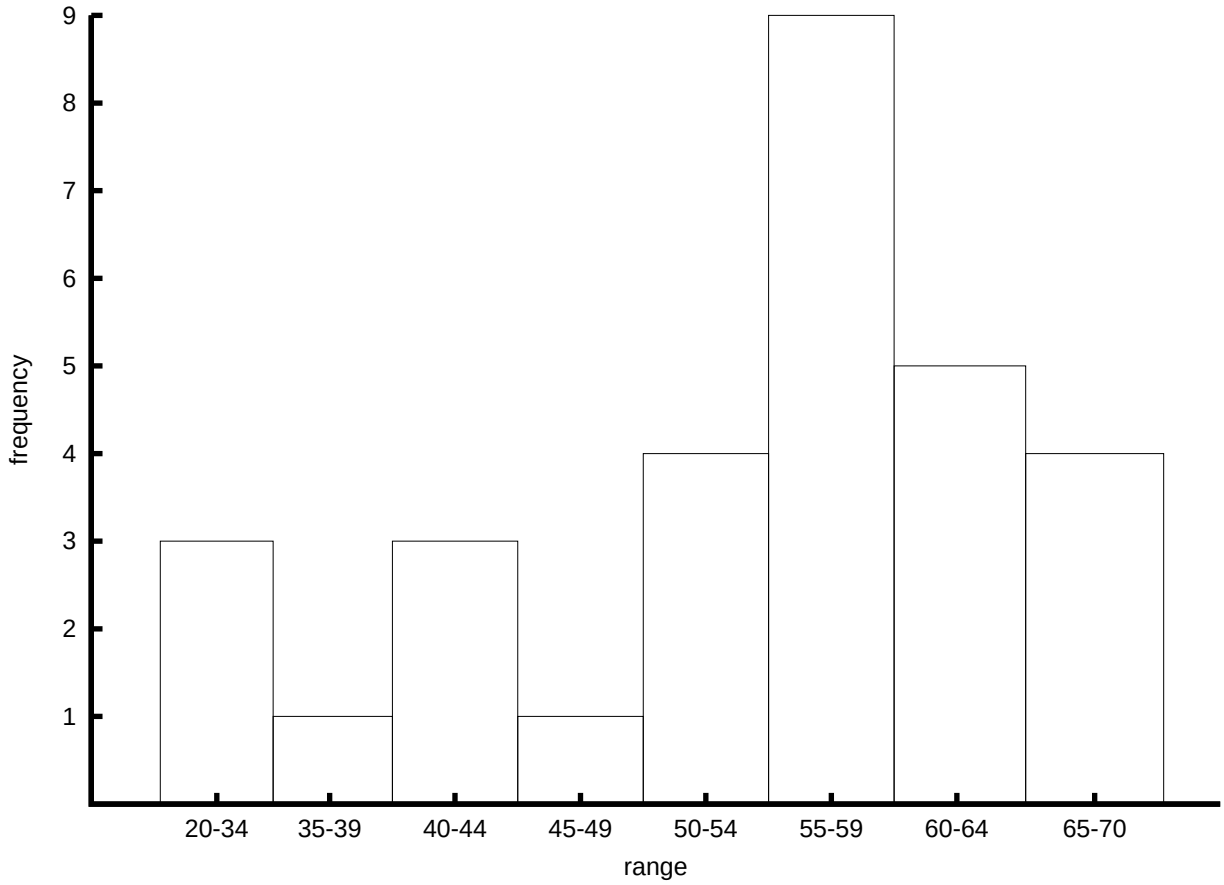
9
8
7
6
5
4
3
2
1
frequency
20-34
35-39
40-44
45-49
50-54
55-59
60-64
65-70
range

The University of Texas at Austin
CS 372H Introduction to Operating Systems: Honors: Spring 2010

Midterm

- This exam is **75 minutes**. Stop writing when "time" is announced at the end. *You must turn in your exam; we will not collect them.* 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 **16 questions** in this booklet. Many can be answered very quickly. However, some questions may be harder than others, and some questions earn more points than others. You may want to skim all questions before starting.
- **This midterm is closed book and notes. You may not use electronics: phones, PDAs, calculators, 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 60 lines per side. **Please leave your UT IDs out.**
- 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 and precise answers. Rambling brain dumps will not work and will only waste time.** *Think before you start writing* so that you can describe an 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, except where noted, give 25%-33% of the credit for any problem left *completely blank* (for example, 1 point for a 3 point question). 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, where wrong answers on individual items will cost more than leaving an item blank, regardless of the other items within the larger problem.
- Don't overthink the above. What you should do is: if you *think* you *might* know the answer, then answer the problem. If you *know* you *don't* know the answer, then leave it blank.
- 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/28)	II (xx/14)	III (xx/17)	IV (xx/11)	Total (xx/70)

Name: Solutions **UT EID:**

I Short answer (28 points total)

1. [6 points] Consider a user-level threading package running on a machine whose operating system does not offer kernel threads. Pat Q. Hacker intends to build a single application, using a single process, and is trying to decide whether to use a user-level threading package or to program in the event-driven style. Assume that the machine has two CPUs.

Circle True or False for each item below:

True / False If the user-level threading package is *cooperatively* (that is, non-preemptively) scheduled, then the user-level threading package can arrange to simultaneously run two threads, one on each processor.

True / False If the user-level threading package is *preemptively* scheduled, then the user-level threading package can arrange to simultaneously run two threads, one on each processor.

True / False Coding in the event-driven style will allow the process to simultaneously handle two events, one on each processor.

All are false. Whether the scheduling is preemptive or not doesn't change the fact that pure user-level threading packages cannot take advantage of multiple processors. The reason is that only the kernel gets to decide which process is running on a CPU, and, to the kernel, all of these different threads look like one entity, since the OS does not offer kernel threads.

The third option is false for the same reason: all events are handled by the same process, which can only run on a single CPU at a time. This is the usual case for event-driven code. While it is conceivable to imagine an event-driven system with different event loops running on different CPUs (and people have even built such systems), the problem specified that there is only one process, and there are no kernel threads; this specification ruled out the otherwise conceivable scenario.

2. [4 points] Now Pat Q. Hacker is using a system that has *kernel threads but only one processor*. Pat is writing an application and decides to structure that application as several kernel threads that share data. Pat reasons that, since the CPU can do only one thing at a time, there is no need to protect the shared data in the application code with synchronization objects like mutexes.

State whether Pat's reasoning is correct, and justify your answer *briefly* below:

Pat's reasoning is incorrect. The operating system can context switch threads at any time, so the threads may observe shared data structures at arbitrary instants, making it essential to synchronize access to them.

3. [3 points] We heard a fair bit in class about coarse-grained vs fine-grained locking. Tinus Lorvalds has written an operating system called Tinux, and Tinus is also taking CS 372H. Inspired by MCS locks, Tinus decides to replace the implementation of all of Tinux's spinlocks with MCS locks, leaving all code otherwise unmodified. (That is, Tinus replaces the implementation of `acquire()` and `release()` but does not otherwise change any of his code.) By doing so, Tinus does what to the granularity of locking in the kernel?

Select one:

A With this change, Tinus makes Tinux's locking more coarse-grained.

B With this change, Tinus has no effect on the granularity of locking in Tinux.

C With this change, Tinus makes Tinux's locking more fine-grained.

Answer: B. Replacing the type of lock does not change the granularity at which locking occurs. Granularity is determined by *where* the lock acquisition happens and how many locks we have, not by their implementation.

4. [6 points] Consider a machine with 32 MB of RAM running an operating system with virtual memory and swapping. The OS's page replacement policy is: if, on a page fault, a process needs a new physical page in RAM, evict the page that has been in RAM in the *longest*, and write it to the disk if it is dirty. The machine owner notices that for some workloads, the operating system does a lot of disk writes, and the owner is unhappy about that. In response, the owner installs an extra 8 MB of RAM, and re-runs the workload.

Circle True or False for each item below:

True / False There are workloads for which the extra RAM will *decrease* the number of page faults.

True / False There are workloads for which the extra RAM will *have no effect on* the number of page faults.

True / False There are workloads for which the extra RAM will *increase* the number of page faults.

All are true. For the first item: the working set might fit into 40 MB but not 32 MB of RAM. In that case, the extra RAM will decrease the number of page faults. For the second item: the workload might be pathological, say looping through all of its memory. Because this workload exhibits no locality of reference, the FIFO cache will not help: *every* reference could generate a page fault, whether there are 32 MB or 40 MB of RAM (in fact there are pathological workloads for which the gains of *any* cache, not just a FIFO-managed one, will be miniscule). For the third item, consider *Belady's anomaly*, which we discussed in class and which was covered again in the homework: for some access patterns and a FIFO replacement policy, increasing the size of a cache may *increase* the number of cache misses.

5. [2 points] Which of the following scheduling disciplines can lead to starving of processes? Assume a system in which processes do not synchronize with each other or otherwise coordinate.

Circle all that apply:

A Strict priority

B Round-robin

C Lottery scheduling (assume each process has at least one ticket)

Of these choices, only **A** could lead to starvation. Round-robin eventually runs every process. In lottery scheduling, even one ticket means that, by probability, eventually every process runs.

6. [4 points] State one actual coding practice that developers follow to avoid deadlocks in their code. State your answer briefly (there is far more white space below than you need). If you cannot think of one, write, "hope and pray" to get 1 point, and do not write anything else below. A blank answer will receive 0 points.

The most common thing is to try to acquire and release locks (or more generally resources) in a partial order (which negates one of the four required conditions for deadlock, namely the circular wait condition). Developers also instrument their binaries (dynamic checking), and they may use static checking tools too. The banker's algorithm is nice in theory, but it is not used in practice.

7. [3 points] Consider a JOS environment: `struct Env e;` which of the following installs `e`'s address space as the current address space on the x86 CPU that runs JOS?

Select one, and note that the definitions below the choices may be helpful:

A `lcr3(e.env_cr3);`

B `lcr3(e.env_pgdir);`

C `env_pop_tf(&e.env_tf);`

D None of the above

Choice A is correct.

Relevant definitions:

`lcr3(arg):` loads register `%cr3` with `arg`.

`env_pop_tf(tf):` restores saved register values and then calls `iret`.

`struct Env` is defined as follows:

```
struct Env {
        struct Trapframe env_tf;      // Saved registers
        LIST_ENTRY(Env) env_link;     // Free list link pointers
        envid_t env_id;               // Unique environment identifier
        envid_t env_parent_id;        // env_id of this env's parent
        unsigned env_status;          // Status of the environment
        uint32_t env_runs;            // Number of times environment has run

        // Address space
        pde_t *env_pgdir;             // Kernel virtual address of page dir
        physaddr_t env_cr3;           // Physical address of page dir

};
```

II The readings (14 points total)

8. [4 points] Answer ONE of the following two questions, **A** or **B**, neither of which we discussed in class (these are testing the textbook reading). The answers ought to be brief. If you don't know the answer, just leave it blank and keep going.

A What is the classic *dining philosophers problem* described in the book? How many philosophers? What is the constraint? (You may answer this question by drawing a picture and then just stating a few words.)

B In the context of deadlock prevention, what is the concept of a *safe state*, as explicated in the textbook?

Your answer here:

A For the dining philosophers, see the text.

B A safe state is a configuration of currently acquired resources by the set of threads/processes/entities in a system such that there *exists* a scheduling order that the OS can impose that is *guaranteed* to avoid deadlock even if every thread/process/entity requests its maximum allocation. The idea is that if the OS keeps the system in a safe state *and* if it schedules properly, then the behavior of the threads (what resources they request) still cannot cause deadlock. Answers that appeared to capture most of this concept received full credit.

9. [4 points] Liedtke writes, "IPC performance is the Master". He follows this creed seriously, going to great lengths to serve the goal of IPC performance in L3. Which of the following techniques does he use?

Circle all that apply:

A Stack doubling

B Changing the system call interface

C Reducing the number of cycles needed to handle IPC below the theoretical minimum of 172

D Using low-level bit and byte tricks, such as strategic placing of needed data within 32-bit words and packing along cache lines.

E When a thread P sends an IPC to a thread Q, mapping the MSRs (model-specific registers) of P read-only in Q's address space.

Only B and D. A is a made-up term. C is impossible (Liedtke's minimum of 172 cycles is the minimum possible; his optimizations are to help him get from whatever he was at previously *toward* 172). As for E, which many students selected, it was a red herring. MSRs have nothing to do with the paper (and are not per-thread). Note that what Liedtke does do (which we discussed in class) is to map *Q*'s memory kernel-only (but writable!) in *P*'s space.

10. [4 points] There were two software errors described in the Therac-25 article. We discussed the first one in class (three threads, one of them ignored signals for eight seconds, etc.). This question is about the second software error. Recall from the paper that this error only happened when the operator positioned the turntable for field light operation, then attended to the patient, then pressed "set" (which was supposed to save the parameters and move the turntable out of field light position), and then pressed "beam on". Further recall that these circumstances did not always cause the error. The pseudocode below simplifies the Therac's control flow (as reported in the paper), but it leaves intact the identical bug reported in the paper:

```
uint8_t class3 = 0; /* 8-bit quantity */

while (1) {

       if (in field light position) {
             increment class3;
       }

       check whether operator pressed "set";

       if (operator pressed set) {
            if (class3 != 0) {
                 move turntable out of field light position;
            }
            break;
       }
}
```

Explain the problem in the above pseudocode *briefly*. You do not have to explain the effect on the Therac-25's mechanics or on patients, or explain the error in detail, and you should ignore larger software engineering issues like that the code was poorly structured to begin with. Just explain the error briefly in the space below:

The fundamental problem is that, because `class3` is only 8 bits, the incrementing can cause it to roll over to 0 periodically. When it does so, and if the operator presses "set" at the exact wrong moment, the inner check in the second group of `if` statements evaluates to false, so the turntable is not moved out of field light mode. This is an error. The result of the error is that "beam on" delivers a beam with no attenuation, inflicting a massive radiation burn on a human in the path of the beam.

11. [2 points] Who wrote, "That's one hell of a good excuse for some of the brain-damages of minix. I can only hope (and assume) that Amoeba doesn't suck like minix does."?

Linus Torvalds wrote this, in the flamefest between him and Andy Tanenbaum. There were some context clues for this one, such as the fact that the question was in the readings section of the exam, that the line sounded like it was from an email war, and that we had as assigned reading an email war. So perhaps you could have narrowed it to Torvalds or Tanenbaum just by knowing what reading was assigned. At that point, to get the question, you would have had to have remembered the debates,

specifically that they kept referencing Tanenbaum's having written Minix. (In fact, the debates started with Tanenbaum's disparaging Linux, and the debates took place *on the minix newsgroup*.) From there, you perhaps could have guessed that the insulting of Minix in the quotation was Torvalds on Tanenbaum, rather than the other way around.

III Monitors and condition variables (17 points total)

12. [5 points] This question concerns the bounded buffer producer/consumer code, using mutexes and condition variables, that we saw in lecture. This code is reproduced on the next page, but it's not important for you to read it carefully; the question references the important line numbers. Recall that `cond_wait()` (see line 12) takes two arguments: a mutex and a condition variable. Further recall that this function releases the mutex and sleeps *atomically*. Then, when the thread is woken by `cond_signal()` (line 32), it acquires the mutex. Your task here is to show why the two steps that are atomic must indeed be atomic. You will answer this specific question: **if lines 11–12 were instead the following, then what could go wrong?**

```
while (count == BUFFER_SIZE) {
    release(&mutex); // no guarantee of atomicity between this and next line
    wait_for_signal(&nonfull);
    acquire(&mutex);
}
```

Answer this question by (a) giving an interleaving in which something undesirable happens and (b) stating what the undesirable thing is. Include line numbers where possible. We have started the answer for you:

```
producer executes "release(&mutex)"
....
```

(a)

producer executes "release(&mutex)"

consumer executes "acquire(&mutex)" (line 25)

consumer continues to "cond_signal(&nonfull)" (line 32)

producer executes "cond_wait(&nonfull)"

which results in ...

(b) ... a lost signal

Some students continued the interleaving until the system reached deadlock; this was also correct and very nice work. Students lost points for interleavings that couldn't exist (e.g., that didn't obey program order) or for incorrect statements about the behavior of the system.

```
Mutex mutex;
Cond nonempty;
Cond nonfull;

void producer (void *ignored) {
    for (;;) {
        /* next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();

        acquire(&mutex);
        while (count == BUFFER_SIZE)
           cond_wait(&nonfull, &mutex);

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal(&nonempty, &mutex);
        release(&mutex);
    }
}

void consumer (void *ignored) {
    for (;;) {

        acquire(&mutex);
        while (count == 0)
           cond_wait(&nonempty, &mutex);

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal(&nonfull, &mutex);
        release(&mutex);

        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```

13. [6 points] Say we have a monitor `M` that houses a mutex, called `mutex` (naturally), and three condition variables, `cv1`, `cv2`, and `cv3`. That is, the monitor's definition starts like this:

```
class M {
    Mutex mutex;
    Cond  cv1;
    Cond  cv2;
    Cond  cv3;
    ....
};
```

Assume that the monitor is implemented correctly ("correct" here means that it is built according to our coding standards, and it is both safe and live) and that it never calls `cond_broadcast()`. Pat Q. Hacker hates wasting memory, so suggests making the following replacements inside the monitor code:

- Replace all three condition variables with a single condition variable, `single_cv`.
- Replace any call to `cond_wait(&cv1, &mutex)`, `cond_wait(&cv2, &mutex)`, or `cond_wait(&cv3, &mutex)` with a call to `cond_wait(&single_cv, &mutex)`.
- Replace any call to `cond_signal(&cv1, &mutex)`, `cond_signal(&cv2, &mutex)`, or `cond_signal(&cv3, &mutex)` with a call to `cond_signal(&single_cv, &mutex)`.

Can the three bulleted changes, taken together, detract from the monitor's *safety*? Explain your answer *briefly*. (Recall that a safe program is, for our purposes, one that will never do anything bad; it has no race conditions, for example.)

By assumption, the monitor is implemented correctly. Recall that in a correct monitor, safety is enforced by two things: the condition in the while loop (in the while() { wait() } construction), and the mutex itself. (Some students said that safety comes from the mutex, but this is only partially true: one can come up with examples, such as a bounded buffer without the while loop, in which there is mutual exclusion but no safety.) Thus, changing what CV is waited on and when signals happen cannot affect safety; the only thing that could affect safety would be the placement of the while() loops, and the conditions inside the loop (not to be confused with condition variables) that indicate when it is safe to progress beyond the loop.

Can the three bulleted changes, taken together, detract from the monitor's *liveness*? Explain your answer *briefly*. (Recall that a live program is one that is guaranteed to make progress in all cases.)

The replacements could have a severe impact on liveness. If a signal were intended for thread A and were directed accordingly by happening only along one CV, and if thread B received it (as it could, after the replacements), then thread A might never hear it, which could cause the program to, say, sleep forever.

14. [6 points] Now assume that the third bullet is instead the following:

- Replace any call to `cond_signal(&cv1, &mutex)`, `cond_signal(&cv2, &mutex)`, or `cond_signal(&cv3, &mutex)` with a call to `cond_broadcast(&single_cv, &mutex)`. *Read that carefully.*

Can the three bulleted changes, relative to the original monitor, detract from the monitor's *safety*? Explain your answer *briefly*. (Recall that a safe program is, for our purposes, one that will never do anything bad; it has no race conditions, for example.)

Can the three bulleted changes, relative to the original monitor, detract from the monitor's *liveness*? Explain your answer *briefly*. (Recall that a live program is one that is guaranteed to make progress in all cases.)

The point of questions 13 and 14 together is to demonstrate that the purpose of having multiple condition variables within a monitor, versus having only one (or none!) is really a *performance* optimization. It's important to realize where the *safety* comes from: the enforcing of invariants via the content and placement of the while loops. Liveness comes from waking up every thread *at least as often as it needs to be woken*. We guarantee to test these same concepts on the final exam. Please do check out Mike Dahlin's coding standards document (linked from lab T or here), and see below for more detail on correct and incorrect answers.

For the first part, *it's never unsafe to spuriously wake up a thread*. Having one CV and broadcast()ing to all threads is an extreme form of delivering lots of spurious wakeups. But, as mentioned in the previous problem, *safety* (as opposed to liveness) does not stem from when and how threads are signaled by CVs but rather by the checks in the while loops. Hence, the change would not detract from safety.

Moreover, this approach would not fundamentally imperil the program's liveness since every thread would be woken, including the "needed" one. (The approach may be a bit of a performance drag, given the spurious wakeups.)

Some students argued that "too many" threads would be awake, possibly deadlocking the system. We awarded full credit for this answer, but this answer depends on something that isn't the case in most threading implementations. In brief, *when coding with CVs, you must assume that wait() can return for any reason, even if it wasn't signaled*. Under that assumption, if the monitor was live originally, it's live if broadcast() delivers lots of wakeups. The argument is simply that the monitor needs to be live no matter *what* wait() does, so broadcast() can't make a live monitor "unlive" (i.e., there's no concept of broadcast causing threads to jump the gun—the monitor needs to be prepared for that case anyway). Because we didn't cover this point in lecture, we assumed, when grading, that wait()s do not wake up without having been signaled. Given this (false) assumption, it is possible to construct a contrived example in which a live monitor becomes unlive as a result of the third bulleted change.

Thus, students who articulated the "too many threads could yield deadlock" argument got full credit, if they constructed a halfway plausible scenario.

Some students mentioned fairness issues (here and in the previous problem). This was creative, outside-the-box thinking but ultimately not quite right. The way that mutexes and condition variables work is this: every mutex has a queue of waiting threads in front of it (see page 5 of the lecture 9 handout); every one of these threads is trying to move past acquire(). Every CV likewise has a queue; every thread in that queue is in the middle of wait(). When a broadcast() occurs on a CV, the kernel (or whatever logic is implementing the synchronization) moves every thread from the CV's queue to the mutex's queue. It's conceivable that there could be fairness issues stemming from which thread on the mutex's waiting queue is chosen (say if the scheduler always picks a high priority thread from the mutex's waiting queue—which is not a good thing to do, but let's assume it). However, even in that case, it's hard to see how this would *permanently* starve a thread on the mutex's waiting queue—when the original monitor was live. However, because the answer was "close" (for some metric), we gave partial credit for this answer.

Answers that tended to do worse, pointswise, were those that displayed a misunderstanding about how CVs and monitors are to be used.

Again, as mentioned above, we guarantee to test the use of monitors and CVs on the final.

IV Spinlocks, plus one last question (11 points total)

15. [10 points] In this problem, you will implement a multiple-reader, single-writer lock as a *spinlock*. Here is the description:

```
struct sharedlock {
    int value; // when the lock is created, value is initialized to 0
};
```

- It allows multiple readers OR one single writer, and there are four functions:
  `reader_acquire(struct sharedlock*)`,
  `reader_release(struct sharedlock*)`,
  `writer_acquire(struct sharedlock*)`,
  `writer_release(struct sharedlock*)`.
  We have given you the first of these, and your task is to write the last three of these. ***Each of these three functions only needs to be a single line of code.***
- When the lock is unlocked (no readers or writers holding the lock), its `value` is 0.
- When there are one or more readers holding the lock (that is, multiple threads have completed `reader_acquire()` but have not called `reader_release()`), the lock's `value` equals the number of readers
- When the lock is held by a writer (i.e., a thread has made it past `writer_acquire()` but has not called `writer_release()`), its `value` is -1.
- We are unconcerned here with fairness, efficiency, or starvation; just write something that is safe and that eventually allows a waiting thread, reader or writer, to make progress, even though a waiting writer may have to wait until there are no readers.
- Assume that the lock is never acquired by an interrupt handler, so you don't need to worry about enabling and disabling interrupts. You may also assume that the hardware provides sequential consistency.

You will likely need to call two atomic primitives. We describe them below. (We also include their pseudocode and inline assembly implementations in the appendix at the end of the exam. However, you do not need to check this appendix material to do the problem.)

- `int cmpxchg_val(int* addr, int oldval, int newval)`, which we saw in lecture. It is an atomic operation. It compares `oldval` to `*addr`, and if the two are equal, it sets `*addr = newval`. It returns the old contents of `*addr`.
- `void atomic_decrement(int* arg)`, which atomically performs `*arg = *arg - 1`.

We repeat, each of the three remaining functions requires only a line of code.

A final note: if you are stuck on this problem, recall that, in lecture, we saw a multiple-reader, single-writer lock implemented as a *monitor*. That code may be useful inspiration, and we include it in the appendix section at the end of the exam. However, you do not need to consult it to do the problem; it's there only because you may find it useful.

```
// we are giving you the code for the first of the four functions:
void reader_acquire(struct sharedlock* lock) {
     int curr_val;
     while (1) {

          // spin while a writer owns the lock
          while ((curr_val = lock->value) == -1) {}

          assert(curr_val >= 0);

          // try to atomically increment the count, based on our best
          // guess of how many readers there had been. if we were
          // wrong, keep looping. if we got it right, then we
          // succeeded in incrementing the count atomically, and we
          // can proceed.
          if (cmpxchg_val(&lock->value, curr_val, curr_val + 1) == curr_val)
              break;
     }
     // lock->value now contains curr_val + 1
}

void reader_release(struct sharedlock* lock) {
     // your code here: only needs to be one line




}



void writer_acquire(struct sharedlock* lock) {
     // your code here: only needs to be one line




}


void writer_release(struct sharedlock* lock) {
     // your code here: only needs to be one line




}
```

Solutions:

```
void reader_release(sharedlock* lock)
{
     atomic_decrement(&lock->value);
}

void writer_acquire(sharedlock* lock)
{
    /* most common error here was not spinning, i.e., not including
    the while() around the cmpxchg. note that spinning is what keeps
    the writer from entering the critical section before it's
    supposed to */
    while (cmpxchg(&lock->value, 0, -1) != 0) {}
}

void writer_release(sharedlock* lock)
{
    xchg_val(&lock->value, 0); /* xchg_val is from class notes */
    /*
     * other answers that work are:
     *  (1)  cmpxchg(&lock->value, -1, 0);
     *  (2)  lock->value = 0;
     * the latter works because the question said to assume
       sequential consistency. Without sequential consistency,
       the latter option might not work because it might
       not guarantee that all of the instructions before or after it
       would appear in program order to the other CPUs in the machine.
     */
     /*
      * some students wrote:
      *    while (cmpxchg(&lock->value, -1, 0) != -1) ;
      * which will have the correct effect, but note that the while
      * isn't necessary (and may indicate a slightly shaky understanding
      * of how spinlocks work). In fact, the above only works
      * because the loop condition is executed exactly once.
      */
}
```

There is one more page and one further question for you to complete.

16. [1 points] Last question: What is the recommended completion date for lab 4A? For reference, today is Thursday, March 11, 2010.

The course Web page recommended completing lab 4A by Friday, March 12, 2010.

End of Midterm

Enjoy Spring Break!!

A Implementation of atomic primitives

cmpxchg_val()

```
/* pseudocode */
int cmpxchg_val(int* addr, int oldval, int newval) {
      LOCK: // remember, this is pseudocode
      int was = *addr;
      if (*addr == oldval)
           *addr = newval;
      return was;
}

/* inline assembly */
int cmpxchg_val(int* addr, int oldval, int newval) {
    int was;
    asm volatile("lock cmpxchg %3, %0"
         : "+m" (*addr), "=a" (was)
         : "a" (oldval), "r" (newval)
         : "cc");
    return was;
}
```

atomic_decrement()

```
/* pseudocode */
void atomic_decrement(int* arg) {
    LOCK: // remember, this is pseudocode
    *arg = *arg - 1;
}

/* inline assembly */
void atomic_decrement(int* arg) {
    asm volatile("lock decl %0" : "+m" (*arg) : "m" (arg));
}
```

B Shared lock code from lecture

```
struct sharedlock {
  int i;
  Mutex mutex;
  Cond c;
};

void AcquireExclusive (sharedlock *sl) {
  acquire(&sl->mutex);
  while (sl->i) {
    cond_wait(&sl->c, &sl->mutex);
  }
  sl->i = -1;
  release(&sl->mutex);
}

void AcquireShared (sharedlock *sl) {
  acquire(&sl->mutex);
  while (sl->i < 0) {
    cond_wait(&sl->c, &sl->mutex);
  }
  sl->i++;
  release(&sl->mutex);
}

void ReleaseShared (sharedlock *sl) {
  acquire(&sl->mutex);
  if (!--sl->i)
    cond_signal(&sl->c, &sl->mutex);
  release(&sl->mutex);
}

void ReleaseExclusive (sharedlock *sl) {
  acquire(&sl->mutex);
  sl->i = 0;
  cond_broadcast(&sl->c, &sl->mutex);
  release(&sl->mutex);
}
```