CS153: Midterm (Winter 15)

Name:

Answer all questions. State any assumptions clearly.

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

1. Faults: **Faults occur when an instruction execution results in an error that has to be handled by the OS. Examples of faults include divide by zero, a memory permission or alignment fault, or a page fault.**

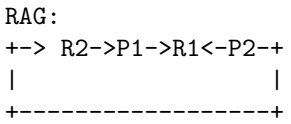
2. Livelock: **A situation similar to deadlock that occurs when multiple threads have the illusion of progress but are actually stuck for an extended period of time. Commonly this occurs when threads detect contention and back off to retry later only to find that contention still exists. Consider a variant of the dining philosophers where the philosophers detect take one chop stick, then release it if they figure out the other is not available. The philosophers can be stuck picking up then releasing the chopstick forever if the timing is unlucky.**

3. System call: **this is the mechanism whereby user programs ask for services from the OS. Any operation that accesses hardware resources, for example, must be protected by the OS, and exposed to the users only through system calls. A system call causes a mode switch to the OS.**

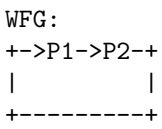
4. Hold and Wait: **One of the four prerequisites for deadlock–a thread holds some resources and waits for others (rather than releasing the resources it is holding or giving up on waiting).**

5. Atomic: **An operation or a group of operations that is atomic is guaranteed to all take effect without interference from any other threads. Atomicity can be guaranteed by the hardware or through synchronization mechanisms such as mutexes.**

Problem 2: (14 points; 10 minutes): Show a resource allocation graph with 2 processes that has deadlock. Show the corresponding WFG graph.



P1 has R2 and is waiting for R1 P2 has R1 and is waiting for R2



P1 waits for P2 and vice versa.

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 1, 2 and 4 time units respectively. Assume that we have 6 jobs that arrive at time 0 with burst lengths 8, 2, 1, 3, 10 and 4 in the order of arrival.

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

```
At time 0, J1(8)->J2(2)->J3(1)->J4(3)->J5(10)->J6(4)

J1 runs for 1 time unit, has 7 remaining and demoted to level 2.  This repeats for each of the jobs

J1|J2|J3|J4|J5|J6

At time 6, we have:

L1: empty
L2: J1(7)->J2(1)->J4(2)->J5(9)->J6(3)

Now, each job runs for up to 2 (J2 uses 1 and finishes).  So, from time 6 schedule continues as:

J1J1|J2|J4J4|J5J5|J6J6

We are now at time 15, with the following state:
L1: empty
L2: empty
L3: J1(5)->J5(7)->J6(1)

Finally, we do round robin with quantum 4 on L3.  J1 runs for 4 (1 remaining); J5 runs for 4 (3 rema

Putting it all together, the schedule is:\\
J1|J2|J3|J4|J5|J6|J1J1|J2|J4J4|J5J5|J6J6|J1J1J1J1|J5J5J5J5|J6|J1|J5J5J5
```

(b) Consider the following policy called selfish round robin (SRR): we delay a new process until it catches up in priority with existing processes. In some ways, SRR seems to be the opposite of MLF which gives higher priority to newly arriving jobs. What are their advantages and disadvantages relative to each other?

Any reasonable discussion would work here. Delaying newly arriving jobs double penalizes I/O bound processes. They do not run for their full quanta, and they have to wait to run everytime they re-enter the system. This can be fixed by remembering their priority across waits.

On the other hand, we avoid giving new processes an advantage over older ones, enabling these older processes to finish faster (helping their turn-around time).

Processes are delayed at the beginning but after they start running, their response time is likely to be better since the system delays newly arrived processes for a while.

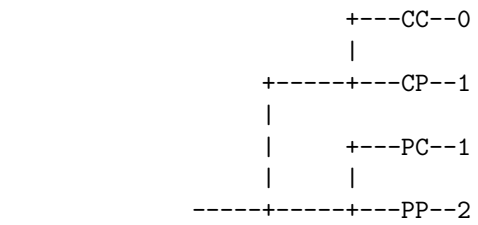
etc...Looking to see if you can think around the different goals of the scheduling process.

Problem 4: (25 pts; 15 minutes) Consider the following C code.

```
int main() {
int count = 0;

if(fork()) {
   count++;
}

if(fork()) {
   count++;
}

//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? **4 processes**. P forks C at the first fork. They each fork a child at the second fork.

(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)



There are no dependencies between the prints. Any permutation of the outputs (0 1 1 2) are possible.

(c) Assume that the comment in the line before last in the program is repaced 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.

With the wait, any process that is the parent of a fork, will wait for any of its children to finish first. So, PP will wait for either PC or CP to finish. Likewise, CP will wait for CC to finish.

Lets look at the first condition. PP outputs 2, so that must happen after either PC or CP finish – both output 1. So, the 2 must happen after at least one of the 1s. Therefore, the output 2011, 0211 are illegal in the second code, but legal in the first.

Likewise, if we look at the second condition (CP waiting for CC to finish): CP outputs one of the 1s, and CC outputs 0. So, one of the ones has to happen after the 0. Thefore, outputs 1102, 1120, 2110, 1210, 2110 are illegal in the second code but legal in the first.

Problem 5: (30 pts; 25 minutes)
A (very small) bus goes through a route between 2 stops. The bus has room for only 1 passenger. At every station, it lets a passenger down, then lets a passenger on, before heading back to the other station. We are building a simulation of this situation with each passenger being a thread arriving at station 0 or 1 (parameter to the thread), and the bus being a thread.
(a) Given the passenger implementation below, write the bus implementation. You may assume the bus already has a passenger on and is about to arrive at a station. Note that the bus should be interacting with the passenger using the same semaphores to make sure the desired behavior occurs (arrive at station, passenger gets down, new passenger gets up ...).

```
//code for a passenger arriving at station i (0 or 1).  Each passenger is a thread

Passenger(i) {
StationSem[i].Down();  //wait for bus to arrive
BusReady.Up();  //Let bus know passenger is on
RideBus();
BusSem.Down(); //wait on bus to get to next station
Arrive();
StationSem[1-i].Up(); //Let a passenger on
}

Bus {
while(1) { //each of these is a trip

BusReady.Down(); //wait for passenger to be on
DriveToNextStation(); //go to station 1-station
BusSem.Up(); //tell passenger we got there
 }
}
```

(b) What should the different semaphores be initialized to? **All semaphores initialized to 0**.

(c) Can the bus leave before a passenger gets on or a passenger leaves? Explain. **No, The first semaphore BusReady ensures that the bus waits for the new passenger to get on before leaving. This new passenger is only let on by the current passenger, who first exits the bus when it gets there (by waiting on BusSem), before letting the new passenger on (by signaling StationSem[1-i]).**

(d) What happens if there are no passengers at the station when a bus arrives? Explain briefly but clearly in words how you could modify the implementation to improve this behavior.

Bus is stuck waiting on BusReady. To avoid this behavior the code needs to be substantially modified to keep track of the number of waiting passengers, and the number of passengers on the bus. We signal a passenger on and wait for them only if there is one waiting. Write this code as an exercise!