COL 380	**January 20, 2019**
Homework 1	
Instructor: Subodh Sharma	**Due: January 25, 23:55 hrs**

NOTE: Answer submissions must be made either in word document or pdfs. No hand-written assignments would be accepted. All assignment submissions will be checked for plagiarism.

Problem 1:

Consider an SMP with a distributed shared-address-space consisting of 4 processing units running at 1 GHz. Each processing unit has a cache and is connected to a shared-memory. It takes negligible time (consider 1ns) to access a word from local cache and 100ns from shared memory. Consider following producer-consumer program:

```
prod = tid % 2;
i = tid;
thread begin:
      if (i < buffersize)
            if (prod == 1)
                  produceItem(i);
                  prod = 0;
            else
                  consumeItem(i);
                  prod = 1;
      i = i+4
end
```

The `produceItem()` and `consumeItem()` takes 20 and 25 instruction cycles respectively with one memroy read/write operation each. Assume that this program is running with buffersize of 40K words, spread equally over memory of all 4 machine (each machine has 10K consecutive words in its memory) with 99% cache hit ratio. Compute the peek achievable performance of system for SIMD and MIMD architecture. Assume that all instructions other than `produceItem()` and `consumeItem()` takes no cycles.

Problem 2:

Let R^i denote a read on a *register* that returns v^i, the unique value written by W^i (where i is the index of totally ordered writes to the register). A register is said to be *regular* if it allows multiple readers and single writer and the following conditions hold true:

- It is never the case that $R^i \rightarrow W^i$
- It is never the case that for some j, $W^i \rightarrow W^j \rightarrow R^i$

Explain whether the Peterson's two-thread ME algorithm would work when the atomic flag registers are replaced by *regular* registers.

Problem 3:

Consider the following generalization of Peterson's ME algorithm for n threads:

```java
class n-thread-Peterson implements Lock{
  private int turn;
  private boolean busy = false;
  public void lock() {
    int me = TID.get();
    do {
      do {
        turn = me;
        } while (busy);
      busy = true;
    } while (turn != me);
  }
  public void unlock() {
    busy = false;
  }
}
```

Show a proof or counterexample for each of the following: Does the protocol satisfy mutual exclusion, deadlock-freedom and starvation-freedom?

Problem 4:

Prove that sequential consistency is non-blocking.

Problem 5:

Consider the following implementation of a Queue:

```java
1    class Q<T> {
2      AtomicInteger head = 0; AtomicInteger tail = 0;
3      Atomic<T> items[];
4      void enq(T x) {
5        if(tail - head == items.length){
6          throw FullException();
7        }
8        items[tail % items.length] = x;
9        tail++;
10     }
11     T deq() {
12       if( tail - head == 0){
13         throw EmptyException();
14       }
15       T x = items[head % items.length];
16       head++;
17       return x;
18       }
19   }
```

Assume that each instruction of enq() and deq() methods executes atomically (*i.e.* without thread interference) but methods themselves are not atomic. Consider the specification of queue object as: (a) If queue is nonempty and not full then a subsequent deq() must return the element at location head; (b) If the queue is empty then a subsequent deq() must throw empty exception; (c) If the queue is full then a subsequent enq() must throw full exception. Assume that the linearization point for enq() is at line 9. For a 2-threaded system, is the queue object linearizable? Explain your answer.

Problem 6:

Consider the two correct lock implementations below:

```
public class Lock1 implements Lock{
  AtomicBoolean state - new AtomicBoolean(false);
  public void lock() {
    while(state.getAndSet(true)){}
  }
  public void unlock() {
    state.set(false);
  }
}

public class Lock2 implements Lock{
  AtomicBoolean state - new AtomicBoolean(false);
  public void lock() {
    while(true){
      while(state.get()){}
      if (!state.getAndSet(true)) return;
    }
  }
  public void unlock() {
    state.set(false);
  }
}
```

The `getAndSet(true)` atomically replaces the current value with `true` and returns the previous value. Compare the Lock1 and Lock2 functions in terms of performance. Explain your answer.