EXAM ON PROCESSES AND SCHEDULING – SPRING 2003
CSC 262 – OPERATING SYSTEMS
NICHOLAS R. HOWE

1. **Vocabulary** (32 points). Identify the term defined in class that matches the following definitions.

a. An executing program, plus its data and resources. – *Process*

b. An executing program with only the minimum data necessary to sustain its execution (typically status registers and call stack). Code, variables, and resources may be shared with other similar entities. – *Thread or Lightweight Process*

c. This piece of hardware is responsible for signaling the CPU that an interrupt has been requested. – *Interrupt Controller*

d. This data structure, maintained by the kernel, maintains vital information on all the current processes in the system. – *Process Table*

e. This term refers to the a microprocessor operation that cannot be split into two or more simpler operations, and thus cannot be interrupted. – *Atomic*

f. This term refers to a situation where the outcome depends unpredictably upon the order of execution of some set of events. – *Race Condition*

g. This term refers to a set of code where interleavings must be avoided in order to prevent the situation described in item (f) above. – *Critical Section*

h. This term refers to a situation in which some set of processes are blocked while waiting for an event that can only occur through execution of the processes in the set. – *Deadlock*

2. **Scheduling** (16 points). A newly proposed operating system will use multiple queues for process scheduling. It will have three process queues: High, Medium, and Low (maintained via a threaded list). Scheduling between the queues is by strict priority scheduling (with pre-emption). Within queues, round-robin scheduling is used. A hypothetical process table for this system is shown below:

Process ID: 0 Priority: Medium Status: Ready Next: 4	Process ID: 1 Priority: Low Status: Ready Next: 1	Process ID: 2 Priority: Medium Status: Ready Next: 0	Process ID: 3 Priority: High Status: Blocked Next: 3	Process ID: 4 Priority: Medium Status: Ready Next: 2

a. If the dispatcher must choose a process to run from the process table above, which processes could possibly be chosen in accordance with the policies described above? – *0, 2, or 4.*

b. Draw a Gantt chart showing the processes run for the next 500 ms, under the following assumptions: No process terminates, blocks or becomes unblocked during that time; the execution quantum is 100 ms and the switching time is negligible; Process 4 is chosen to run first.

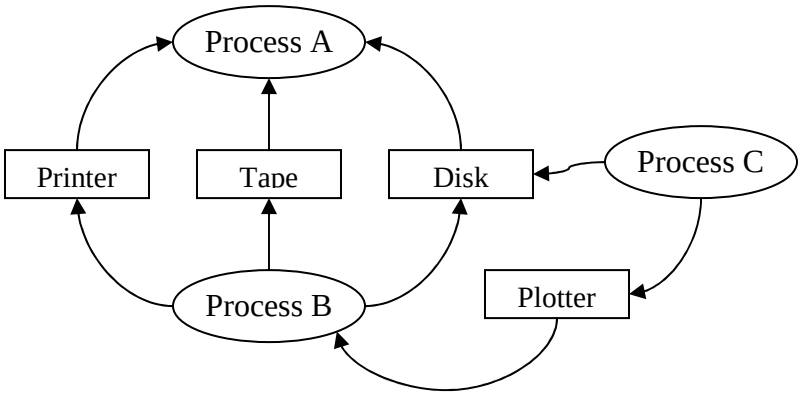
0 ms	*100 ms*	*200ms*	*300 ms*	*400 ms*
Process 4	*Process 2*	*Process 0*	*Process 4*	*Process 2*

c. What change in circumstances would result in Process 1 running? – *All other processes must be blocked.*

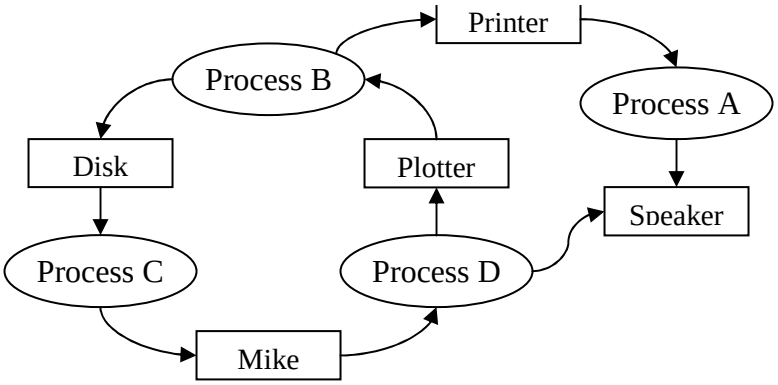
d. What change in circumstances would result in Process 3 running? – *Process 3 must become unblocked.*

3. **Deadlock** (16 points). Draw the resource allocation graph for each of the following situations, and determine which processes (if any) are deadlocked in the following situations. Note that in any given scenario, some, none, or all of the processes may be deadlocked.

a.	Printer	Plotter	Tape Drive	Disk Buffer
Process A	Has		Has	Has
Process B	Wants	Has	Wants	Wants
Process C		Wants		Wants



b.	Printer	Plotter	Speaker	Microphone	Disk Buffer
Process A	Has		Wants		
Process B	Wants	Has			Wants
Process C				Wants	Has
Process D		Wants	Wants	Has	



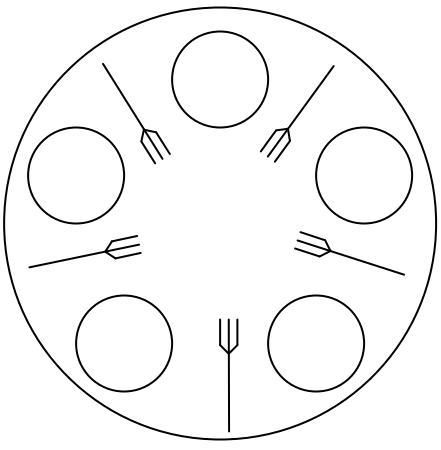
4. **Linux Processes** (12 points). Explain how to set and/or modify the priority of a process in Linux, assuming the kernel is using the `SCHED_OTHER` protocol. What limitations are placed on ordinary users (as opposed to root) as far as the ability to set process priorities?

Priorities are represented by a number from -19 to +20, and default to 0. The **`nice`** *command may be used to run a process at a different priority level. Only root may run a process at priority less than 0. The* **`renice`** *command adjusts the priority of a running process. Only root may upgrade the priority (i.e., lower the number).*

5. **Message Passing** (8 points). A friend is working on a protocol for a branch library's computer system using message passing. The checkout computer in the branch library cannot check out the book without making sure the central library records are updated. The central library computer won't update the records until it is certain the item is being checked out. The cable connecting the branch to the main library has been shorting out lately, and so your friend has been hired to come up with a foolproof mechanism that will make sure both computers act at the same time. She is having some trouble getting a working protocol, and has come to you for advice. What do you suggest to her? Can you help her to write such a protocol?

There is no protocol that can achieve the desired result. This is an application of the Two Generals problem. Since the communications channel is unreliable, whoever sends the last message cannot be sure that it has gotten through, and therefore cannot be certain that the other computer will act. No protocol or amount of communication will ensure that.

6. **Semaphores** (16 points). The dining philosophers problem is a classic example in concurrent programming. Five philosophers (represented by processes) sit at a table alternately thinking and eating. To eat, they must pick up the fork to both their left and right, which are shared with the neighboring philosophers on either side. A philosopher attempting to eat will wait (potentially forever) until she has both forks. A philosopher who is done eating will put down



both forks and begin thinking. Consider the following solution to the dining philosophers problem, using semaphores:

```
1:    var semaphore fork[5] init 1;
2:    philosopher[i]: process
3:       {think}
4:       if even(i) then  // even(i) tests whether i is not odd
5:          DOWN(fork[i]);
6:          DOWN(fork[(i+1) mod 5]);
7:       else
8:          DOWN(fork[(i+1) mod 5]);
9:          DOWN(fork[i]);
10:      endif;
11:      {eat}
12:      if even(i) then
13:         UP(fork[(i+1) mod 5]);
14:         UP(fork[i]);
15:      else
16:         UP(fork[i]);
17:         UP(fork[(i+1) mod 5]);
18:      endif;
19:   endprocess;
```

a. How would the protocol's behavior change if lines 5 and 6 were exchanged? If there could be a change in behavior, describe a specific scenario where it would be evident.

If lines 5 and 6 are exchanged, then we have the classic dining philosophers problem except that every philosopher picks up the left fork first. If all philosophers try to eat at the same time, and each picks up the her left fork, then deadlock will occur.

b. How would the protocol's behavior change if lines 13 and 14 were exchanged? If there could be a change in behavior, describe a specific scenario where it would be evident.

The behavior will not change.

c. Is this protocol deadlock-free? If so, which of the four necessary conditions for deadlock is negated by this algorithm?

Circular waiting is impossible. The protocol above effectively establishes ordered classes of resources, where even-numbered forks must be picked up before odd-numbered forks. Because of this restriction, circular waiting cannot occur.

d. What are the possible values that fork[i] might take on during the execution of this protocol?

The only possible values are 0 and 1. (Recall that a semaphore's value cannot be negative; a process attempting to execute a DOWN on a semaphore with value equal to zero will block until the semaphore's value increases.)