Problem 1: (20 points; 10 minutes) Indicate whether each of the following statements is true or false: Then use exactly one sentence to describe why you chose your answer. Without stating the precise reasoning, you will not get any points.

(T) (F) Faults and interrupts are synchronous and unexpected.

False, Faults are synchronous (caused in response to an instruction being executed and depending only on the state of the CPU), while interrupts are asynchronous. Both are unexpected.

(T) (F) existence of a loop in WFG graph guarantees deadlock existence.

True, Waits-for Graph; an edge from a process, P1, to another process, P2, represent that P1 is waiting for a resource that belongs to P2. So, if we have a loop in WFG, it means that there is that cycle of dependencies between processes in that loop. Therefore, this is deadlock.

(T) (F) The system call "wait" always lead to a process moving to the wait state

False, While typically wait causes a parent to wait on a child, if it has no children or the child has already exited it does not move to wait.

(T) (F) Threads in the same process share the same program counter

False, Program Counter (PC) is part of the execution state and therefore each thread gets a separate PC.

(T) (F) Shortest Job first scheduling is less responsive than Round robin

True, SJF is non preemptive and those are less responsive than preemptive algorithms such as round robin. Responsiveness (or response time) is the average time waiting for a process every time it stops running. With non-preemptive algorithms if it arrives after a long process starts execution, it can wait a long time.

Problem 2: (5 points, 5 minute) choose all the options that is true for each statement. Remember there are possibly MULTIPLE choices to each question. Choosing wrong answer will have 1 negative point.

(a) Suppose that a process is executing “counter=counter+1” while another process is executing concurrently and independently “counter=counter-2”, where the counter is a variable shared between the two processes and is accessed only by the two statements. Given that the value of counter is five before execution, the possible value(s) after both processes finish their statement are ...........................................[ ]

It is either no updates lost or update of one of the threads is lost by the other thread’s update.
• If no updates lost; counter will be: 5+1-2=4
• If the update of the first thread lost: 5-2 = 3
• If the update of the second thread lost: 5+1 = 6

i. Three.
ii. Four.
iii. Five
iv. Six.

(b) The following algorithm is proposed to solve the critical section problem between two processes $P_1$ and $P_2$, where lock is a shared variable.

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\textbf{do}$</td>
<td>$\textbf{do}$</td>
</tr>
<tr>
<td>$\textbf{while}$ (lock) { NULL;}</td>
<td>$\textbf{while}$ (lock) { NULL;}</td>
</tr>
<tr>
<td>lock = TRUE;</td>
<td>lock = TRUE;</td>
</tr>
<tr>
<td>critical section;</td>
<td>critical section;</td>
</tr>
<tr>
<td>lock=FALSE;</td>
<td>lock=FALSE;</td>
</tr>
<tr>
<td>reminder section;</td>
<td>reminder section;</td>
</tr>
<tr>
<td>$\textbf{while}$(TRUE);</td>
<td>$\textbf{while}$(TRUE);</td>
</tr>
</tbody>
</table>

Which of the following statements is true regarding the proposed algorithm? .......... [ ]

i. Mutual exclusion to the critical section is guaranteed.
ii. Both processes can be in their critical section at the same time.
iii. Lock should be initialized to TRUE.
iv. When a process is waiting on “while(lock){NULL}” it will move to the waiting state.

Since the lock itself is global variable and shared, there is also a race condition for the lock itself. So, if $P_1$ receive first and get the lock and before assign TRUE, $P_2$ gets running and sees the lock is still FALSE and will go into the Critical Section as well; So Mutual exclusion is not guaranteed and both $P_1$ and $P_2$ can be in the Critical Section.

If the lock be initialized to TRUE, no one can go in to the critical section and it doesn’t work.

The last statement is also false since while(lock){NULL} is a non-blocking spinlock so it does not move the waiting process to the waiting state.
Problem 3: (20 points; 15 minutes): Consider the following scenario:
Process P1 owns resources R1, R2 and requests for R3. Process P2 owns process R4 and requests for R5 while R5 along with R3 belong to process P3 requesting for R1.

(a) (5 points) Draw the Resource Allocation Graphs for the above scenario.

(b) (5 points) Draw the corresponding waits-for Graph

(c) (5 points) Indicate whether there is a deadlock or not? If yes indicate whether adding another copy of R5 will resolve the deadlock, why?

Yes, a deadlock is there since there is loop in the WFG which guarantees that there is cyclic dependencies between processes P1 and P3.

Adding another copy of R5 does not solve the deadlock. It only allows the P2 to be able to finish which is not part of the deadlock.

(d) (5 points) Explain how preventing “hold and wait” condition may have been used to prevent deadlock.

The cycle of dependencies is made when processes are allowed to wait for another resource while they already hold some other; by not allowing the processes to hold any resource while they wait on the other one, basically P1 must use or release R1 and R2 before it is able to waits on R3. And, the same thing is true for P3. So, this prevents the cycle in the WFG from happening.
Problem 4: (20 points; 25 minutes) Assume that the following processes are the only processes in a computer system and that there are no input/output requests from all the given processes. Given the following arrival time and CPU burst time (also called service time) for each process, answer the following questions.

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>CPU Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>P₃</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>P₄</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

(a) (15 points) Draw the timeline for the following and compute the average waiting time for each by completing the following table. (Waiting time = turnaround time – burst time)

i. First Come First Serve (FCFS) (aka. FIFO)

ii. Shortest Job First (SJF)

iii. Round Robin with the quantum of 4 (Assume the new process will be inserted at the end of the ready list)

(b) (5 points) According to your results, which scheduling algorithm among those given in (a) gives the shortest average waiting time? Is that consistent with your theoretical prediction? Justify your answer.

We expected that SJF gives us the shortest average waiting time since it always tries to pick the smallest one, but in this case, RR gave us the shortest average waiting time. This is because of the arrival times; if all arrived at the same time then SJF would give us the best average for the waiting time.
Problem 5: (20 points; 15 minutes) Recall the code for the bounded buffer synchronization problem shown below for producer and with a buffer that can hold two items.

1. Semaphore empty(2), full(0), mutex(1); //number indicates initial value
2.
3. producer {
4.   while (1) {
5.     Produce resource; wait(full); 
6.     wait(empty); wait(mutex);
7.     wait(mutex); Get resource;
8.     Add resource; signal(mutex);
9.     signal(mutex); signal(empty);
10.    signal(full); Consume resource;
11.  }
12. }
3. consumer {
4.   while(1) {
5.     wait(full); 
6.     wait(mutex);
7.     Get resource;
8.     signal(mutex);
9.     signal(empty); 
10.    signal(full);
11.  }
12. }

(a) (5 points) Explain what happens when a consumer arrives first, and then a producer arrives later. Be specific and refer to the code lines.

Consumer waits on full and gets blocked since full is initialized to 0. Next producer arrives and waits on empty, decrementing it to 1. It takes the mutex semaphore, adds resource to the queue, signals mutex (line 9) and then signals full to indicate that there is a full slot in the buffer for a consumer to consume. Signaling full unblocks the consumer waiting on line 5 of the consumer code. The consumer acquires the mutex, gets the resource (line 7), signals the mutex and signals the empty semaphore indicating that there is an additional empty slot (increasing that semaphores count to 2).

(b) (5 points) What happens if we move line 5 in the producer to be after the current line 7? Which version is better?

The code will still run correctly. However, pushing the work to produce the item into the critical region causes it to become longer reducing concurrency (it is a more conservative approach). We would like to keep the critical region as small as possible while maintaining correctness.

(c) (5 points) If the code was initialized by mistake to have empty start as 1 instead of 2, what is the effect?

If empty is initialized to 1, only one slot in the buffer would be used. Once a producer goes in and consumes the empty token decreasing it to zero, an additional producer cannot go in until a consumer removes that item. Effectively the buffer becomes of size 1 instead of 2.

(d) (5 points) If the code was initialized by mistake to have full start as 1 instead of 0, what is the effect?

In this case, this is a correctness issue. If full is initialized to 1, a consumer is able to go in even when the buffer is empty causing errors or deadlock (if the consumer is stuck inside the critical region because it cannot find an item to remove).
Problem 6: (15 points; 10 minutes) Consider the following code:

```c
Main()
    int pid1=0, pid2=0;
    pid1 = fork();
    pid2 = fork();
    if(pid1 == 0 && pid2 == 0){
        --------------wait(status); // wait added only for part c
        print("A\n");
    }
    if(pid1 == 0)
        print("B\n");
}
```

(a) (3 points) How many processes will exist when we run the program?

4 processes. See the chart.

(b) (6 points) what are all the possible outputs?

So, one A and two Bs are being printed. The only restriction is that A needs to come before at least one B. Because A and one of the Bs are printed by the same process (CC). so, the only outputs are:

i. A-B-B
ii. B-A-B

(c) (6 points) what are all the possible outputs if the wait statement is added as shown?

In this case, since the only thread that goes to if statement is the thread (CC) that does not have any children to wait so no effect will be made by "wait()" the output will be the same

i. A-B-B
ii. B-A-B