CS153: Midterm (Winter 21)

Name:
Student ID:

Answer all questions. State any assumptions clearly.

<u>Problem 1:</u> (20 points; 5 minutes) Indicate whether each of the following statements is true or false:

(T) (F) Handling an interrupt may result in moving a process from waiting to ready

An interrupt may be from I/O or another event for which a process was waiting. When the interrupt is handled, this can cause this waiting process to become ready again. Not that this is not the process that is currently running (which is not in the waiting state.

(T) (F) Hardware support prevents user code from directly accessing the OS

True. This is the privilege bit and the trap mechanism that makes sure it is set securely when an event causes a trap to the OS.

(T) (F) The system call wait does not always lead to a process moving to the wait state

True. 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 stack

Stack is part of the execution state and therefore each thread gets a separate stack.

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

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.

<u>Problem 2:</u> (20 points; 10 minutes):

(a) (15 points) Draw two Resource Allocation Graphs with two processes each such that one demonstrates deadlock and the other does not.

```
P1-->R1-->P2-->R2  (no deadlock)

P1-->R1-->P2-->R2---+
^                   |   (deadlock)
|                   |
+-------------------+
```

(b) (5 points) Explain how the banker's algorithm may have been used to prevent deadlock.

Bankers algorithm requires each process to make a maximum claim (or credit line) on resources ahead of time. From the deadlock figure, we can infer that both processes can claim both R1 and R2 so they each claim both. Lets say P1 starts execution first, and asks for R2. In this case, we can honor this request since this is a safe state: P1 can make maximum claim and ask for R1 and get it and be able to finish. Next, P2 asks for R1. However, if we honor this, the resulting state is unsafe: neither P1 nor P2 can finish if they make their maximum claim. The system denies P2 the ability to get R1, and deadlock is not possible (later P1 asks for R1 and is able to get it since that is a safe state. Finally it finishes and P2 is able to continue and get R1 and R2.)

Partial credit if you explain Banker's algorithm in general.

<u>*Problem 3:*</u> (34 pts; 20 minutes) Consider the following C code.

```
1. int main() {
2. int pid1=0, pid2=0, count=0;
3.
4. pid1=fork();
5.
6. if(pid1 == 0)
7.     {
8.     count ++;
9.     pid2=fork();
10.    } else {
11.    count = count + 3;
12.    pid2=fork();   }
13.
14. if (pid2 != 0)
15.      {
16.      if(pid1 == 0)
17.         printf("%d\n", count+1);
18.      else
19.         printf("%d\n", count + 2); }
20.
21. //For part C, consider what would happen if the following statement is uncommented:
22. // if(pid1 != 0) waitpid(pid1, status, NULL);
23.
24.    if(pid1 != 0)
25.         printf("%d \n",count + 3);
```

(a) (16 points) Draw the process tree representing the above program execution showing outputs. Explain the possible outputs and list two examples (just the numbers that are printed in order; ignore the formatting)

```
CC             +---------exit
               |
               |
               |
CP        +-----+-----2---exit
          |
PC        |     +---------6---exit
          |     |
          |     |
PP--------+-----+----5----6----exit
```

PP is the original process, it forks a child, CP at the first fork on line 4. The child goes into the if clause at line 6. It increments count to 1, and forks CC. PP goes into the else clause increasing count to 3, and forking PC.

At line 14, PP and CP have pid2 != 0 from the return value of the forks (at line 9 from CP and line 12 for PP). CP has pid1 == 0 so it prints count + 1 (which is 2 for it). PP goes to the else and prints count + 2 which is 5 for it since count got set to 3 on line 11.

Finally, all 4 processes reach line 24. At this point PP and PC have pid1 != 0, for PP it got it from the return value of the fork on line 4, while PC got a copy of it when PP forked it on line 12. Only they print count + 3 which is 6 for both of them.

Outputs are 2 6 5 6 in any order with the requirement that the 5 appears before at least one of the 6s. Possible outputs include 2 6 5 6, 2 5 6 6, 5 2 6 6, 5 6 2 6, etc...

(b) (6 points) How many total processes (including the original process) are created? *4 processes total (tagged as PP, PC, CP, and CC above).*

(c) (12 points) Assume that the last line of commented code with waitpid marked for part C is uncommented. Give one output that was possible in (a) that is no longer possible

pid1 != 0 at line 22 for PP and PC as was discussed in part (a). In this case, each do a waitpid on the value in pid1 which is the process id of CP. Waitpid can only wait on a child process so only PP will wait for CP to exit before printing its 6 (PC does not wait since CP is not its child; if you assume it does, then we will work with that assumption).

So, the 2 printed by CP has to appear before the 6 from PP. So, an output such as 5 6 6 2 is possible without the wait but not possible with the wait since PP printed the 6 before CP printed the 2 and finished.

Problem 4: (24 pts; 10 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 {                     consumer {
4.   while (1) {                     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. }                                }
```

(a) (6 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 2) and then signals full to indicate that there is a full slot in the buffer for a consumer to consume. Signalling 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 semphore indicating that there is an additional empty slot (increasing that semaphores count to 2).

(b) (6 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) (6 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) (6 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.