CS-537: Midterm Exam (Fall 2005)

CPU Management

Please Read All Questions Carefully!

There are eight (8) total numbered pages.

Name and Student ID: **Solution Sketch** ____________________

Grading Page

	Points	Total Possible
Part I: True or False		$(15 \times 3) \rightarrow 45$
Part II: Short Answers		$(10 \times 10) \rightarrow 100$
Total		145

Part I: True or False

Circle **True** or **False** for each of the following statements. **Each of the 15 questions is worth 3 points (45 total)**.

1. **True** False One of the roles of the OS is to make hardware easier for applications to use
2. True **False** The primary performance goal of a modern OS to maximize the utilization of expensive resources
3. True **False** A process is equivalent to a program
4. True **False** A process is equivalent to a thread
5. **True** False Cooperative multi-tasking assumes that processes do not misbehave
6. **True** False Hardware support is required to implement true multi-tasking
7. True **False** A process that is waiting to use the CPU is in the blocked state
8. **True** False In Unix, the fork() system call creates a new process
9. True **False** Lamport's Bakery Algorithm can be used to protect critical sections between only two threads
10. True **False** A context switch may occur in the middle of performing the hardware instruction TestAndSet.
11. **True** False Assume a thread is running on a multiprocessor system and is waiting to acquire a lock. If one knows that the lock will be released in time t, where t is less than the cost of a context-switch, then the thread should spin-wait while waiting for the lock.
12. **True** False One can prevent deadlock by having threads acquire locks in a predefined order.
13. True **False** A non-preemptive scheduler may choose to switch between processes when an I/O completes.
14. **True** False Using average waiting time as the performance metric, a STCF scheduler always performs as well as or better than a FCFS scheduler.
15. True **False** Using average waiting time as the performance metric, a Round Robin scheduler always performs as well as or better than a FCFS scheduler.

Part II: Short Answer

Each of the 10 questions is worth 10 points (100 points total).

1. Multiple threads within a single process share which of the following? **(Circle all that apply)**
    - **Process ID (PID)**
    - Thread ID (TID)
    - **User ID (UID)**
    - **Address space**
    - **Code segment**
    - Register contents
    - Program counter value
    - Stack pointer value

2. Assume the following C code is compiled to create the executable program, `pointless`, which is then run on a UNIX system.

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  int a = 0;
  int b = 0;
  a++;
  if (fork() == 0) {
    b +=2;
    printf("If!\n");
    printf("a: %d\n", a);
    printf("b: %d\n", b);
  } else {
    b++;
    printf("Else!\n");
    printf("a: %d\n", a);
    printf("b: %d\n", b);
  }
  a++;
  if (fork() == 0) {
    b++;
    printf("If #2!\n");
    printf("a: %d\n", a);
    printf("b: %d\n", b);
  } else {
    b++;
    printf("Else #2!\n");
    printf("a: %d\n", a);
    printf("b: %d\n", b);
  }
}
```

A) How many processes will be part of the `pointless` program?

4

B)Assume a preemptive scheduling policy in which a newly created process is always given a higher, static priority than any of the existing processes; also assume that processes never voluntarily relinquish the CPU until they exit (even if the call printf). What output will the user see on the terminal when they run `pointless`? You may show this output above, next to the code listing.

```
If!
a: 1
b: 2
If #2!
a: 2
b: 3
Else #2!
a: 2
b: 3
Else!
a: 1
b: 1
If #2!
a: 2
b: 2
Else #2
a: 2
b: 2
```

3. Consider the following attempt to protect a critical section across two threads, $t0$ and $t1$ (you may assume that `tid` is set to 0 or 1, appropriately).

```
boolean lock[2] = {False, False}; // Shared

void deposit(int amount) {
 lock[tid] = True;
 while (lock[1-tid]) /* wait */ ;

 balance += amount;

 lock[tid] = False;
}
```

A) Does this solution provide *mutual exclusion*? If not, why not?

Yes, it provides mutual exclusion. It is not possible for more than one thread to be in the critical section since each thread checks whether the other thread has set its lock flag.

B) Does this solution guarantee *progress*? If not, why not?

No, it does not guarantee progress. It is possible that neither thread enters the critical section. If a context-switch occurs after each thread sets its lock to True, then each thread will see that the other's lock is set, and each will busy-wait forever.

C) Does this solution guarantee *bounded waiting*? If not, why not?

Yes, this solution will guarantee bounded waiting *as long as there is progress*. **After one thread exits the critical section, the other thread will be allowed to enter if it wanted to (i.e., if it had already set its lock to True, or if it does so before the first thread tries again).**

4. Interrupts are sometimes *disabled* as a way of protecting a critical section.

A) Why isn't disabling interrupts necessarily a good way to protect a critical section? List the disadvantages of this approach.

- **Disabling interrupts doesn't prevent another process from being scheduled on a multiprocessor.**
- **User processes should not be able to control when interrupts are disabled. The user code could either accidentally (e.g., by enterring an infinite loop) or maliciously not re-enable interrupts, thereby preventing the OS or any other process from ever running.**
- **Disabling interrupts means that absolutely no interrupts can be handled in that interval (e.g., handling page faults, I/O, or updating the clock).**
- **Disabling interrupts keeps any other process from being scheduled, which is more restrictive than what is needed to protect the critical section; we really just need to ensure that no other processes that want to enter the critical section are scheduled.**

B) For what type of critical sections is it most appropriate to disable interrupts? Give a concrete example.

Disabling interrupts is appropriate within the OS for short periods of time. It should be used only to optimize performance, and not when it is needed for correctness (since in a multiprocessor environment, another process may be scheduled anyway). A good example is within the code that implements lock_acquire: by turning off interrupts before the busy-wait code protecting the lock queue, the likihood of having to busy-wait decreases.

5. Semaphores and condition variables can both be used to specify scheduling constraints within a multi-threaded application. However, there are obvious differences between the two synchronization primitives.

   A) What **two** pieces of state are associated with a **semaphore**?

   - **The integer value of the semaphore (when this value is greater than 0, a process calling wait on the semaphore does not block).**
   - **A queue of (blocked) processes or threads waiting on this semaphore.**

   B) What **one** piece of state is associated with a **condition variable**?

   - **A queue of (blocked) processes or threads waiting on this condition variable.**

6. Imagine that you have a **single** thread, $t1$, waiting on a condition variable `x` and then another thread, $t2$, signals the condition variable `x`. What steps must happen before $t1$ can return from its `cond_wait(&x)` call? Assume Mesa semantics.

   When $t2$ signals $t1$, $t1$ is removed from the queue of threads waiting on this condition variable and is added to the queue of threads waiting on the associated monitor lock. Next, $t2$ must release the monitor lock. At some point, $t1$ will be given the monitor lock (but, other threads may acquire it first) and move to the RUNNABLE state. $t2$ will then return from the `cond_wait(&x)` when it is scheduled.

7. Assume that we have five dining philosophers and five chopsticks, each numbered from 0 to 4. Consider the following "solution" to the dining philosopher's problem, where we use a semaphore, `chop[i]` for each chopstick i.

```
take_chopsticks(int i) {
    if (i) {
        sem_wait(&chop[i]);
        sem_wait(&chop[(i+1) % 5]);
    } else {
        sem_wait(&chop[1]);
        sem_wait(&chop[0]);
    }
}
```

   Does this solution lead to deadlock? Why or why not?

   No, this solution cannot cause deadlock. Philosopher 0 acquires its chopsticks (i.e., semaphores) in a different order from the others. Therefore, there is no circular dependency in the allocation requests.

8. Assume you have the following jobs to execute on one processor, with the jobs arriving at the following times and requiring the following amount of CPU.

Job	Arrival Time	CPU Time
A	0	5
B	2	4
C	4	5
D	6	2

For each of the following scheduling algorithms, compute the **waiting time of each job**. Note that drawing the Gantt charts may be helpful for you and may allow us to give you partial credit if you make a mistake, but is not required for full credit.

FCFS:

Time 0-5: A

Time 5-9: B

Time 9-14: C

Time 14-16: D

Waiting time: A:0, B: 5-2=3, C: 9-4=5, D: 14-6=8

SJF:

Time 0-5: A

Time 5-9: B

Time 9-11: D

Time 11-16: C

Waiting time: A:0, B: 3, C: 7, D: 3

STCF:

Time 0-5: A

Time 5-6: B

Time 6-8: D

Time 8-11: B

Time 11-16: C

Waiting time: A:0, B: 3+2=5, C: 7, D: 0

9. Describe two practical problems that exist with a STCF scheduler.

- **Scheduler cannot know for certain the future CPU burst length of a job**
- **Jobs with long burst lengths may be starved (by continuously arriving shorter jobs)**

10. Consider a **multilevel feedback queue scheduler** that is notified every 10 ms that a timer has expired. What operations should the scheduler perform at this point? Please describe **briefly** the high-level functionality that is needed (and do not write the code).

The scheduler should track how many timer ticks have been given to this job so far. If the timeslice of the running job has expired (i.e., some number of timer ticks have gone by), then the priority of this job should be reduced (i.e., it has longer CPU bursts than the other jobs at this priority). This job should be given a longer timeslice at its lower priority and the job should be added to the new priority queue (probably at the back).

The scheduler should also periodically check whether there are any starving jobs (i.e., it should watch to see if a job has not been scheduled over some interval). If a job is starving, then its priority should be increased, thus improving its chances of being scheduled in the future.

At the end of the routine, the scheduler should tell the dispatcher to run the highest priority job in its run queues.