CS153: Midterm (Fall 17) Key

Name:
Student ID:

Answer all questions. State any assumptions clearly.

Problem 1: (18 points; 10 minutes) For each problem, check all statements that are true (there may be none).

1. Which of the following are true about scheduling

   (T) (F) FIFO does NOT achieve good response time

   True – as a non-preemptive policy, response time is generally terrible.

   (T) **(F)** Round robin achieves good turnaround time

   False: round robin is good for responsiveness, but generally delays the end time of the processes; rather than focusing on one process and finishing it, we keep switching between them, and they all finish late
   (T) **(F)** Multi-level feedback requires that we know the run time of each process
   False: we simply let processes work through our queues and no estimate of how long they run is ever needed or used
   (T) **(F)** Scheduling decides which waiting process to unblock first
   False: it decides which ready process to run next.

2. Threads and processes were two abstractions used in an OS.

   (T) **(F)** A process needs its own stack, but not a thread
   A stack is part of the dynamic state of a thread; each thread needs its own stack. If they share a stack, other than that being a race condition, two threads may call two different functions messing up the state of the stack for both (they would be both working in the same stack frame)
   (T) (F) Its more expensive to switch processes than to switch threads

   True: switching processes requires us to switch the memory management registers and state, flush the TLBs, etc. Switching threads is significantly more light weight because only the execution state is switched, not the resource state.
   (T) (F) A process has its own address space (memory) but not a thread

   T **(F)** Both threads and processes share global variables but not local variables
   Threads share the same address space of the process they belong to, but processes have separate address space. Threads share global variables, but processes dont.

3. Which is true about the following operations.

   (T) (F) I/O requires a system call
   True – I/O cannot be exposed directly to user processes and they must go through the OS using a system call to access it.
   (T) **(F)** A fault may or may not cause a trap to the OS

False – a fault causes a trap to the OS.

(T) **(F)** A call to a shared library requires a system call

False: library code, even dynamic library code can be linked and used through function calls without requiring system calls. The library code itself is read only and the functions can be written to be re-entrant/thread safe so that there is no interference between different programs using the same shared library.

(T) (F) A process in the ready state cannot move directly to the waiting state

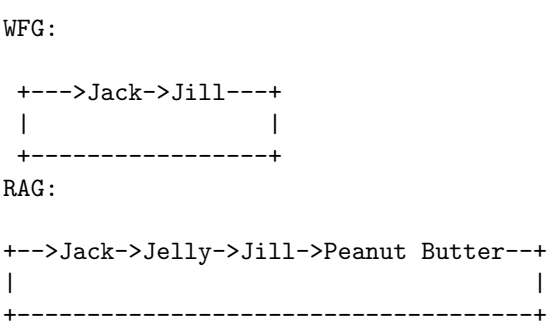
Generally true unless your OS is doing something weird. To move to the wait state your process should execute something that causes it to wait. It is not possible to execute while in ready – you have to be running.

<u>*Problem 2:*</u> (18 points; 10 minutes): Jack and Jill both want to make a sandwich. They need peanut butter and jelly. Jack gets the peanut butter first, but Jill gets the Jelly first and they are now waiting for each other.

(a) (8 points) Show using the four ingredients that this situation is deadlock. (deadlock ingredients not food ingredients!)

1. *Mutual exclusion: only one person can have the peanut butter or jelly at a time.*
2. *No preemption: we do not let go of the peanut better or jelly once we have them (and we dont steal them from each other*
3. *Hold and wait: each of us is holding a jar and waiting for the other jar*
4. *Circular wait: Jack is waiting for Jill and Jill is waiting for Jack – circle*

(b) (5 points) Show the Waiting For Graph for part (a)



You should partial credit if you provide the RAG instead of WFG (-1 point)

(c) (5 points) Suggest a solution based on preventing the circular wait ingredient for part (a)

To prevent circular wait, we have to ensure that the resources cannot form a circle. If we follow the approach we discussed in the class, we make it so PB and Jelly are in two different resource classes and we always compete for them in the same order. So, they both compete say for PB first, and only one of them gets it and the other is waiting. The person that won the PB, will then win the Jelly (if the Jelly is not available, the person holding it will never ask for the PB because of the resource order constraint and deadlock is not possible).

If you suggested get both at the same time, you should get 3/5 points credit since that prevents hold and wait rather than the circular dependency.

Problem 3: (20 pts; 15 minutes): An OS uses a multiple level feedback scheduler with 2 round-robin levels. The quantum for the two levels are 1 and 8 respectively. Assume that we have 5 jobs that arrive at intervals of 5 msec starting at time 0. Assume that once a quantum starts, it runs until it ends (or the process finishes); it is not interrupted by a newly arriving process. The processes have a burst lengths 17, 5, 1, 11, and 8.

(a) (15 points) Show the scheduling timeline for the processes until the end. Track the state of the queues.

P1 arrives 0, length 17; P2 arrives 5, length 5; P3 arrives 10, length 1; P4 arrives 15, length 11; P5 arrives 20, length 8.
P1 runs 0-1, moves to lower queue with 16 left
P1 runs 1-9, has 8 left; P2 arrives at 5
P2 runs 9-10, moves to lower queue with 4 remaining (now has P1–P2); P3 arrives 10.
P3 runs 10-11, finishes.
P1 runs 11-19, finishes; P4 arrives at 15.
P4 runs 19-20, has 10 left, moves to lower queue which now has P2–P4; P5 arrives
P5 runs 20-21, has 7 left, moves to lower queue which now has P2–P4–P5
P2 runs 21-25, finishes
P4 runs 25-33, has 2 left
P5 runs 33-40, finishes
P4 runs 40-42, finishes

(b) (5 points) What metric would you use to show that this scheduler is better than Shortest Job First? Explain.

SJF is not preemptive, so we should be thinking response time. If you argued from this perspective but did not demonstrate, you should get 4/5
SJF schedule would be:
P1 0-17; meanwhile P2 arrives 5, P3 arrives 10, P4 arrives 15. Pick shortest (P3).
P3 runs 17-18. Pick shrotest among P2 and P4 (P2)
P2 runs 18-23. Meanwhile P5 arrives. Pick shortest between P4 and P5 (P5).
P5 runs 23-31. Pick P4 (last remaining)
P4 runs 31-42.

Response time is very high here. P1 =0, P2 = 13 (starts executing 18, arrived 5), P3 = 7, P4 =16, P5 = 3; average = 39/5 or 7.8.

For Multi-level feedback, the initial response time is much shorter (P1 =0, P2 =4, P3 = 0, P4 = 4, P5 =0) average 1.3. Average response time is also short (0 + 4 + 0 + 2 + 4 + 0 + 11 + 5 + 12 + 7)/10 = 45/10 or 4.5. The average response time is computed by checking every time we start a new process running how long it has been since it ran last (or arrived if this is the first time it runs) and average this out by the number of these events.

Problem 4: (20 pts; 15 minutes) Consider the following C code.

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

pid1 = fork();

if(pid1==0) {
    pid2 = fork();
    printf("%d \n", count);
    count++;}

if(pid2 == 0) {
```

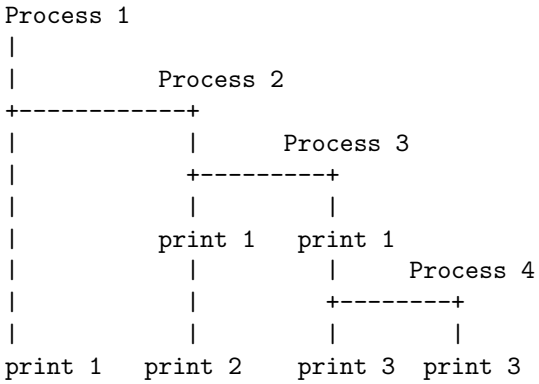
```
    count++;
    fork();}

//For part C, consider what would happen if the following statement is uncommented:
// if(pid1) waitpid(pid1, status, NULL);

printf("%d \n",count);
```

(a) (5 points) How many total different processes will be created by the above program? 4 total processes

as can be seen below. Each fork is executed by exactly one process, adding another process to the total.
(b) (10 points) Draw the process tree representing the above program execution including outputs. Explain the possible outputs and list two examples (just the numbers that are printed in order; ignore the formatting)



Three 1's a 2 and two 3's will be printed. The two will follow at least one of the 1's due to sequential execution of print 1 and print 2 in process 2. Both 3's must follow at least one 1 since 1 is printed in process 3 before it forks 4 and both of them print 3. Note that this 1 has to be different than the 1 preceeding the 2 (i.e., 1 2 3 3 1 1 cannot happen since the first 1 is either the one in process 2 preceeding the print of 2, or the 1 in process 3 preceeding the 3s – it cannot be both). Any permutation that preserves these conditions works. For example:

1 2 1 3 3 1, 1 3 1 3 1 2, 1 2 1 3 1 3, 1 1 1 3 2 3, 1 1 2 3 1 3

and many others. You need to list only two correct outputs to get full credit.

(c) (5 points) Assume that the commented code marked for part C is uncommented. **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.

Note that Only P1 has pid1 not equal to zero. It waits for Process 2 to finish. Therefore, there has to be at least a 1 that prints after process 2 finishes and prints its 2.

Therefore, any output that does not have a 1 after the 2 is not possible with the wait. For example, from the outputs above valid without the wait, 1 3 1 3 1 2, and 1 1 1 3 2 3 are both no longer possible.

Problem 5: (24 pts +3 bonus; 20 minutes)
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.

You are given the implementation of oReady and hReady below.

```
int numHydrogen = 0;
Semaphore pairOfHydrogen(0); //initialized to 0
Semaphore oxygen(0); //initialized to 0

void hReady() {
  numHydrogen ++;
  if ((numHydrogen % 2) == 0) {
  pairOfHydrogen.signal();
}
oxygen.wait();
}
```

```
void oReady() {
    pairOfHyrdrogen.wait();
    makeWater();
    oxygen.signal();
    oxygen.signal();
    }
```

(a) (8 points) Two hReady threads arrive one after the other, followed by an oReady. Assume the threads arrive such that only one thread executes at a time. Explain what will happen clearly, referring to the code.

The first hReady comes in, increments the numHydrogen counter, fails the if statement since numHydrogen is 1, then goes to wait on the oxygen semaphore.

The second hReady comes in, increments the numHydrogen counter, goes into the if statement since numHydrogen is 2, signalling the pairOfHydrogen semaphore incrementing its value to 1, then goes to wait on the oxygen semaphore.

oReady comes in, waits on pairOfHydrogen decrementing its value to 0, but not waiting. Makes water (all the ingredients are here!) then signals oxygen twice allowing the two waiting hydrogens to proceed. It works!

(b) (4 points) Briefly explain the opposite scenario where oReady arrives first.

oReady executes first, and blocks at the first semaphore wait since the value is 0. The hydrogens come after that as before, waiting on oxygen wait. The second hydrogen signals pairOfHydrogen, allowing the oxygen to go, make water, then signal oxygen twice to unblock the waiting hydrogens. It works again!

(d) (12 points+3) Even though the logic looks mostly fine, you realize that your code sometimes does not work correctly. Explain the problem with a specific example of what could go wrong and explain how you would fix it.

The problem is the race condition between the hydrogens on numHydrogen increment and check. Imagine we lose one update and both find numHydrogens = 1 not signalling the oxygen. Alternatively, numHydrogen can be incremented twice before either thread checks if it needs to signal. Since numHydrogen checks out to be 2 for both, they both signal oxygen, allowing another oxygen thread to make water without waiting for hydrogens.

The fix is to add a lock or a binary semaphore around the first two lines of hReady (including all of the if statement).