

**The University of Texas at Austin**  
**CS 372H Introduction to Operating Systems: Honors: Spring 2011**  
**Midterm Exam**

- This exam is **75 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 70 and 75 minutes. The instructor will leave the room 78 minutes after the exam begins and will not accept exams outside the room.
- There are **13** questions 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 you can answer crisply. Be neat. If we can’t understand your answer, we can’t give you credit!
- To discourage guessing and brain dumps, we will give 15%-20% of the credit for any problem left completely blank. If you attempt a 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.
- The exception is the True/False problems. There, we grade by individual True/False item: correct items earn positive points, blank items earn 0 points, and incorrect items earn negative points. However, the minimum score on any question—that is, any group of True/False items—is 0.
- 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/31)	II (xx/31)	III (xx/18)	IV (xx/20)	Total (xx/100)

Name:

UT EID:

## I Short answer (31 points total)

**1. [10 points]** This question is about deadlock and other coding errors. Assume that the programmer uses a kernel-level threading package and monitors (mutexes and conditional variables).

**Circle True or False for each item below:**

**True / False** To guarantee no deadlock, it is sufficient to negate just one of the four conditions that contribute to deadlock.

**True / False** In practice, one way to guarantee no deadlock is to acquire mutexes in the same partial order.

**True / False** The existence of a race condition in the code implies that there is an execution schedule that will result in deadlock.

**True / False** If the code is vulnerable to starvation, then there is an execution schedule that will result in deadlock.

**True / False** If there is an execution schedule that will result in deadlock, then the code has a liveness error.

**2. [8 points]** This question is about cache replacement. Suppose that there is a page reference string (also known as a reference pattern; in class, these were strings of the form  $A, B, C, A, \dots$ ). You don't know anything about the reference string except that it has length  $p$  and has  $n$  distinct pages occurring in it. Suppose also that there is a cache (of pages) that these page references will encounter. You don't know anything about the cache except that it has  $m$  entries and that it starts out empty. From this description,  $p \geq n$ . Also, suppose  $n > m$  and  $m > 0$ .

**What is the *minimum* possible number of cache misses, over all combinations of reference pattern and cache replacement policy? You do not need to justify your answer.**

**What is the *maximum* possible number of cache misses, over all combinations of reference pattern and cache replacement policy? You do not need to justify your answer.**

**3. [5 points]** In class, we said that whereas test-and-set (or test and test-and-set) spinlocks are not fair, MCS spinlocks are fair. This question asks: what is the precise reason that MCS spinlocks, as implemented on an x86, are fair? (NOTE: the answer is not that a thread trying to acquire a lock is spinning on nearby or cached memory. That fact is true, but it is not the fundamental reason that MCS spinlocks are fair.)

**State briefly below the precise reason that MCS spinlocks are fair.**

**4. [8 points]** This question is about scheduling. Assume that the jobs to be scheduled each take finite time.

**Circle True or False for each item below:**

**True / False** Round-robin scheduling optimizes (minimizes) average completion time (defined as the time from when a process enters the system to when it completes).

**True / False** Priority scheduling optimizes (minimizes) average waiting time.

**True / False** If Shortest Remaining Time to Completion First (a preemptive scheduling discipline often written as SRTCF) could be implemented, it would optimize (maximize) CPU utilization (defined as fraction of CPU cycles that are dedicated to process execution).

**True / False** During periods when the processor spends all of its cycles handling interrupts, the process scheduler is irrelevant to the machine's performance.

## II JOS and virtual memory (31 points total)

- 5. [9 points]** Consider a JOS environment: `struct Env e`. Suppose the JOS kernel first sets up the page directory and page tables for `e` and then executes the following line:

```
e->env_pgd[0] = e->env_cr3 | PTE_P | PTE_W | PTE_U;
```

**Fill in the three blanks below, and note that the definition below the question may be helpful:**

The above line exposes `e`'s \_\_\_\_\_ (i) \_\_\_\_\_ to `e`'s user-space code; that code sees the exposed data or data structures at virtual addresses \_\_\_\_\_ (ii) \_\_\_\_\_ through  $2^{22} - 1$  (4 megabytes minus one), with the following associated memory permissions: \_\_\_\_\_ (iii) \_\_\_\_\_.

(i):

(ii):

(iii):

Here is a partial definition of `struct Env`:

```
struct Env {
    struct Trapframe env_tf;          // Saved registers
    LIST_ENTRY(Env) env_link;         // Free list link pointers
    envid_t env_id;                  // Unique environment identifier

    .....

    // Address space
    pde_t *env_pgd;                 // Kernel virtual address of page dir
    physaddr_t env_cr3;              // Physical address of page dir
};
```

Name:

UT EID:

**6. [6 points]** In JOS, to enter kernel mode through a system call, a user-level environment executes:

```
int $0x30 ; this is in lib/syscall.c
```

Before the line above, the environment has placed the system call number in %ax and the system call arguments in the other registers (%dx, %cx, etc.). If the JOS kernel has set up the IDT correctly, then, after the int call above, the processor begins executing in kernel mode, with the value of %cr3 after the transfer into kernel mode equal to the value of %cr3 before the transfer. Thus, after the transfer, the processor's %cr3 points to an environment-specific page directory. Yet, the processor can load from, and store to, kernel memory—without generating page faults.

**Why is there no page fault after the transfer? Which of the specific steps that you took, as part of doing the labs, ensures that page faults do not happen? Assume that the IDT is set up correctly (i.e., we are not asking about setting up the IDT). You do not need a lot of space; state your answer briefly below:**

**7. [8 points]** In lab 4a, you implemented the `sys_page_map()` system call. An environment makes this call to request that the kernel map a given page of memory,  $P$ , into a target environment's virtual address space. The target environment can be the caller itself or any other environment. Which of the following are arguments to this system call?

**Circle ALL that apply:**

- A** The environment id of the target.
- B** The target's data segment descriptor.
- C** The physical address of the target's page directory.
- D** The virtual address that  $P$  should have in the target.
- E** The physical address of  $P$ .
- F** The permissions that the kernel should associate to the mapping.

**8. [8 points]** Note that there are two sub-parts here. Suppose that %fs on the x86 is currently referencing a segment descriptor for which the base is 0x1000 0000, the limit is 0x4000 0000, and the permissions are maximally permissive. Now suppose that a process executes a memory store to the segment given by %fs, using offset 0x3000. The code might look like this:

```
movl $0x3000, %eax  
movl $0xdeadbeef, %fs:(%eax)
```

**As a result of the instructions above, the processor stores the value 0xdeadbeef to which *linear* address? If the processor would raise an error, write “error”. If you do not have enough information to answer the question, write “not enough information”.**

Now suppose that the descriptor pointed to by %fs has base set to 0xf000 0000 and the limit set to 0x1000, and suppose that the programmer’s intended offset is again 0x3000.

**With the new segment descriptor, the result of the above instructions is to store the value 0xdeadbeef to which *linear* address? If the processor would raise an error, write “error”. If you do not have enough information to answer the question, write “not enough information”.**

### III The readings (18 points total)

**9. [6 points]** Note that this question consists of a multiple choice followed by another question. Tanenbaum's book has a section about The Ostrich Algorithm, as a way of addressing a particular class of software errors. Which class of software errors is Tanenbaum proposing the algorithm for?

**Circle the BEST answer below:**

- A Overflows from addition of fixed-width integers
- B Starvation
- C Deadlock
- D Race conditions
- E Priority inversion
- F Performance problems from coarse-grained locking
- G Broken modularity from locking

**State below how the algorithm works at a high level (you do not need to give pseudocode):**

**10. [8 points]** This question is about the assigned paper on the Therac-25 (“An Investigation of the Therac-25 Accidents”) and linear accelerator disasters.

**Circle True or False for each item below:**

**True / False** The authors of the paper explain the race conditions that they uncovered when they read the source code for the Therac-25.

**True / False** The authors of the paper explain the liveness bugs that they uncovered when they read the source code for the Therac-25.

**True / False** Previous experience with the interface and operation of the machine led Therac-25 operators to *expect* the machine to malfunction.

**True / False** The thesis of the New York Times articles on linear accelerator disasters is that software developers should be licensed.

**11. [4 points]** From which of our assigned readings is the following excerpt drawn?

Throughout the paper I use examples written in Modula-2+. These should be readily understandable by anyone familiar with the Algol and Pascal family of languages.

**Identify the author or the reading:**

Name:

UT EID:

## IV Shared memory multiprogramming (20 points total)

**12. [8 points]** This question is about the correctness of the pseudocode below. The programmer intends that g() not execute unless f() has executed; the two functions are called by different threads. Assume POSIX thread semantics (Hansen semantics); that is, the thread package provides the same guarantees that it did in lab T. *Read the code carefully.*

```

int f_ran = FALSE;
Mutex mutex;
Cond cv;

// called by a thread
Monitor::t1() {
    mutex.acquire();
    if (f_ran == FALSE)
        cv.wait(&mutex);

    g(); /* <-- It is an error if g() executes before f() */

    mutex.release();
}

// called by another thread
Monitor::t2() {
    mutex.acquire();

    f();
    f_ran = TRUE;

    cv.broadcast(&mutex);
    mutex.release();
}

```

Under which conditions is the above pseudocode correct?

**Circle the BEST answer:**

- A** The code executes on a single processor.
- B** The memory model is sequential consistency.
- C** The threads are user-level threads.
- D** The system contains only two threads, one that calls t1() and one that calls t2().
- E** The broadcast() is replaced with a signal().
- F** The code is correct if conditions **A**, **C**, and **D** all hold simultaneously.
- G** The code is correct if conditions **B** and **D** both hold simultaneously.
- H** The code is correct if conditions **A**, **C**, **D**, and **E** all hold simultaneously.
- I** None of the above.

Name:

UT EID:

**13. [12 points]** Consider the function `doublecheck_alloc()` below, which is intended to be invoked from multiple threads on a multiprocessor machine. Its purpose is to avoid a mutex acquisition in the common case that `ptr` is already initialized. The requirements for this function are:

- (i) `doublecheck_alloc()` must call `alloc_foo()` no more than once over the whole execution.
- (ii) A caller of `doublecheck_alloc()` must, after the function returns, observe `ptr` as non-zero.

The machine does **not** offer sequential consistency. Thus, a processor is not guaranteed to see the memory operations of another processor in program order. However, each of `mutex_acquire()` and `mutex_release()` is implemented correctly; in particular, each of them internally contains a memory barrier (`mfence` on the x86). Recall that `mfence` ensures that all memory operations before the `mfence` barrier appear to all processors to have executed before all memory operations after the `mfence` barrier.

On the other hand, the compiler **preserves** program order (it does not reorder instructions).

```

struct foo {
    int abc;
    int def;
};

static int ready = 0;
static mutex_t mutex;
static struct foo* ptr = 0;

void
doublecheck_alloc()
{
    if (!ready) { /* <-- accesses shared variable w/out holding mutex */
        mutex_acquire(&mutex);
        if (!ready) {
            ptr = alloc_foo(); /* <-- sets ptr to be non-zero */
            ready = 1;
        }
        mutex_release(&mutex);
    }
    return;
}

```

The above code certainly violates our coding standards, but this problem is about whether it violates requirements (i) and (ii), above. The questions are given on the next page.

Does the code above violate requirement **(i)**? In other words, can `alloc_foo()` be called more than once from `doublecheck_alloc()`? If so, give an interleaved execution or an interleaving of memory operations as observed by one of the threads. If not, argue from invariants.

Does the code above violate requirement **(ii)**? In other words, can the caller of `doublecheck_alloc()` observe `ptr == 0` after `doublecheck_alloc()` returns? If so, give an interleaved execution or an interleaving of memory operations as observed by one of the threads. If not, argue from invariants.

**End of Midterm**  
**Enjoy Spring Break!!**

**Name:**

**UT EID:**