CS153: Midterm (Winter 19)

Name:
Student ID:

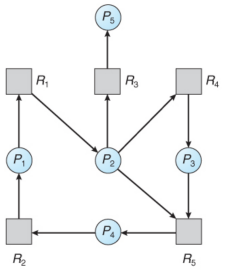
Answer all questions. State any assumptions clearly.

Problem 1: (24 points; 5 minutes) Indicate whether each of the following statements is true or false:
–(T)– (F) Handling an I/O interrupt may result in moving a process from waiting to ready
Yes, if the interrupt is from an I/O device completing I/O that a process is waiting on, it can move from waiting to ready.
–(T)– (F) Hardware support prevents user code from directly accessing the OS
Yes; without hardware support (e.g., mode bit and secure switching into the OS), users can directly bypass OS protections
–(T)– (F) The system call wait does not always lead to a process moving to the wait state
If the child already exited, or does not exit, no wait results.
(T) –(F)– Threads in the same process share the same register values

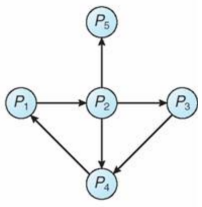
(T) –(F)– Round robin scheduling is less responsive than Shortest Job first

–(T)– (F) Shortest Job first is better than Round robin in terms of average turnaround time

(T) –(F)– Deadlock and livelock both require a circular dependency
Livelock does not require a circular dependencies – just a form of preemption.
–(T)– (F) Priority inversion can be addressed by the scheduler
Yes, tricks like priority inheritance are implemented in the scheduler



Problem 2: (20 points; 10 minutes):
(a) (15 points) Consider the Resource Allocation Graph in the figure. Draw the equivalent WFG. Identify any deadlock in the graph.



There are two cycles/deadlock situations in the WFG here. Identify either cycle P1,P2,P3,P4, or the smaller P1,P2,P4.

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

A general explanation of banker's algorithm should be ok here for 4 out 5 points. For the specific case you have to make some assumptions since we dont have the sequence of the requests. Note that since some processes are waiting for multiple resources, the requests must allow asking for multiple resources (otherwise they would each be waiting for just one resource).

We need each process to ahead of time identify the resources it may ask for in the future. In this case, at a minimum: P1 needs (R1 and R2), P2 needs (R1, R3, R4, and R5), P3 needs (R4 and R5), P4 needs (R2 and R5), and P5 needs (R3).

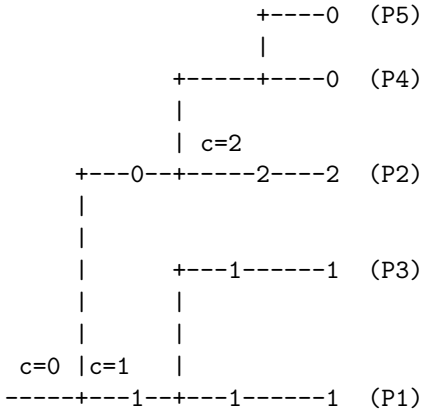
The requests that were successful occur first (otherwise the processes would block before they could ask for them). Lets assume they occur in the order of the processes to see where banker's algorithm would have interfered.

- **P1 requested R2 (allowed, state is safe)**
- **P2 requested R1 (allowed, state is safe. P1 and P4 cannot finish since R1 and R2 are not available, but the others can finish eventually unblocking them).**
- **P3 makes a request for R4 (allowed, P2 now cannot finish if it asks for everything, but P3 and P5 can eventually releasing resources and allowiing others to finish)**
- **P4 makes request for R5. This should be denied since it leads to an unsafe state. If its honored, then all the denied requests (on the RAG, arrow from process to resource) could be made and no process would be able to finish (other than P5), leaving the deadlock that we see.**

Problem 3: (34 pts; 20 minutes) Consider the following C code.

```
int main() {
int pid1=0, pid2=0, count=0;

if(pid1 = fork())
    count ++;

printf("%d\n", count);
pid2 = fork();

if((pid1==0) && (pid2!=0)
    count= count + 2;


if(pid1!=pid2) {
    printf("%d \n", count);
  } else
    fork();
  }
//For part C, consider what would happen if the following two statements are uncommented:
// if(pid1) waitpid(pid1, status, NULL);
// if(pid2) waitpid(pid2, status, NULL);  --corrected in exam

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

(a) (16 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)



This is a tricky problem! Note the following:

- **c refers to the variable count on the figure above (shortened for space).**
- **First if statement is executed only by P1 (P2 fails the check). Others have not been forked yet.**
- **The second if statement is executed only by P2 (P1 and P3 have pid1 not equal to zero, and P4 has pid2 equal to zero. P5 has not been forked yet.**

- **Third if statement is executed by P1 (pid1=P2, pid2=P3), P3 (pid1=P2, pid2=0), and P2 (pid1=0,pid2=P4). The else part is executed by P4 only, forking P5.**

There will be 3 0's, 5 1's and two 2's

Constraint 1: The first two will be preceded by at least 1 zero. There are a number of other orders between equal numbers that do not matter to the output since we cannot tell them apart. For example, the first 2 has to happen before the second 2, but since we cannot tell them apart, no difference in output is expected and we can ignore them.

There is no other ordering. Examples

0 0 0 2 2 1 1 1 1 1
0 2 1 2 1 0 1 0 1 1

Many others are possible (almost all permutations as long as the first two appears after a 0 etc...

(b) (6 points) How many total processes will execute the last printf? **5 processes**

(c) (12 points) Assume that the two lines of commented code with waitpids marked for part C are uncommented. What are the possible outputs now?

In this first line, the processes on the bottom part of the tree (P1, and P3) have pid1 not equal to zero. P1 has it after the fork, and P3 gets a copy once P1 forks it. They will all both for the first child (P2, which has the pid stored in pid1).

New Constraint 2 on the output: This forces the last two 1s (one from each of P1 and P3) to happen after the 0 2 2 from the first child.

In the second wait the processes that are the parents of the second fork P1 and P2 will wait for their children from their respective forks. Neither P1 or P2 fork after that, so they are the only processes that pid2 not equal to zero.

For P1, the child in its second fork whose pid is stored in pid2 is P3. P1 waits on it, but this just forces order on the 1s, and does not change the visible outputs; P1's 1 comes after P3 exits.

For P2, the child of the second fork is P4, and it has to wait on it before printing its last two.

New constraint 3 on the output: the last 2 has to happen after P4 prints its 0 and exits.

So, all permutations that obey these new constraints continue to be possible. Some possible outputs:

0 2 0 2 0 1 1 1 1 1
1 0 0 0 2 1 2 1 1 1

Not possible:
0 0 1 1 1 0 2 1 2 1 (breaks constraint 2 since one of the last two 1s did not wait for the top branch outputs, the 0's and 2s all to happen)
0 2 2 0 0 1 1 1 1 1 (breaks constraint 3, since the second 2 must happen after two 0's (one from constraint 1 from the original output without wait, and one from constraint 3)

Problem 4: (22 pts; 10 minutes)
A local laundromat with 5 washing machines is implementing synchronization between customers using semaphores. Each customer first allocates a machine (the machine number is returned by allocate(), and when it is done it returns the machine using release. Available is an array keeping track of whether each of the 5 machines is available (which is represented as 1; 0 is unavailable).

```
1. int allocate()   /* Returns index of available machine. */
2. {
3.  int i;
4.  nfree.wait();  /* wait until a machine is available */
5.  for (i=0; i < 5; i++)
6.   if (available [i] != 0 {
7.     available[i] = 0;
8.     return i;
9.    }
10. }
11. release(int machine)  /* Release machine */
12. {
13.   available[machine] = 1;
14.   nfree.signal();
15. }
```

(a) (4 points) What should the nfree semaphore be intialized to? **The number of washing machines: 5.**

(b) (8 points) Unfortunately, sometimes two customers were getting assigned the same machine, and a fight followed. Explain by tracking two threads representing the customers that fought, exactly how they got assigned the same machine. **This is a race condition on the reading of available. Lets say available[0] is currently set to 1. Two threads can come in and see it equal to 1 as they check the if statement on line 6, before either can reserve it (in line 7), due to the scheduler switching after the first one has checked, or because they are running on two cores concurrently.**

(c) (10 points) Fix the code so that the problem in b does not happen anymore. **Add a lock/mutex before line 5. Here it is important to note that since we have nfree, we are guaranteed that there is one machine available. So, we are guarateneed that eventually every thread will reach line 8. We can release the mutex just before line 8 (after line 8 we would return before releasing). Before line 7 is also wrong since the race condition can occur still. Two threads would be able to find the same washing machine to be available before we set it to not available on line 7. Line 7 must be part of the critical region.**