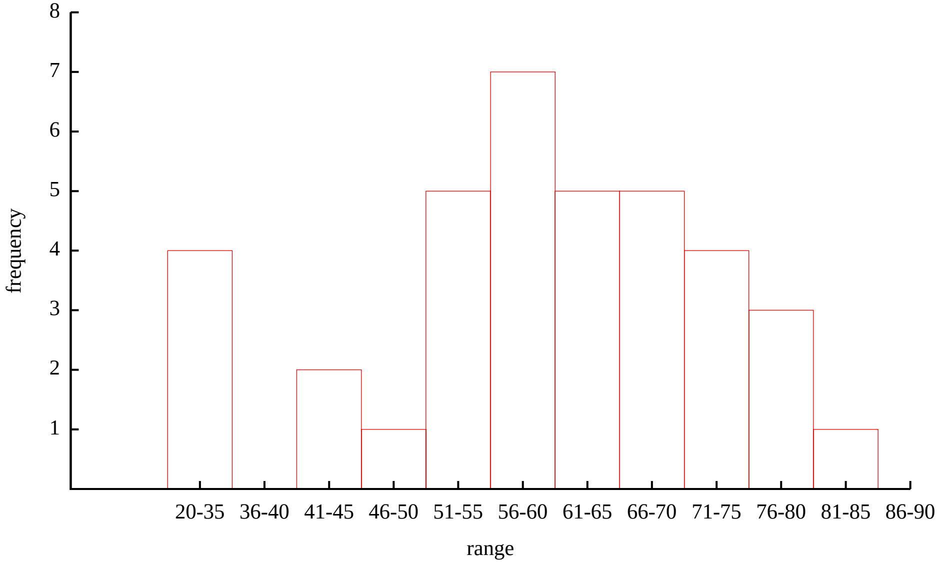
frequency
8
7
6
5
4
3
2
1
20-35
36-40
41-45
46-50
51-55
56-60
61-65
66-70
71-75
76-80
81-85
86-90
range

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: Solutions **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. Unfortunately, for many programming problems, negating even just one of the conditions is hard.

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

False. I think almost everyone got this wrong, but the statement as literally worded is false. Acquiring mutexes in the same partial order does not *guarantee* no deadlock. For most reasonable cases, acquiring mutexes in a partial order is *necessary* to avoid deadlock but not sufficient. The reason is that condition variables are part of the mix (per the given above). Thus, deadlock can happen despite following the partial order. One way is through the nested monitor example. Another example is in lab T (where you all followed the partial order but wherein your code deadlocked). Here is a third example: while(something) wait (cv1); signal(cv2); then in another thread: while(somethingelse) wait(cv2); signal(cv1);

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

False. Race conditions and deadlocks are separate kinds of coding errors.

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

False. Deadlock can result in starvation, but starvation can happen without deadlock, as in the unfair readers-writers lock that we saw.

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

True. Code that can deadlock is not live, as such code is not guaranteed to make progress in all cases.

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 $A, B, C, A, \ldots$). 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.

Each page must certainly generate a cache miss (the first time that the page is presented to the cache). Thus, the minimum is n. The best that could happen is that each of these cache misses is the only one for that page. To pick one of many examples of how the minimum could be achieved (independent of cache replacement policy), imagine that the access pattern is $A, A, A, A, A, A, A, B, B, B, B, B, C, C, ..., Z, Z, Z, Z, Z,$ with, say, 10 entries in the cache. The number of misses will be $n = 26$, even though $p > n$.

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.

The worst thing that could happen is that every single reference generates a cache miss. Thus, the maximum is p. Here's a case that encounters that behavior (independent of cache replacement policy). Imagine that the access pattern is $A, B, C, ..., Z, A, B, C, ..., Z$, and imagine that $m = 1$. Then all p references generate a miss. Here is another one. Say that $m = 25$ and that the access pattern is $A, B, C, ..., Z$. The number of misses is $p = n = 26$.

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.

The precise reason is that a would-be acquirer on a contended lock adds itself to the tail of a queue. The would-be acquirers thus get the lock in FIFO order.

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). False. Round-robin is not optimized to get jobs out of the system quickly.

True / False Priority scheduling optimizes (minimizes) average waiting time. False. If the high priority jobs takes a long time and the low priority jobs take less time, then priority scheduling will do worse on the waiting time metric.

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). False. SRTCF might have to spend time in context switches that a discipline that runs jobs to completion does not have to spend. The key here is that SRTCF was listed as preemptive. Any preemptive discipline pays a context switching cost. As discussed in class and the book, SRTCF optimizes average waiting time, not CPU utilization.

True / False During periods when the processor spends all of its cycles handling interrupts, the process scheduler is irrelevant to the machine's performance. True. If the OS is always executing in the interrupt handler, then it is not running processes.

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_pgdir[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):

(i) e's page tables. (ii) 0. (iii): user-writable (and hence readable). A lot of people, for (i), wrote "e's page directory", but this is not quite right. As a result of the given line of code, the page directory appears in the virtual address space at addresses 0 through 4KB (minus 1). But the problem mentioned that the upper limit was 4MB, which is why "page directory" was not correct. This question used the recursive page table trick, wherein the page directory contains a pointer back to itself as a way of making all 4MB of page tables appear contiguously in virtual address space.

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_pgdir;             // Kernel virtual address of page dir
        physaddr_t env_cr3;           // Physical address of page dir

};
```

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:

The kernel maps its own memory (everything above UTOP) into every environment's address space. Thus, the kernel can run with any environment's `%cr3` value loaded on the processor.

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.

A, D, F. To rule out the others: B is about segmentation, which is distinct from paging. To rule out C, E: user-level environments specify the arguments to system calls, yet user-level environments do not see physical addresses so should not be telling the kernel where the target's page directory is located. Similarly, E is wrong because if environments were allowed to specify which physical pages they could map, then chaos could result, as environments could map, and then overwrite, any physical pages on the machine. (While it is possible to imagine an even more exokernelly design wherein processes do in fact see and request physical addresses and wherein the kernel's job is just to reject invalid requests or requests with invalid permissions, JOS is not designed that way.)

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".

`0x10003000`.

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".

error.

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

C.

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

The Ostrich Algorithm means ignoring the problem. The idea is that the problem may occur so rarely that it's simply not worth it to address it in practice.

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.

Both of the above are false, as the authors did not have access to 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. (But they obviously did not expect the machine to malfunction in a way that would inflict massive radiation burns.)

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

False.

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

Name: Solutions **UT EID:**

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: Andrew Birrell's introduction to programming with threads.

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.

I. This code is never correct; none of the proposed conditions in any combination makes it correct. As we mentioned a number of times in lecture, the `if (foo) cv.wait()` pattern is always wrong. Why? Several reasons. Beyond the fact that the coding standards ruled out this pattern, the threading package is allowed to return from `wait()` at any time, a point that we discussed in lecture. (In fact, the POSIX documentation says the same thing, and that waiters should thus spin in a while loop.) This is true regardless of whether we have one or two threads, user-level or kernel-threads, signal or broadcast, sequential consistency or not, etc. Since a thread can wake at any time, even when not signaled, the code *must* check any required barrier conditions after waking from `wait()` and before proceeding.

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.

The code abides by requirement (i). At most one thread can be in between acquire/release, and after acquire, a thread either found ready==1 (in which case it does not call alloc_foo), or it found ready==0, calls alloc_foo, and sets ready==1 ensuring that no other thread executes it. The total number of times alloc_foo can be executed is thus 1.

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.

Yes, the code violates requirement (ii). From the vantage point of a processor not executing the code above, the `ready = 1` and `ptr = alloc_foo()` lines can appear to happen out of order. The execution might look like this to another thread:

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

That means that another thread can find `ready == 1` but not have observed the write to `ptr`. That is actually enough to answer the question. Some of you thought that the thread that had acquired the mutex needed to pause before releasing it, which will not lose points, but that interleaving is not actually necessary to generate the counter-example. The memory barrier doesn't say, "after a thread crosses the barrier, everything before the barrier will be visible to all processors"; it just says, "*if* a process observed a memory write that another process did after the barrier, then it has also observed every memory access that the process issued before the barrier". There's a subtle difference.

Because of issues like this, `doublecheck` is broken. Unfortunately, double-checked initialization (or double-checked locking as it's sometimes known) is an extremely common coding pattern. Even Birrell's 1989 document suggests it! Still, it's broken.

While you can fix it (in C) by adding another barrier (exercise: where?), this is not recommended, as the code is tricky to reason about. One of the points of this example is to show you why it's so important to protect global data with a mutex, even if "all" one is doing is reading memory, and even if the shortcut looks harmless.

Finally, here is a reference that explores issues with this pattern in C++ [`http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf`] and here is "The Double-Checked Locking is Broken" Declaration: [`http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`]

End of Midterm

Enjoy Spring Break!!