CS-537: Midterm Exam (Fall 2004)

The Legend of the Fall

Please Read All Questions Carefully!

There are eight (8) total numbered pages.

Please put your student ID (but NOT YOUR NAME) on every page.

Name and Student ID: ____________________

Grading Page

	Points	Total Possible
Part I: Short Answers		$(12 \times 5) \to 60$
Part II: Long Answers		$(2 \times 20) \to 40$
Total		100

Part I: Short Questions

The following questions require short answers. **Each of 12 is worth 5 points (60 total)**.

1. What is a *context switch*? Describe what the OS must do to implement a context switch correctly.

2. When a Unix process calls fork(), the system creates (in the child) a nearly exact copy of the calling (parent process). Why is the copy "nearly exact" and not an "exact" copy of the parent process?

3. Consider the properly synchronized routine:

   ```
   // assume ''balance'' is a global variable
   void update(int amount) {
       mutex_lock(lock);
       balance = balance + amount;
       mutex_unlock(lock);
   }
   ```

   However, there is one "rogue" thread that updates balance in a different way, by calling its own special routine:

   ```
   // assume ''balance'' is a global variable
   void rogue_update(int amount) {
       balance = balance + amount;
   }
   ```

   When we are writing multi-threaded code, is there some way to prevent threads from accessing and updating shared data structures in this "rogue" fashion?

4. In class, we talked about the following possible solution to the two-thread synchronization problem:

```
while (turn == (1 - threadID))
    ; // spin
balance = balance + amount; // critical section
turn = 1 - threadID;
```

In this solution, assume that “turn” is initialized to “0”, and that there are only two threads that run. Does this solution provide **mutual exclusion?** (Explain)

5. Some architectures provide instructions to aid with writing synchronization routines. One such instruction was “test-and-set”, as described in class. **Describe how test-and-set works**; in doing so, **show how one can use it to provide mutual exclusion**.

6. Imagine a new synchronization primitive (a close cousin of our favorite Semaphore) called the `WhatsItFor`. A `WhatsItFor` has an initial value (which is initialized by the user), and two related routines, `One()` and `Done()`, that work as follows. `One()` waits for the value of the `WhatsItFor` to be less than zero, and then increments the value by one. `Done()` decrements the `WhatsItFor` by one, and then wakes one waiting thread (if there is one). Both `One()` and `Done()` execute atomically.

   Show how to use a `WhatsItFor` (specifically, `One()` and `Done()`) to build a simple lock around a critical section. Make sure to **specify the initial value of the `WhatsItFor`.**

7. In class we discussed the following "solution" to the dining philosopher's problem:

```
acquire(int i) {
    if (if < 4) {
        sem_wait(chop[i]);
        sem_wait(chop[i+1]);
    } else {
        sem_wait(chop[0]);
        sem_wait(chop[4]);
    }
}
```

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

8. Reader/writer locks were implemented in class as follows, with the semaphores `writeLock` and `mutex` initialized to 1, and the counter `readers` initialized to 0.

```
void getWriteLock() {
    sem_wait(writeLock);
}

void getReadLock() {
    sem_wait(mutex);
    readers++;
    if (readers == 1)
        sem_wait(writeLock);
    sem_post(mutex);
```

```
void releaseWriteLock() {
    sem_post(writeLock);
}

void releaseReadLock() {
    sem_wait(mutex);
    readers--;
    if (readers == 0)
        sem_post(writeLock);
    sem_post(mutex);
```

What is the basic problem that can occur if there is a continuous stream of readers that grab the read lock?

9. When discussing the difference between Hoare and Mesa semantics for condition variables inside of monitor routines, we focused on the following producer/consumer code segment:

```
produce () {
    if (fullEntries == MAX)
        cond_wait(empty);
    // fill buffer
    fullEntries++;
    cond_signal(full);
}
```

```
consume () {
    if (fullEntries == 0)
        cond_wait(full);
    // empty the buffer
    fullEntries--;
    cond_signal(empty);
}
```

When using **Mesa** semantics, the semantics of condition variables demanded that the `if` statements be changed to `while` statements. **Describe why Mesa semantics demand the change from "if" to "while" in the producer/consumer code above**.

10. Assume a set of five jobs arrives in a system to be scheduled at roughly the same time; each job runs for 10 seconds if running by itself on the CPU. Compute the **average response time** and the **average turnaround time** for a **round-robin** policy with a **500 millisecond** time slice. Show your work (as much as possible)

11. The shortest-job-first (SJF) and shortest-time-to-completion-first (STCF) policies are both unrealistic policies to implement in a general purpose operating system. Why?

12. Assume a multi-level feedback queue scheduling policy. In class, we discussed a provision that periodically moved all jobs back to the topmost priority queue. **What problem does this rule solve?**

Part II: Longer Questions

The second half of the exam consists of two longer questions, **each worth 20 points (total 40)**.

1. **The Battle for Control.** In this question, we discuss different methods that the operating system can use to "gain control" of the CPU.

   a): The classic mechanism the OS uses is a *timer interrupt*. Please describe how the OS uses a timer interrupt to gain control of the CPU, and why this is important.

   b): Imagine a new hardware mechanism that counts the the total number of instructions that have been executed and then raises an interrupt after some fixed number of instructions have passed. The OS uses this mechanism by configuring an *instruction-counter interrupt register* (ICIR); by setting the value of the ICIR to x, the OS makes sure that after x instructions are executed, an interrupt is raised. Is the ICIR a good mechanism for the OS to use to gain control of the CPU? Why or why not?

   c): Assume we are using the instruction-counter interrupt register, but we wish to mimic the behavior of the good old timer interrupt; that is, we want a clock tick to go off every 10 milliseconds or so. How would you use the instruction-counter interrupt mechanism to achieve this effect?

   d): Now imagine a new hardware mechanism that counts the number of load and store instructions that have been executed, and raises an interrupt after some (configurable) number of loads/stores have been executed. Is this a good mechanism for the OS to use to gain control of the CPU? Why or why not?

2. **The Best Hash I've Ever Had.**

Assume you have the following code for a multi-threaded hash table, which inserts an object `obj` into the hash table, to be associated with the key `key`. Assume that the hash table is implemented as an array of linked lists; first, we hash (using modulo) to find which list to put an item on, and then we insert the item into that list.

```
void hash_insert(int key, object_t *obj) {
    int whichList = key % HASH_SIZE;
    list_insert(hashLists[whichList], key, obj);
}
```

Assume the code to insert an item onto a list looks something like this (some details omitted for clarity):

```
    // create ``tmp'', which contains the key ``key'' and object ``obj''
    tmp->next = head;
    head = tmp;
```

a): Why does the code inside of `list_insert` need to be synchronized? (i.e., what problems can occur if two threads both try to insert items into the same list at nearly the same time?)

b): Add a single lock into `hash_insert()` to fix the problem.

c): What is the biggest performance problem with this single-lock solution?

d): Now add more locks to `hash_insert` (as many as you want)to improve the performance (or rather, the potential concurrency) of accessing the hash-table data structure.