The University of Texas at Austin
CS 372H Introduction to Operating Systems: Honors: Spring 2012

Midterm Exam

- This exam is **80 minutes**. Stop writing when "time" is called. *You must turn in your exam; we will not collect it.* Do not get up or pack up between 75 and 80 minutes. The instructor will leave the room 83 minutes after the exam begins and will not accept exams outside the room.
- There are **12** problems in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. You may want to skim all questions before starting.
- **This exam is closed book and notes. You may not use electronics: phones, calculators, laptops, etc.** You may refer to ONE two-sided 8.5x11" sheet with 10 point or larger Times New Roman font, 1 inch or larger margins, and a maximum of 55 lines per side.
- If you find a question unclear or ambiguous, be sure to write any assumptions you make.
- Follow the instructions: if they ask you to justify something, explain your reasoning and any important assumptions. **Write brief, precise answers. Rambling brain dumps will not work and will waste time.** Think before you start writing so that you can answer crisply. Be neat. If we can't understand your answer, we can't give you credit!
- *There is no credit for leaving the "small" problems blank.* However, if a problem is worth 10 or more points, then completely blank answers will get 15%-20% of the credit. For these problems, if you attempt the problem, you start at zero points for the problem. Note that by *problem* we mean numbered questions for which a point total is listed. *Sub-problems* with no points listed are not eligible for this treatment. Thus, if you attempt any sub-problem, you may as well attempt the other sub-problems in the problem.
- Don't linger. If you know the answer, give it, and move on.
- **Write your name and UT EID on this cover sheet and on the bottom of every page of the exam.**

Do not write in the boxes below.

I (xx/20)	II (xx/36)	III (xx/19)	IV (xx/25)	Total (xx/100)

Name: **UT EID:**

I Short answer (20 points total)

1. [2 points] Fill in the blank:

In the context of programming errors, we said in class that confidence and ____________________ are inversely correlated.

2. [18 points] Consider a fragment of a program `foo` in a file called `moo.c`:

```
void moo(char* to_print)
{
    const char* cheerful = "printf is awesome\n";
    printf("%s", to_print);

    write(1, cheerful, strlen(cheerful));
}
```

Between the time that the programmer writes the code above and the time that the program's output actually appears, there are layers of indirection and references that have to be resolved. This problem will explore when these resolutions happen: compile time, assembly time, link time, load time, or run time.

Compile time, as usual, means "while the compiler is running, including the instant it finishes". Same for assembly time and link time. We use the term *load time* to refer to the moment when the OS loader brings the program image into memory in response to `exec()`. *Run time* means "any time after the program begins running".

For each piece of information below, place an X in the column that indicates the *earliest* that the information is available and explicit. Assume that the code does not modify itself once loaded.

The number of arguments that `moo()` passes to `printf()` is known at:

compile time	assembly time	link time	load time	run time

The number of arguments that `printf()` receives the first time that it is called in `foo` is known at: (Note that `printf()`'s first caller may not be `moo()`.)

compile time	assembly time	link time	load time	run time

The virtual addresses of the instructions that implement `printf()` are known at:

compile time	assembly time	link time	load time	run time

The virtual addresses of any instructions that call `printf()` *directly* are known at:

compile time	assembly time	link time	load time	run time

The virtual addresses of any instructions that call `printf()` *indirectly* are known at:

compile time	assembly time	link time	load time	run time

The virtual address (on the stack) of `printf()`'s second argument is known at:

compile time	assembly time	link time	load time	run time

Whether file descriptor 1 represents a device, a pipe, a file (or none of these) is known at:

compile time	assembly time	link time	load time	run time

Whether the output of `printf()` goes to file descriptor 1 is known at:

compile time	assembly time	link time	load time	run time

The number of symbolic references used by `moo.c` is known at:

compile time	assembly time	link time	load time	run time

II Concurrency and multicore (36 points total)

3. [3 points] Assume that the code below runs on machine X. Machine X does *not* offer sequential consistency (also, do *not* assume that Machine X offers the x86's memory model). One thread executes the function `p1()`, and another thread concurrently executes the function `p2()`. Assume that `data` and `ready` are initialized to 0 before the two threads begin executing.

```
int data = 0, ready = 0;

void p1 () {
    data = 2000;
    ready = 1;
}
int p2 () {
    while (!ready) {}
    return data;
}
```

What are the possible return values of `p2()`?

Circle the BEST answer below:

A 0, only.

B 2000, only.

C 0 or 2000.

D The return value of `p2()` is unspecified; that is, `p2()` could return anything.

4. [4 points] Recall the graph that we saw in class, which compared the performance of MCS spinlocks to the spinlocks that were standard in Linux at the time that the graph was produced (in year 2008). The graph suggested that although MCS spinlocks have asymptotically better performance, they have a higher fixed cost. (The larger point was that the fairness of MCS spinlocks has a cost.)

What is the likely source of MCS spinlocks' higher fixed cost? State your answer briefly below.

5. [5 points] Consider a job (meaning a computation plus an input) that an application developer has decomposed into several (kernel-level) threads. The application developer initially runs the job on a single-processor machine. Then, the developer adds three CPUs to the machine without modifying the code or the input. It turns out that adding these CPUs might make the performance *worse*. Why?

Below, explain why adding CPUs could make performance *worse*. There are multiple ways to answer this question: you can explain in general what's going on, or you can give an example, etc.

6. [4 points] Recall that locks are in conflict with modularity.

Give one example of this conflict. (We discussed a number of examples in class; you only need to give one here.) Be brief; there is far more space below than you need.

7. [20 points] In this problem you will write code to drive a simplified linear accelerator ("LinAc") that is reminiscent of the Therac-25. This LinAc is an I/O device that is attached to your (32-bit) computer. The LinAc has a *mode setting* and a *position setting* that are (unfortunately) set independently, via different hardware registers that your computer accesses through memory mapped I/O.

The host's interface to the linear accelerator hardware is as follows:

```
#define LINAC_MODE     0xfd000000  /* assume this is both virtual and physical */
#define LINAC_POS      0xfd000400  /* ditto */
#define LINAC_BEAM     0xfd000800  /* ditto */

#define LINAC_BUSY     0x00000001
#define MODE_PHOTON    0x00000010
#define MODE_ELECTRON  0x00000020
#define POS_FLATTENER  0x00000040
#define POS_BENDER     0x00000080
```

- To set the mode of the linear accelerator, the software writes to memory address `LINAC_MODE`. Legal values to store are `MODE_PHOTON` and `MODE_ELECTRON`. After such a write, a read from memory address `LINAC_MODE` returns either `LINAC_BUSY` (meaning that the linear accelerator is adjusting) or the value just stored (meaning that the adjustment is done).
- The position of the linear accelerator is set analogously: the host software writes to address `LINAC_POS`. Then, reading from `LINAC_POS` returns either the requested value or `LINAC_BUSY`.
- Enabling the radiation beam is similar: host software writes to `LINAC_BEAM`. The LinAc decides when to turn off the beam, at which point reads from `LINAC_BEAM` return 0.

Your first task is to implement part of a primitive device driver to interface with the hardware. Here is the interface exposed by the device driver:

```
uint32_t LINAC_GET_MODE();       //   Returns the LinAc's current mode.
                                 // YOU WILL IMPLEMENT THIS FUNCTION BELOW.


void LINAC_SET_MODE(uint32_t mode);  //   Sets the requested mode and returns
                                     //   _only after the machine has completed
                                     //   the adjustment_.
                                     // YOU WILL IMPLEMENT THIS FUNCTION BELOW.


uint32_t LINAC_GET_POS();             // analogous to LINAC_GET_MODE


void LINAC_SET_POS(uint32_t pos);     // analogous to LINAC_SET_MODE


void LINAC_BEAM_ON();              // turns on beam; returns when beam off
```

Fill in the function below, whose purpose is to return the current mode of the LinAc:

```
uint32_t LINAC_GET_MODE()
{
     volatile uint32_t* mode_addr = LINAC_MODE;
     // FILL THIS IN. YOU NEED ONLY ONE LINE.




}
```

Fill in the function below. You should return only after the machine has completed the adjustment. You should not busy wait; thus, `sleep()`ing for an appropriate interval is acceptable.

```
void LINAC_SET_MODE(uint32_t mode)
{
     volatile uint32_t* mode_addr = LINAC_MODE;
     // FILL THIS IN. DON'T BUSY WAIT.






}
```

Next, you will synchronize access to the LinAc.

Note: the hardware interface to the linear accelerator permits four combinations of (mode, position). However, the software should only ever put the machine in one of two configurations: (`MODE_PHOTON`, `POS_FLATTENER`) and (`MODE_ELECTRON`, `POS_BENDER`).

The above note concerns safety. For performance, however, we observe that setting the mode and position each take on the order of seconds to complete. Thus, we want these tasks to proceed in parallel. To balance the safety and performance goals, you decide to structure your software as follows:

- You create a single producer thread and a pool of multiple (more than 5) consumer threads that interact through a message queue. Specifically, the producer thread interprets the operator's requests, places descriptions of those requests in `Task` structures, and enqueues the `Tasks`; the consumer threads dequeue `Tasks` and carry them out. (We saw code for a message queue in class.)
- To avoid mixing up modes and positions, you establish the following invariants:
  1. The task of mode setting can run only if (a) no other mode setting task is in progress; and (b) no beam enabling task is in progress.
  2. The task of position setting can run only if (a) no other position setting task is in progress; and (b) no beam enabling task is in progress.
  3. The task of beam enabling can run only if (a) there is no mode setting, position setting, or beam enabling task in progress; and (b) the machine's actual position and mode correspond to the operator's request at the time that the operator requested "beam on". *If a beam enable task is otherwise ready to run but condition (b) is violated, the beam enabling task must return an error.*

The monitor defined and partially implemented below is intended to respect the invariants above. However, it is missing a method that you must fill in. For context, note that the monitor is used by the consumer threads; the consumer thread implementation is at the end of this problem.

```
class LinacMonitor {

    public:
        void set_mode(uint32_t mode);
        void set_position(uint32_t pos);

        // YOU WILL IMPLEMENT THIS FUNCTION ON THE NEXT PAGE
        void beam_on(uint32_t mode, uint32_t pos);

    private:
        Mutex mutex;
        Cond cv;

        bool setting_mode;
        bool setting_position;
        bool delivering_radiation;
};

void
LinacMonitor::LinacMonitor()
{
    setting_mode = false;
    setting_position = false;
    delivering_radiation = false;
}

void
LinacMonitor::set_mode(uint32_t mode)
{
    mutex.acquire();

    while (setting_mode || delivering_radiation)
         cv.wait(&mutex);

    setting_mode = true;

    LINAC_SET_MODE(mode);

    setting_mode = false;

    cv.broadcast(&mutex);

    mutex.release();
}
```

```
void
LinacMonitor::set_pos(uint32_t pos)
{
    mutex.acquire();

    while (setting_pos || delivering_radiation)
         cv.wait(&mutex);

    setting_pos = true;

    LINAC_SET_POS(pos);

    setting_pos = false;

    cv.broadcast(&mutex);

    mutex.release();
}

int
LinacMonitor::beam_on(uint32_t mode, uint32_t pos)
{
    // FILL THIS IN. Return -1 if there is an error and 0 otherwise.
    // You must handle operator error, among others.











}
```

```
typedef enum {PLEASE_SET_POS, PLEASE_SET_MODE, PLEASE_TURN_ON_BEAM} req_t;

struct Task {
    req_t request_type;
    uint32_t mode;
    uint32_t pos;
};

MessageBuffer msg_buf;
LinacMonitor linac_monitor;

// there are many threads, each of which is running this consumer() function.
void consumer()
{
    struct Task task;

    while (1) {

        task = msg_buf.dequeue();

        switch (task.request_type) {

            case PLEASE_SET_MODE:
                linac_monitor.set_mode(task.mode);
                break;

            case PLEASE_SET_POS:
                linac_monitor.set_pos(task.pos);
                break;

            case PLEASE_TURN_ON_BEAM:
                if (linac_monitor.beam_on(task.mode, task.pos))
                     raise_error();
                break;

            default:
                     raise_error();
        }
    }
}
```

III Readings (19 points total)

8. [3 points] This question concerns the assigned reading, “Andy Tanenbaum hasn’t learned anything” (which was a newsgroup posting by Rob Pike et al., 1992).

Briefly state one of the things that Andy Tanenbaum hasn’t learned, according to Pike et al. Your answer should NOT use the word “microkernel”.

9. [4 points] This question concerns “The xv6 book”, chapter 3 (which was assigned reading). This chapter discusses recursive locks, which are locks that permit a CPU to acquire the lock multiple times without releasing it. What is the chapter’s principal argument against recursive locks?

Circle the BEST answer:

A Recursive locks sometimes violate lock orderings.

B Recursive locks make it hard to reason about invariants.

C Recursive locks are in conflict with transparency and modularity.

D Recursive locks can lead to endless recursion.

E Recursive locks make it hard to avoid priority inversion.

F This question concerns “The xv6 book”, chapter 3 (which was assigned reading). This chapter discusses recursive locks, which are …

10. [12 points] Recall that in the paper "Efficient Software-Based Fault Isolation" (Wahbe et al., Proc. SOSP 1993), the authors propose to sandbox `STORE R1, R0` with the following instructions:

```
Ra <- R0 & Re    // zero out segment ID
Ra <- Ra | Rf    // replace with the valid segment ID
STORE R1, Ra
```

This question asks about how to sandbox the instruction `STORE R1, offset(R2)`. Here, the memory address for the store is `offset+R2`, and `offset` is a 17-bit signed number (meaning that it can represent numbers between -64KB and 64KB). A naive way to sandbox this instruction is:

```
Ra <- offset + R2
Ra <- Ra & Re
Ra <- Ra | Rf
STORE R1, Ra
```

However, the approach above requires three instructions of overhead. Instead, the authors propose sandboxing `STORE R1, offset(R2)` using (a) overhead of *two* instructions and (b) mapping two "guard" regions (that is, invalid areas of virtual memory; stores to these regions will generate page faults and justly crash the program). This question asks about both of these aspects.

Below, give the pseudocode (in the same format above) for sandboxing `STORE R1, offset(R2)` with an overhead of two instructions.

What are the address ranges of the two guard regions? You should express your answer concisely, and you may invent some notation if that will help.

Name: **UT EID:**

IV JOS + essay (25 points total)

11. [5 points] Recall that in lab 4a, you enabled multiprocessor support. As part of doing so, you ensured that each CPU has its own kernel stack. You also used a big kernel lock (BKL) to ensure that at most one CPU runs in the kernel at once. This question asks: since the big kernel lock provides mutual exclusion, why do you need separate kernel stacks for each CPU?

Describe a sequence of events that would create a problem if you used a single kernel stack, shared among all CPUs. Assume that the big kernel lock is acquired and used correctly. Be brief; you do not need more than a few sentences.

12. [20 points] This question asks you to write a short essay that refutes, supports, or partially supports the following statement:

JOS is an exokernel.

Please note that there is no single correct answer. To get full credit, your answer must be well-structured, and it should draw on the specifics of JOS and the exokernel vision as described in "Exokernel: An Operating System Architecture for Application-Level Resource Management" (Engler et al., Proc. SOSP 1995). You do not need more than one or two paragraphs. *Do not brain dump your knowledge about the relevant systems. Instead, think and outline before you start writing. Structure and effective argument are important here; the exact length is not.* You may use the next page too.

End of Midterm