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		$(12 \times 5) \rightarrow 60$
Part II: Long Answers		$(2 \times 20) \rightarrow 40$
Total		100

Name: _____

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.

   A context switch is loosely defined as the stopping one process from running, saving its state, restoring the state of another job, and starting it running. One could give varying levels of details on what the OS must do to enact the context switch, but all we were really looking for was the saving and restoring of state (e.g., registers, PC, stack pointer, etc.)

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?

   One major difference is the PID, which is reflected in the return code of the fork(): the parent process's return code is the PID of the child, and the child's return code is 0.

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?

   We gave full credit to all on this one, because it was a bit unclear. What we meant to ask was: can we change the `update()` *routine in any way to avoid the problems caused by* `rogue_update()`*? (assuming you still have to use the 'balance' variable). The basic answer was no: threaded code assumes trust among threads, and hence you have to basically "get it right". Some people cleverly suggested that you could make all routines that touch balance "monitor" routines, which would force a lock and solve the problem – nice!*

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)

The turn variable does indeed provide mutual exclusion: the turn can only be one value at a time and is set atomically. The problem with this code is of course that is leads to unbounded wait – it only works properly if both threads are periodically entering the critical section.

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**.

Test and set works as follows. Assume you have: `test-and-set <address>, value`*. This returns the old value at* `address` *and sets the new value to be* `value`*. This is all done atomically.*

Test and set can be used as follows:

```
while (test-and-set(lockaddr, true) == true)
    ; // spin wait
<critical section>
lock = false;  // note, no need to test-and-set here
```

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`**.**

A WhatsItFor is kind of like an inverted semaphore; it just counts the other direction. Hence, to provide mutex, just init to -1 instead of 1 (as we would with a binary semaphore.)

```
WhatsItFor w;
WhatsItFor_Init(&w, -1); // init to -1

WhatsItFor_One(&w);
<critical section>
WhatsItFor_Done(&w);
```

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?)

No, it does not, because of the `else` *clause in the code. Specifically, the 4th philosopher grabs the locks in a different order (0 then 4, instead of the expected 4 then 0). Hence, the cycle is broken and deadlock is avoided. However, it is not an optimal solution from a concurrency standpoint.*

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?

Once a single reader grabs the lock, no writer can proceed until all readers have released the lock. Hence, writers can starve with this solution.

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**.

Mesa semantics say that the signal, while waking a single waiting thread, does not immediately transfer control to that thread. Hence, when the waking thread finally runs, it must recheck the condition, simply because the condition it relies upon being true may no longer be true. Why? Because another thread could have slipped in after the first thread changed the condition and changed it back.

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)

    Response time is the time from submit to first run. Hence, for five jobs that are round-robin scheduled with a 500 millisecond (ms) time slice, we have: 0 ms (first job) + 500 ms + 1000 ms + 1500 ms + 2000 ms. Average of these values is the sum divided by 5, or 5000 ms divided by 5 which is 1000 ms.

    Turnaround time is the time from submit to completion. With five jobs that each run for 10 seconds, we know that the total time for completion for all jobs is 50 seconds. Hence, the last job completed at 50 seconds, the job before that at 49.5, the job before that at 49.0, and 48.5, and 48.0. The average turnaround time is thus the average of 48, 48.5, 49, 49.5, and 50, which is 49 seconds.

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?

    These are unrealistic primarily because one cannot in general know what the run-time of a job will be before it is run. Starvation is also an issue but not the primary problem here.

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?**

    This rule solves two problems. The primary problem it solves is one of starvation – long-running jobs are now guaranteed to get some share of the processor. A secondary problem it solves is that jobs that switch their behavior (e.g., from CPU-bound to interactive) now will get better served by a scheduler that periodically re-learns about their behavior.

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.

   The OS sets the timer interrupt to go off every so often (say every 10 ms) in order to gain back control of the processor. With this mechanism, the OS knows it will get to run again and make scheduling decisions. It is important because without such a mechanism, the OS is relying upon user processes to relinquish control of the CPU – something that buggy or malicious programs might not do.

   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?

   Assume that this mechanism, just like the timer interrupt, is priviledged (i.e., a user process cannot set the ICIR). Then we have a basic mechanism to regain control of the CPU. Instead of setting the timer interrupt to go off every 10 ms, with the ICIR, we set an interrupt to go off every so many instructions.

   Many of you pointed out that instruction times may vary and hence this is a little harder to use – a point with which we agree. However, that in and of itself does not make it impossible.

   Some of you pointed out that if an instruction could get into an infinite loop, then the mechanism will not work – a point with which we also agree.

   Finally, please do not confuse mechanism with policy! Specifically, even if ICIR interrupts go off at irregular intervals, the scheduling policy still could track how long each process has run for and achieve a particular scheduling goal. The period of interrupts does not equal the time given to each process!

   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?

   Set the ICIR to an initial value (say x instructions). Record the current time (t_1). Then run a process. When the ICIR interrupt goes off, record the time again (t_2). By calculating the average time per instruction, $\frac{t_2-t_1}{x}$, we can make a guess at what value to set x to in order to have the ICIR interrupt us every 10 ms. Of course, the more such measurements you take, the better your guess will be. You could do the measurement on a per-process basis, assuming that each process will achieve a different average time per instruction.

   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?

   This is not a good mechanism – if a process enters an infinite loop that simply branches back to the same point in the code (and more importantly, does no loads or stores), then the OS will not be able to regain control.

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?)

The update of head presents us with a race condition. Imagine this interleaving:

```
      thread 1                          thread 2
tmp->next = head;
                                  tmp->next = head;
head = tmp;
                                  head = tmp;
```

In this case, the two (separate) tmp variables next fields will get set to the old head, and then the new head will only end up pointing to the tmp in thread 2 – thread 1's insertion will get lost.

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

You could do this in any way you wanted. Simplest approach:

```
int whichList = key % HASH_SIZE;
mutex_lock(&lock);
list_insert(hashLists[whichList], key, obj);
mutex_unlock(&lock);
```

Note you don't have to lock the setting of the thread-private stack variable `whichList`.

What pained us is when people tried to write their own locks (which would be OK), but did so incorrectly:

```
while (lock == true)
    ; // spin
lock = true;
<critical section>
lock = false;
```

This is **NOT** *a working lock!*

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

The biggest problem with the single-lock solution is that it prevents concurrent updates to separate lists in the hash table.

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.

Simple – let's add a lock per list.

```
int whichList = key % HASH_SIZE;
mutex_lock(&lock[whichList]);
list_insert(hashLists[whichList], key, obj);
mutex_unlock(&lock[whichList]);
```

This enables concurrent updates to separate lists.