Problem 1: (16 points; 10 minutes) Explain any four of the following

1. Mutual exclusion: is a concept in concurrency that refers to having one thread in a critical region at a time. If we enforce mutual exclusion, the operations in a critical region appear to be atomic.

2. Spinlock: Is a type of lock where the lock acquire operation involves a thread spinning (i.e., busy waiting) while it repeatedly checks the state of the lock. Spinlocks may waste CPU time if the critical region is long (especially on single processor systems).

3. Kernel level thread: KLTs are threads supported directly by the OS (as opposed to user level threads which are supported using a user level thread library). It has advantages over ULTs in that if one thread does a blocking operation, other threads can continue execution (this is not true for ULTs). However, one drawback is that it is more expensive to do thread operations. ULTs also can exploit application information to improve thread scheduling.

4. Mode switch: A switch from user mode to the kernel or vice versa. It involves changing the mode bit from user to OS or back. A mode switch occurs when there is an exception that causes us to trap to the OS, or back to the user when the OS returns to the user using an instruction such as iret or sysexit.

5. Unsafe state (in deadlock avoidance): A state where we cannot guarantee that deadlock cannot occur. An unsafe state is however not a deadlocked state. More precisely, in an unsafe state, there is no sequence that if all processes make a request for the maximum resources they declared that would allow all the processes to finish without deadlock.

Problem 2: (16 points; 10 minutes):
(a) List the four ingredients necessary for deadlock to occur?

- Mutual Exclusion
- Hold and wait
- No preemption
- Circular dependency
(b) Show a resource allocation graph with 3 processes and 3 resources (could be of different types) that has deadlock. Show the corresponding WFG graph. RAG:

```
+--->R1--->P1--->R2--->P2--->R3--->P3--+-
  |                                           |
+-------------------------------------------+
```

WFG:

```
+--->P1--->P2--->P3--+-
  |               |
+---------------------+
```

**Problem 3:** (20 pts; 20 minutes): An OS uses a multiple level feedback scheduler with 3 round-robin levels. The quantum for the three levels are 2, 8 and 16 time units respectively. Assume that we have 6 jobs that arrive at intervals of 4 msec starting at time 0. The processes have a burst lengths 32, 12, 2, 3, and 20.

Show the scheduling timeline for the processes until the end. Track the state of the queues.

<table>
<thead>
<tr>
<th>P</th>
<th>Arrive</th>
<th>Length</th>
<th>Finished (after solving)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>32</td>
<td>69</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>12</td>
<td>53</td>
</tr>
<tr>
<td>P3</td>
<td>8</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>P4</td>
<td>12</td>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td>P5</td>
<td>16</td>
<td>20</td>
<td>63</td>
</tr>
</tbody>
</table>

Timeline:

P1 (1-10) | P2 (11-12) | P3 (13-14) finished | P4 (15-16) | P5 (17,18) | P2 (19,26) |

P4 (27) finished | P5 (28-35) | P1 (36-51) | P2 (52-53) finished | P5 (54-63) finished | P1 (64-69) finished |

Explanation:

P1 (Only process at the beginning in Q1, gets slots 1 and 2, moves down to queue 2 (Q2) and gets slots 3-10). Moves down to Q3 with 22 units left.

Meanwhile P2 (arrives 4) and P3 (arrives 8) are in Q1 by time 11, we schedule P2 which gets slots 11 and 12 then moves to Q2 with 10 units left.

At 12, P4 arrives and is queued in Q1 behind P3. P3 runs 13 and 14 and is finished.

At 15, P4 runs for 15 and 16 and moves down to Q2 with 1 left which now holds P2-¿P4. P5 arrives at 16 and is queued in Q1.

At 17, P5 runs for 17 and 18. It moves down to Q2 behind P4 with 18 left. The state of the queues at this point:

Q1 (empty)
Q2 P2(10)--->P4(1)--->P5(18)
Q3 P1(22)

Next, P2 runs 8 units (19 to 26) and moves to Q3, followed by P4 for 1 unit (27) and finished and P5 for 8 units (28 to 35) and down to Q3. At this point the queues have:

Q1 (empty)
Q2 (empty)
Q3 P1(22)--->P2(2)--->P5(10)
We round robin between them with time slice 16, so P1 (36 to 51) followed by P2 (52 to 53 and finished) P5 (54 to 63 and finished) then P1 again (64 to 69 and finished).

(b) Pick a metric (such as response time, wait time, etc...) to show that the schedule is better than FCFS.

FCFS schedule:


Here you have several options.

A main advantage of MLQ and any preemptive scheduler over FCFS is response time. For FCFS response time is the wait time (or time it starts running minus arrival time):

P1 0, P2 (33-4), P3 (45-8), P4 (47-12) and P5 (50-16). Average is 0+29+37+35+34/5 = 27.

Computing response time is a little more tricky for MLQ. It is the average of each period of waiting.

For: P1 it is (0 + 25 + 28)/3 = 17.6666
P2 (7 + 6 + 25)/3 = 12.666
P3 (5/1) = 5
P4 (3 + 10)/2 = 6.5
P5 (1 + 9 + 18)/3 = 9.3333

Average response time is significantly lower (in fact, even the worst process P1 has a response time lower than the average process for FCFS).

Partial credit policy for this problem: If you name a valid metric (25% of credit). If you name response time (50% of credit). If you compute correctly for any metric (60% credit). If you compute correctly (or nearly so – off by one mistakes are ok) for response time 100% of credit.
**Problem 4:** (20+ 5 bonus pts; 15 minutes) Consider the following C code.

```c
int main() {
    int count = 1, pid1=1, pid2=1;

    pid1 = fork();

    if(pid1==0) {
        pid2 = fork();
        count++;
    }

    if(pid2 == 0) {
        count++;
        fork();
    }

    //For part C, consider what would happen if there is a statement here. wait();
    printf("%d \n",count);
}
```

(a) How many total different processes will be created by the above program? 
four total processes (i.e., 3 additional processes created). Only the child will go into the first if statement. Only the child of the child will go into the second if statement. See process tree for details.

(b) Draw the process tree representing the above program execution including outputs and list two possible outputs (just the numbers that are printed in order; ignore the formatting)

```
+---CCC--3
  |
  +----CC--+---CCP--3
  |
  +-----+----CP--2---------
  |
  -----+----P----1--------------
```

Possible outputs are any permutation of 1 2 3 3.

(c) (Bonus 5 points) Assume that the comment in the line before last in the program is replaced by a wait statement. Recall that wait causes a process to wait until any one of its children exits. If a process does not have children, or if a child already exited, it does not wait. **Give one output that is possible without the wait that would not be possible with the wait.** In other words, the solution output is possible in the original code, but not possible when the wait is inserted.

In this case, every process will wait for a child if there is one. So, P will wait for CP, CP will wait for CC, and CC will wait for CCC. The output will be 3 3 2 1. Any other output will not be possible.
Problem 5: (28 pts; 25 minutes)
Consider the following situations:

(a) (6 points) Thread A has two parts, ThreadA1() and ThreadA2(). ThreadA1 has to happen before ThreadB() (executed by another thread B), while ThreadA2() has to happen after ThreadB(). Show the implementation

For these problems, pseudocode is enough. Partial credit if you describe the idea but do not have a clear pseudocode implementation. You should clearly specify the type of synchronization primitive you are using (semaphore and its initial state, lock, etc...) but we are not particular about the syntax (e.g., lock.acquire() or acquire(lock) or lock.wait(), etc... are all fine)

Semaphores A and B both initialized to 0;

```java
threadA() {
    threadB() {
        ThreadA1();
        B.wait(); //wait for A to finish first part
        B.signal();
        ThreadB();
        A.wait();
        A.signal(); //tell A to go ahead
        ThreadA2();
    }
}
```

(b) (10 points) One thread repeatedly does three operations A, B and C in that order. Another thread does A, C then B. All operations use shared variables and therefore use three locks (LockA, LockB and LockC) before each operation. Show if deadlock can occur.

Locks LockA, LockB and LockC; //Locks, but semaphores initialized to 1 are also acceptable

First thread:

```java
LockA.acquire();
A();
LockA.release();
LockB.acquire();
B();
LockB.release();
LockC.acquire();
C();
LockC.release();
```

Second thread:

```java
LockA.acquire();
A();
LockA.release();
LockC.acquire();
C();
LockC.release();
LockB.acquire();
B();
LockB.release();
```

Deadlock cannot occur since each thread holds only one lock at a time.

If you assume that the locks are all acquired at the beginning and released at the end, deadlock can occur since B and C are acquired in different orders in the two threads. So, thread 1 can have B and be waiting for C, while thread two can have C and be waiting for B. We’re willing to accept this as an alternative solution (80% of the credit) if you clearly show the lock acquire and release order and how deadlock occurs.
(c) (12 points) You’ve just been hired by Mother Nature to help her out with the chemical reaction to form water, which she doesn’t seem to be able to get right due to synchronization problems. The trick is to get two Hydrogen (H) atoms and one Oxygen (O) atom together at the same time to produce H2O. The atoms are threads. Each H atom invokes a procedure hReady when it is ready to react, and each O atom invokes a procedure oReady when it’s ready. When the oxygen atom has two hydrogen atoms available, it increments the number of total water molecules made. Show the implementation of hReady and OReady. Think about using a counting semaphore for each of the hydrogen and oxygen atoms.

Semaphore h,o; //semaphores initialized to 0.
Lock mutex; //lock, or alternatively semaphore initialized to 1

hReady() {
    o.signal();
    h.signal();
}
oReady() {
    h.wait();
    o.wait();
    h.wait();
    o.signal();
    mutex.acquire();
    water_molecules++;
    mutex.release();
}

Hydrogens signal their presence, and wait for an O to grab them. Oxygens wait for two hydrogens, signals them that water is made so that they are done. It then grabs a mutual exclusion lock (since their could be other oxygens that make water at the same time) and increment the molecule count.