COS 318: Midterm Exam (October 24, 2013)

(80 Minutes)

Name:

This exam is closed-book, closed-notes.

No laptops, no mobile devices of any kind, and no Internet accesses are allowed during the exam.

The points in the parenthesis at the beginning of each question indicate the points given to that question.

Write all your answers directly on this paper. Use the blue book only if you run out of space. Make your answers as concise as possible.

Partial credit will be given if you can show your work in arriving at solutions.

1	
2	
3	
4	
5	
Total	

Pledge (please write out "I pledge my honor that I have not violated the Honor Code during this examination" and then sign):

1 Short Questions (10 points)

Answer the following questions. Use exactly **one or two** sentences to describe why you choose your answer. Without the reasoning, you will not receive any points.

a. (2 points) You write a UNIX shell, and in the main loop your code first calls exec() and then fork() like the following. Does this work?

```
shell (...) {
 ….
    exec (cmd, args);
    fork();
 ….
}
```

Answer: It doesn't work. The shell's address space is entirely replaced with the new command (cmd), and so the shell will terminate once cmd is terminated.

b. (2 points) Suppose you want to implement a multithreaded web server where each thread serves one incoming request by loading a file from the disk. Assume the OS only provides the normal blocking read system call. Should you use user-level or kernel-level threads?

Answer: Kernel-level threads. Each thread will make blocking I/O calls; with user-level threads, one thread will block all the other threads.

c. (2 points) Can an interrupt occur while a kernel thread is running?

Answer: Yes. Interrupts can occur at any time unless they are explicitly disabled, e.g., for a critical section.

d. (2 points) Can a microkernel implement its virtual memory subsystem inside a process?

Answer: No. Virtual memory is used to provide a process's address space abstraction, so it must run inside the (micro)kernel.

e. (2 points) In UNIX, what is the responsibility of the *linker*?

Answer: It links together multiple object files and static libraries into an executable. It resolves symbols between object files and determines how arrange the process's address space.

2 CPU Scheduling (10 points)

Consider a preemptive priority scheduler with N priority queues. If the highest non-empty priority queue contains multiple jobs, the scheduler runs them round robin with a time-slice of 200ms. The priorities of jobs are dynamically adjusted as follows:

Rule 1: When a job enters the system, it is placed on the highest priority queue.

Rule 2: Once a job accumulates 100ms of CPU time at a given level, its priority is reduced by 1.

Rule 3: After 1 second, all the jobs in the system are moved to the highest priority queue.

a. (3 points) What kinds of jobs are favored by rules #1 and #2? What are the benefits of this approach?

   Answer: A long-running CPU-bound job will have its priority frequently reduced by rule #2. Therefore jobs that don't use much CPU, like short jobs and I/O-bound jobs, are favored because they will maintain high priority levels. The benefits are better interactive response time for short jobs, and more opportunities to achieve I/O parallelism by running I/O-bound jobs.

b. (3 points) What condition is avoided by rule #3? How?

   Answer: Starvation of CPU-bound jobs is avoided. Without this rule, long-running CPU-bound jobs would have their priorities reduced to the lowest level; these jobs could then be starved by a stream of new jobs entering the system. Resetting the priorities every second gives CPU-bound jobs an opportunity to run.

c. (4 points) Suppose the scheduler has three priority queues. The system is running three jobs: Job A uses 50ms of CPU and then does 200ms I/O; jobs B and C constantly use the CPU. At time *t*, the priorities have all been reset according to rule #3, and the highest priority queue is ordered (front to back): A, B, C. Assume that a timer interrupt fires every millisecond and accurately accounts the CPU time to the currently running process. Trace the execution of jobs until time *t + 1 second* by filling out the following table; add more rows if required.

Start time (ms)	Current Job	Runs for (ms)	At priority
0	A	50	3
50			

Answer: The key insight is that a job's priority can be reduced in the middle of a timeslice so that it is no longer be the highest priority runnable job. A preemptive priority scheduler always runs a job with the highest priority. Here's a trace of the execution:

Start time (ms)	Current Job	Runs for (ms)	At priority	Reduced?
0	A	50	3	
50	B	100	3	Y
150	C	100	3	Y
250	A	50	3	Y
300	B	100	2	Y
400	C	100	2	Y
500	A	50	2	
550	B	200	1	
750	A	50	2	Y
800	C	200	1	

3 Deadlocks (6 points)

a. (4 points) What are the necessary conditions for deadlock? Write one sentence per condition.

Answer: (1) mutual exclusion - at least one resource is non-sharable; (2) hold and wait - a process is holding some resource while waiting for another; (3) no preemption - cannot take a held resource away; (4) circular wait - P0 waits for P1 waits for .... Pn waits for P0

b. (2 points) Fix the following code to avoid the possible deadlock:

Process 1:
```
acquire(L1)
acquire(L2)
release(L2)
release(L1)
```

Process 2:
```
acquire(L2)
acquire(L1)
release(L1)
release(L2)
```

Answer: The best fix is to remove circular wait and make both processes acquire the resources in the same order.

4 Input and Output (5 points)

Briefly list the steps of a blocked read system call in a typical monolithic operating system (such as Unix) by a process. This system call uses DMA hardware to read from a disk device.

Answer:

A process issues a read call, which executes a system call
System call: checks for correctness and buffer cache
If it needs to perform I/O, it will issue a device driver call
Device driver: allocates a buffer for read and schedules I/O
Initiate DMA read transfer
Block the current process and schedule a ready process
Device controller: performs a DMA read transfer
Device controller: sends an interrupt upon completion
Interrupt handler: wakes up blocked process (make it ready)
System call: Move data from kernel buffer to user buffer
returns to the caller
User process continues

5 Monitors (10 points)

You just joined the project to design a simulation system to simulate traffic. Your job is to design and implement the code to synchronize traffic over a narrow bridge. Vehicle traffic may only cross the bridge in one direction at a time. In the system, each vehicle is represented by a thread, which executes the procedure **`CrossBridge()`** when it arrives at the bridge:

```
enum Direction (East, West, NULL);
CrossBridge(Direction direc){
  Arrive(direc);
  Cross(direc);
  Leave(direc);
}
```

The variable *direc* gives the direction in which the vehicle will cross the bridge.

Your job is to design necessary data structures and implement **`Arrive(), Cross(),`** and **`Leave()`**. **`Arrive()`** must not return until it safe for the vehicle to

cross the bridge in the given direction. **`Cross()`** will take 2 seconds to cross the bridge and will log (print) the time the vehicle leaves the bridge. **`Leave()`** is called after the caller has finished crossing the bridge and to let additional cars cross the bridge.

Your implementation should obey the following requirements:

1. No head-on collisions.
2. No more than 4 vehicles can be on the bridge at the same time (otherwise, the bridge may collapse).

Your implementation should allow up to 4 vehicles on the bridge when they are heading in the same direction. Your implementation does not need to worry about fairness and starvation. You can make your own assumptions about sleeping for certain amount of time and printing the current timestamp.

You are required to use Mesa-style monitor to implement the synchronization among multiple threads. If you use other synchronization mechanisms, no points will be given.

Answer:

```
Mutex lock;
Condition safe;
int currentCars;
int currentDirec;

Init() {
  MutexInit(lock);
  CondInit(safe);
  currentCars = 0;
  currentDirec = NULL;
}

Arrive(Direction: direc) {
  Acquire(lock);
  while (!SafeToCross(direc))
    Wait(lock, safe);
  currentCars++;
  currentDirec = direc;
  Release(lock);
}

Cross(Direction: direc) {
  sleep(2);
  printf("direction: %d, time: %d\n", direc, gettime());
}
```

```
Leave(Direction: direc) {
  Acquire(lock);
  currentCars--;
  Broadcast(safe);
  Release(lock);
}

SafeToCross(Direction: direc) {
  if (currentCars == 0)
    return TRUE;
  if (currentCars < 4 && curretDirec == direc)
    return TRUE;
  return FALSE;
}
```