CSE 150 Homework 1
Spring 2019

1. (18 points) Given the three thread states: running, runnable (i.e., ready), and blocked (i.e., waiting), state which of the six possible thread transitions are allowed, and which ones are not. Also, for each case, provide an example of how the transition occurs, or a reason why it cannot (as a justification for each of your answers). Assume Mesa-style scheduling.

2. (14 points) The following pseudocode swaps two values in a data structure if both entries (holding the values) are not null. If either entry is NULL, the data structure is left as before the swap was attempted. The routine must appear to occur atomically and must be highly concurrent – it must allow multiple swaps between unrelated entries in parallel. You may assume that `index1` and `index2` never refer to the same entry.

```
SwapNonNull(index1, index2) {
   data[index1]->lock.Acquire();
   data1 = getData(index1);    /* May involve disk I/O */
   data[index2] ->lock.Acquire();
   data2 = getData(index2);    /* May involve disk I/O */
   if (data1 == NULL || data2 == NULL) return;
   else {
      temp =  data1->val;
      data1->val = data2->val;
      data2->val = temp;
   }
   storeData(data1);   /* Involves disk I/O */
   storeData(data2);  /* Involves disk I/O */
   data[index1]->lock.Release();
   data[index2]->lock.Release ();
   return;
```

State whether the aforementioned routine either (i) works, (ii) doesn't work, or (iii) is dangerous – that is, sometimes works and sometimes doesn't. If the implementation does not work or is dangerous, explain why (there may be several errors) and show how to fix it so it does work.

3. (13 points) List three ways …

- **a. (4 points)** in which the processor can transition from user mode to kernel mode. Can the user execute arbitrary code after transitioning?
- **b. (3 points)** in which context-switching can happen in a non-preemptive scheduler.
- **c. (6 points)** of preventing deadlock between these processes in a system with several processes and resources. Explain each method and its consequences to programmer or applications.

4. (23 points) Nachos related questions

- **a. (4 points)** List two reasons why Nachos disables interrupts when a thread/process sleeps, yields, or switches to a new thread/process?

- **b. (3 points)** In Nachos, does disabling interrupts prevent context switches. If so why, if not why not?

c. **(4 points)** Consider the following variation of the implementation of condition variables in Nachos, where we comment out lines that assert that a lock is held by the current thread. 4

```
public class Condition {

    private Lock conditionLock;
    private LinkedList<Semaphore> waitQueue;

    public Condition(Lock conditionLock) {
        this.conditionLock = conditionLock;
        waitQueue = new LinkedList<Semaphore>();
    }

    public void sleep() {
        // Lib.assertTrue(conditionLock.isHeldByCurrentThread());
        Semaphore waiter = new Semaphore(0);
        waitQueue.add(waiter);
        conditionLock.release();
        waiter.P();
        conditionLock.acquire();
    }

    public void wake() {
        // Lib.assertTrue(conditionLock.isHeldByCurrentThread());
        if (!waitQueue.isEmpty())
        ((Semaphore) waitQueue.removeFirst()).V();
    }
}
```

Is this still a correct implementation of condition variables? If so, explain why. If not, explain why not. More specifically, suppose we have 2 threads. The first one can context switch right before calling `P()` to the second thread that calls `wake()`. Can this signal still be caught by the first thread? What about multiple calls to `wake()` by the second thread before returning to the first thread?

d. **(12 points)** Consider the following interfaces for a lock and condition variable as given in Nachos:

```
public class Condition {
    public Condition(Lock lock) {
            ...
    }

    public void sleep() {...}

    public void wake() {...}

    public void wakeAll() {...}
}
```

```
public class Lock {
    public Lock() {...}

    public void acquire() {...}

    public void release() {...}

    public boolean isHeldByCurrentThread() {
            ...
    }
}
```

Show how to implement the `Semaphore` class (whose interface is shown below) using the `Lock` and `Condition` classes. Assume mesa-style scheduling. Make sure to include any member variables in your implementation.

```
public class Semaphore {
    public Semaphore(int initialValue) {
        ...
    }

    public void P() {
        ...
    }

    public void V() {
        ...
    }
}
```

4. (32 points) Here is a table of processes and their associated priorities (higher the number, higher the priority) and running times (in seconds). You may assume all processes arrive in the order listed a millisecond apart and scheduling decision is made after all processes arrive.

Process ID	Priority	CPU Running Time
P1	3	4
P2	2	6
P3	1	2
P4	4	10
P5	5	8

For each of the following policies provide the scheduling order, waiting time for each process, the average waiting time, completion/response time for each process and the average completion/response time **(2+2.5+0.5+2.5+0.5 = 8 points each)**:

- **a.** First Come First Serve (FCFS)
- **b.** Shortest-Job-First (SJF)
- **c.** Round-Robin (RR) with timeslice quantum = 1 second
- **d.** Priority

5. (20 points) Suppose that we have the following resources: A, B, C, and D and threads: T1, T2, T3, T4, T5. The total number of each resource is as follows:

Total			
A	**B**	**C**	**D**
6	12	7	12

Further assume that the threads have the following current allocations and maximum needs:

	Current allocation	Maximum Need

Thread ID	A	B	C	D	A	B	C	D
T1	2	0	0	0	2	5	7	0
T2	0	3	0	4	6	5	6	6
T3	0	1	0	2	0	1	0	2
T4	0	3	3	2	0	5	6	2
T5	2	5	3	4	4	5	3	6

a. **(6 points)** What is the available vector for each resource and the need matrix for each thread. Note that an available vector depicts the quantities of resources of each type that are available in the system after current allocation. The need matrix depicts of the quantities of resources of each type still needed by each thread, with each column corresponding to a thread and each row corresponding to a resource and an entry in location [i, j] of the matrix showing the quantity of a resource r_j still needed by a thread t_i.

b. **(7 points)** Is this system currently deadlocked (i.e., unsafe), or can any thread become deadlocked? Why or why not? If not deadlocked, give one possible execution order.

c. **(7 points)** If a request from T2 arrives for (0, 0, 1, 0), can that request be safely granted immediately? In what state (deadlocked, safe, or unsafe) would immediately granting the whole request leave the system? Which threads, if any, are or may become deadlocked if this whole request is granted immediately?