CS-537: Midterm Exam (Spring 2002)

The Mid-Semester Blues

Please Read All Questions Carefully!

There are nine (9) total numbered pages

Please put your name on every page.

Name: ____

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

Name: ______________________________

Part I: Short Questions

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

1. Processes (or threads) can be in one of three states: **Running**, **Ready**, or **Blocked**. For each of the following four examples, write down which state the process (or thread) is in:

   (a) Waiting in `Domain_Read()` for a message from some other process to arrive.

   (b) Spin-waiting for a variable `x` to become non-zero.

   (c) Having just completed an I/O, waiting to get scheduled again on the CPU.

   (d) Waiting inside of `pthread_cond_wait()` for some other thread to signal it.

2. An operating system runs in *privileged* mode, a hardware state where it has full access to machine resources. Why is such a mode needed, and why can't normal user processes and threads enter privileged mode?

3. In a system with pure paging, assume we have a 32-bit address space, and a 4 KB page size.

   a) How many bits of an address specify the *logical page number* (a.k.a. *the virtual page number*), and how many bits specify the *offset*?

   b) Let's say we are translating the logical address 0x00010033; if each logical page is mapped to a physical page that is a single page number higher (i.e., logical page 10 is mapped to physical page 11, logical page 11 is mapped to physical page 12), what is the final translated physical address?

4. Three jobs (A, B, and C) arrive to the job scheduler at time 0. Job A needs 10 seconds of CPU time, Job B needs 20 seconds, and Job C needs 30 seconds.

   a) What is the *average turnaround time* for the jobs, assuming a *shortest-job-first* (SJF) scheduling policy?

   b) What is the *average turnaround time* assuming a *longest-job-first* (LJF) policy?

   c) Which finishes first, Job C in SJF or Job A in LJF?

5. In class, we gave the following code as an implementation of mutual exclusion:

```
boolean lock[0] = lock[1] = false;
int turn = 0;
void deposit (int amount) {
    lock[pid] = true;
    turn = 1 - pid;
    while (lock[1-pid] && (turn == (1 - pid)))
        ; // spin
    balance = balance + amount;
    lock[pid] = false;
}
```

Let's say we replace the statement `turn = 1 - pid` with the statement `turn = BinaryRandom()`, where the function `BinaryRandom()` returns a 1 or 0 at random to whomever calls it. **Will the code still function properly? If so, why, and if not, what problem could occur?**

6. A number of threads periodically call into the following routine, to make sure that a pipe that is shared between them has already been opened (after calling this routine, a thread might go ahead and call write() on that pipe, for example). Assume there is a global integer `pipe`, which is set to -1 when the pipe is closed, and a global lock `lock`, which is used for synchronization. Here is the code:

```
void MakeSurePipeIsOpen() {
    mutex_lock(&lock);
    if (pipe == -1)
        pipe = open(‘‘/tmp/fifo’’, O_WRONLY);
    mutex_unlock(&lock);
}
```

However, you get clever, and decide to re-write the code as follows:

```
void MakeSurePipeIsOpen() {
    if (pipe == -1) {
        mutex_lock(&lock);
        if (pipe == -1)
            pipe = open(‘‘/tmp/fifo’’, O_WRONLY);
        mutex_unlock(&lock);
    }
}
```

Does this code still work correctly? If so, what advantage do we gain by using this implementation? If not, why doesn't it work?

7. Assume you are implementing a producer-consumer shared buffer (which can be used by producer threads to pass data to consumer threads), but that the buffer is *unbounded*; in other words, it does not have a limit as to how big it can get.

   a) How many condition variables will you need in order to implement this buffer properly, and why?

   b) How is this different than a standard bounded buffer implementation?

8. For deadlock to occur, four conditions must hold: *mutual exclusion*, *hold and wait*, *no preemption*, and *circular wait*. If any one condition does not hold, no deadlock can occur. Assume we want to allow “preemption”, and thus get out of deadlocks; in other words, if a deadlock is detected, we will forcibly take a lock away from a thread; by repeatedly doing this, we will eventually undo the deadlock. What new problems are introduced by this preemptive approach?

9. Someone has written new memory allocator to replace the standard malloc()/free() implementation. It works as follows: one half of available memory is divided into fixed-sized units of 4KB, and the other half is managed by a best-fit free list. If an allocation request is less than or equal to 4KB and there is space in the fixed-sized half, a 4KB unit is allocated from the fixed-sized half; otherwise, the best-fit algorithm is used over the other half of memory, and the requested size is returned (if space is available).

   a) Assuming 32KB of total memory is available, what series of allocation requests will most quickly lead to all of memory getting allocated, all while requesting the least total amount of memory?

   b) What type(s) of fragmentation occurs with this new allocator?

10. A mechanism that can be used for synchronization is the ability to turn on and off interrupts.

a) How can you use this to implement a critical section?

b) Why does does it work?

c) Why is this generally a bad idea?

11. In class, we talked about two kinds of message sends: *blocking* and *non-blocking*. In communicating through a Unix pipe, consider the sender side (i.e., the side doing the `write()` call to the pipe). Is the `write()` to a pipe blocking, non-blocking, or both? **Explain.**

12. For the following question, please **circle all answers that apply.** A translation lookaside buffer (TLB) is generally used to:

    (a) translate virtual page numbers into physical page numbers

    (b) translate physical page numbers into virtual page numbers

    (c) make segmentation have the benefits of a pure paging approach

    (d) translate the addresses generated by loads

    (e) translate the addresses generated by stores

    (f) translate the addresses generated by instruction fetches

    (g) remove the need for a full-sized page table

    (h) make translations happen quickly

Part II: Longer Questions

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

1. **Staying In-Bounds.**

   You are dealing with a system that performs *static relocation*. In static relocation, a *loader* rewrites the addresses of a process as it is getting loaded into the system so as to "relocate" the address space of that process to an arbitrary address in physical memory. In this system, **all programs are compiled as if they will get loaded at address 1000**. Then, when the loader is "loading a process", it must re-write any addresses within the program in order to generate addresses at the correct offset in physical memory.

```
load 0(R1), R2   # loads value at address 'R1 + 0' into R2
add R2, 5, R2    # add 5 to R2
store 0(R1), R2  # store value at address 'R1 + 0' back into R2
```

   a): Assuming that the process gets loaded at **physical address 2500**, how would the loader re-write the statements above so as to provide proper static relocation?

   b): Let's say we want to implement some additional checks in our static relocation scheme. Specifically, we want to make sure that all addresses generated by the process do not extend beyond its address space. If an address is outside of the limit, the program should just be forced to exit. What would we have to do before each `load` and `store` instruction in order to guarantee that they stay within the address space of the process?

   c): In contrast with static relocation, **dynamic relocation** is a hardware approach to relocating the address space of a process in physical memory. What **hardware** is required to implement dynamic relocation?

   d): If you contrast software-based static relocation with the extra checks (as described in this question in part (b)) to traditional hardware-based dynamic relocation, are they equivalent, or does one approach give you more capabilities than the other? **Explain.**

2. **Synchronization: Primitive?**

Different hardware architectures provide different low-level instructions to allow one to implement synchronization primitives. In this question, we will examine two different sets of synchronization instructions (available on two different architectures), and will use each of them to implement a critical section.

Load-linked, store-conditional: The first hardware primitive is actually a pair of instructions available on the MIPS architecture, and they are called the *load-linked* and *store-conditional* instructions. They are used in combination to build mutual exclusion.

```
    ll <address>, RD
    sc <address>, RS
```

In the load-linked instruction (`ll`), the value at address `<address>` is placed into the register `RD`, much like a normal load. With the store-conditional instruction, the value inside of the register `RS` is placed into the value at `<address>`, *if the value at* `<address>` *has not been changed by some other thread since the load-linked instruction (*`ll`*) was executed.* If the store-conditional succeeds (and stores the value in `RS` into the address `<address>`), the register `RS` will be set to the value 1; if the store-conditional fails (in other words, someone else has updated the value at `<address>` in the meanwhile), the store-conditional does not update the value at `<address>` and RS is set to the value 0.

Fetch-and-add: The second synchronization instruction is available on the now defunct Alpha architecture, and is called *atomic fetch-and-add* (abbreviated `fetchadd`). The format of the `fetchadd` instruction is as follows:

```
    fetchadd <address>, RS
```

where `<address>` holds an address of some variable, and register `RS` holds an integer value. When `fetchadd` executes, it atomically adds the value inside of `RS` to the variable stored at `<address>`.

The code that must be implemented in properly synchronized form is our standard synchronization routine:

```
int balance = 0; // global variable, accessible by all threads.

void update(int amount) {
    balance = balance + amount; // must synchronize access to 'balance'!
}
```

Your job: implement the `update()` routine so that it is properly synchronized, using the different synchronization instructions available. In other words, in part a), implement `update()` by using the load-linked and store-conditional instructions (but not the atomic fetch-and-add or compare-and-swap). In part c), use just the fetch-and-add. Of course, in both parts, you may use other standard instructions such as loads, adds, stores, and so forth.

Assumptions: Assume you have 16 registers at your disposal (you won't need nearly that many), and call them `R1` through `R16`. Also, assume that when `update()` is called, the value of the `amount` variable is placed inside of register `R1`.

You may need to use some other instructions to implement the correct code. To get you started, this is what the unsynchronized version of the `update()` routine looks like:

```
    load <balance>, R2   # load account balance into R2
    add R1, R2, R3       # add amount (R1) and balance (R2), result in R3
    store <balance>, R3  # store value of R3 back into balance
```

(continued on next page)

You may also need to use a `branch` instruction of some kind. If so, just write some pseudo-C (instead of assembly) and use `goto` statements and labels.

```
top: load <variable>, R1
     if (R1 == 1) goto top
```

In the code snippet above, the variable `variable` keeps getting checked to see if its value has become anything other than 1; as long as it stays at 1, the code keeps branching back to the label `top`.

a): Implement the `update()` routine with the **Load-linked, store-conditional** instructions.

b): Are there any limitations or problems with your solution to part (a)? If so, please describe them. If not, please say why.

c): Implement the `update()` routine with the **Atomic fetch-and-add** instruction.

d): Which of is more appropriate to use in implementing the `update()` routine, the load-linked/store-conditional, or the atomic fetch-and-add? **Explain.**