CS153: Midterm (Fall 16)

Name:

Answer all questions. State any assumptions clearly.

Problem 1: (16 points + 2 bonus; 10 minutes) Mark any 8 of the following statements as True or False. Answer all 10 to get the bonus.

1. —T— / F: Each thread has its own stack
2. —T— / F: Each process has a kernel level and user level stack
3. T / —F—: The OS is a special process
4. T / —F—: Global variables are shared in processes but not threads
5. T / —F—: Kernel level threads are faster than user level threads
6. —T— / F: Its possible for an interrupt to result in a process going from waiting to ready state
7. T / —F—: Its possible for an interrupt to result in a process going from ready to waiting state
8. T / —F—: Its possible for a timer interrupt to cause a process to go from a waiting to ready state
9. T / —F—: An unsafe state (in banker's algorithm for deadlock) is a deadlocked state
10. —T— / F: Locks are used to provide mutual exclusion

Problem 2: (15 points; 10 minutes):
(a) (8 points) We often hear it said in negotiations that two sides are deadlocked. Consider the case of a sports player negotiating a new contract with a team or two political entities negotiating a settlement (e.g., Land for peace). Are such situations deadlock according to our definition? Think about what the resources are. Make a case for either of the above scenarios by showing how the four ingredients for deadlock hold or do not hold for your scenario.

Sports player scenario or country scenario is fine; pick one. Both scenarios can be viewed as deadlock. If you argue they are not and show why you should get credit as well.

Sports player: resources are (1) playing time or player signing contract to play for the team, wanted by the team but held by the player; and (2) money wanted by the player but held by team owner.

Ingredients:

1-Non-preemptive: if you have the money (or the contract) I cannot get it unless you release it.

2-mutual exclusion: the contract (and the money) can either be held by the team or the player

3-Hold and wait: each is holding one and waiting for the other.

4-circular wait: the dependency is in a circle Player to team and team to player.

Countries: the ingredients are land (held by first country and wanted by the other) and agreement to peace (held by second country and wanted by the first). Otherwise, the rationale is the same.

Grading: you should show the four ingredients and how they apply or not. You can argue this is not deadlock, because if/when they agree, an exchange is possible to break the deadlock and you should get full credit if you do this. (b) (7 points) Show a resource allocation graph and a WFG for your scenario in part (a).

```
   +--->Team->contract->Player->money----+
   |                                     |
   |                                     |
   +-------------------------------------+

WFG:

+-->Team-->Player--+
|                  |
+------------------+
```

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 5 jobs that arrive at intervals of 2 msec starting at time 0. The processes have a burst lengths 10, 2, 1, 8, and 4.

(a) (14 points) Show the scheduling timeline for the processes until the end. Track the state of the queues.
P1 (0-1), moves to second queue
P1 (1-3), P2 arrives at time 2 into first queue, P1 demoted to 3rd queue
P2 (3-4), P2 demoted to second queue, P3 arrives at time 4 into first queue
P3 (4-5), P3 done
P2 (5-6), P2 done, P4 arrives
P4 (6-7), P4 demoted to second queue
P4 (7-9), P4 demoted to third queue, P5 arrives at 8
P5 (9-10), P5 demoted to second queue
P5 (10-12), P5 demoted to third queue
P1 (12-16), P1 has 3 left
P4 (16-20), P4 has 1 left
P5 (20-21), P5 done
P1 (21-24), P1 done
P4 (24-25), P4 done
(b) (6 points) What are the advantages of this algorithm compared to round robin scheduling? List at least two. **Compared to round robin: It Gives priority to short burst / I/O bound processes.**
It does not require us to know the burst time ahead of time (but neither does round robin).
It reduces context switches for long process since they eventually percolate to the bottom queue.
It effectively allows adaptive slice size: short size for processes where response time is most important, and long slice for cpu bound processes.

Problem 4: (24 pts; 15 minutes) Consider the following code.

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

pid1 = fork();
pid2 = fork();

if(pid1!=0 && pid2 !=0) {   //both conditions are true
     printf("A \n");
  }

if(pid1!=0 || pid2 !=0) {  //either condition is true
     --------wait(status); //wait added only for part c
     printf("B \n");
  } else {
     printf("C \n");
}
```

(a) (4 points) How many different process will exist when we run the program?

4 processes will execute the statement (first process forks, so we have 2, then each of them forks, making 4 total processes).

(b) (10 points) What are the possible outputs?

```
            --------CC----prints C and exits
           |
           |
     ------ --------CP----prints B and exits
    |
    |       --------PC----prints B and exits
    |      |
    |      |
----------- --------PP----prints A then B and exits
```

The diagram above shows the 4 processes; lets call them PP (for the parent of the first fork, and the parent of the subsequent fork), PC (the parent of the first fork, but the child of the second), CP, the child of the first fork, and the parent of the second, and CC, which is the child of the first and second forks.

pid1 for the PX processes (PP and PC) is nonzero, while pid2 for the XP processes (PP and CP) is nonzero.

1 A (only the original process, PP, will have pid1 and pid2 not equal to 0). 3 Bs (processes PP, PC, and CP will have pid1 —— pid2 equal 1) 1 C (only process CC will have neither pid1 nor pid2 equal 1)

There are no wait statements to synchronize the processes, the only requirement is that the A comes before at least one B (since process PP prints A then B). Any permutation of ABBBC that satisfies this condition (the A happening before at least one of the B's) is legal output. You dont have to enumerate them all.

(c) (10 points) What are the possible outputs if the wait statment is added as shown?

In this case, we introduced a wait statement that adds more constraints. Process CC does not wait, so the C can come anywhere.

Process PC has no children and therefore does not wait (if you assumed it blocks this is fine too, but in unix, wait returns immediately if there are no children). One of the B's can come anywhere.

Process PP waits for EITHER PC or CP but not CC. Thus, AB has to happen after at least one B.

Process CP waits only for CC. One B has to happen after the C.

The following are not possible: Anything that starts with an A since AB has to happen after at least one B.

Also, anything that ends with a C since the B from CP has to happen after the C.

Finally, the strings BBCAB and BBACB are not possible since the last B is the one from PP (because its the only one that follows the A) and therefore the B from CP does not wait for CC to terminate after printing C.

Remaining legal outputs:

CABBB BABCB BACBB BCABB CBABB BCBAB CBBAB

Grading: if you get everything correct +3 bonus points. If you identify the constraints (other than the last one) and describe the pattern, you should get full score.

Problem 5: (25 +5 bonus; 20 minutes)
We would like to implement a barrier synchronization pattern, which is a popular pattern in parallel programming. The barrier is used to make sure n threads have reached that point in the program before any of them is allowed to move forward.

When the first (n-1) threads reach the barrier (i.e., calls `barrier()`) they each get blocked. When the last thread calls barrier, it wakes up all the other threads so that they can all proceed.
(a) (10 points) Show an implementation of `barrier()` using semaphores

Note: the parts did not add up to the total score, so part a is worth 10 points rather than 15.

The main idea here is similar to readers writers lock. We track the number of threads that reach the barrier. If we are the last thread we signal the others to go in.

```
Semaphore mutex(1);// binary semaphore to implement mutual exclusion
Semaphore barrier_wait(0);
int barrier_count=0;  //no one is waiting yet...

thread_code:
...
barrier();
...

barrier implementation:
barrier(){

mutex.wait();
barrier_count++;
if(barrier_count < N) {
  mutex.signal();       //careful not to wait inside mutual exclusion
  barrier_wait.wait();  //<-- Lets talk about this later
}
else {  //we are last thread
  while(--barrier_count >0) {//loop n-1 times to unblock waiters
     barrier_wait.signal(); //signal waiters
     }
  mutex.signal();
}
```

Note that we cannot wait while holding the mutex (inside the if statement wherethe arrow is). You should be concerned about the wait outside the critical region, but it is ok given the semantics of the semaphore; if the barrier is signaled before we wait, the counter will be incremented, allowing us to go in. (b) (10 points) Outline an implementation of barrier using monitors (pseudo code, or a clear description with the relevant technical details is fine)

This implementation is similar, but the condition variable takes care of the wait above.

The state of the barrier is just the counter.

```
Monitor barrier {
int count = 0;
Condition wait_for_last_thread;

barrier() {// arrive barrier
count++;
if(count < N)
   wait_for_last_thread.wait(); //this is ok since condition variables
```

```
                                           //implicitly release mutex
else
   wait_for_last_thread.broadcast(); //or a loop to signal n-1 times
}
```

(c) (5 points) For part (b), does it matter if your monitor uses Hoare or Mesa semantics? Explain.

If we assume that the barrier is used just one time, then it does not matter who runs next since both waiting threads and the last thread just exit the monitor at this point. Either one is fine.

(d) (5 points bonus) We want the barrier to be reusable. That is, after the barrier condition is met and the threads proceed again, we may want to meet use the barrier again at the next synchronization point. Explain clearly why your implementation works or does not for this scenario.

This is a problem for the semaphore implementation: if new threads arrive before the old threads exit the barrier, the whole logic breaks down (the value for the number of threads is not correct, and the threas waiting for the new barrier might be able to pass through while threads from the previous barrier may not.