
19     }
20   }
21   public void unlock() {
22     QNode qnode = myNode.get();
23     if (qnode.next == null) {
24       if (queue.compareAndSet(qnode, null))
25         return;
26       while (qnode.next == null) {}
27     }
28     qnode.next.locked = false;
29     qnode.next = null;
30   }
31
32   static class QNode {
33     boolean locked = false;
34     QNode   next = null;
35   }
36 }
```

Figure 2: MCS Lock

want to account for JVM warm-up in your measurements. Simply take measurements 3 times (through a loop), and consider the last one.

(a) `testAndSet` lock

(b) `testAndTestAndSet` lock

(c) CLH queue lock

(d) MCS queue lock

3. [20 points] Priority-based CLH Lock

   The CLH lock provides FCFS fairness, which is a useful property for many applications. But some applications may attach priorities to threads (e.g., some thread may have a higher priority than others). Design and implement a priority CLH lock that ensures that the highest priority thread that is waiting for the lock is always granted access to the critical section than all other waiting threads.

   Note that we don't care about the priority of the thread that is currently holding the lock, which may or may not have a higher priority than the highest priority waiting thread. Thus, whenever the lock is released, the highest priority waiting thread is granted access first. (If you include the priority of the lock holder in the total priority ordering of the lock, then it may require aborting the lock holder, executing roll-back logic, etc., which is outside the scope of this homework.)

   Beside the *lock*() and *unlock*() methods for your CLH lock, also implement a *trylock*() method that attempts to acquire the lock (respecting thread priorities), and if it can't acquire it within a predefined amount of time (constructor argument in ms), it fails/aborts. *trylock*() will return a `boolean` value indicating whether the lock was successfully acquired.

   For each thread, measure the waiting time before starting critical section execution, and multiply it by its priority (where priority=1 is the highest priority, and priority=5 is the lowest).

   Obtain the average of your calculated data as a function of number of threads and plot the data (similar to Figure 7.4 of textbook). Include both priority CLH lock (only the lock() method) and FCFS CLH lock in the plot.

Hints: Don't try to implement the P-CLH lock the same way the CLH lock is implemented. Instead, use the principles behind the CLH lock. This is easier if you use Java's PriorityBlockingQueue, and let threads spin on their own node, instead of the predecessor's.