Computer Science 15-410: Operating Systems
Mid-Term Exam (A), Fall 2010

1. **Please read the entire exam before starting to write. This should help you avoid getting bogged down on one problem.**
2. Be sure to put your name and Andrew ID below *and also* put your Andrew ID at the top of *each* following page.
3. This is a closed-book in-class exam. You may not use any reference materials during the exam.
4. If you have a clarification question, please write it down on the card we have provided. Please don't ask us questions of the form "If I answered like this, would it be ok?" or "Are you looking for ...?"
5. The weight of each question is indicated on the exam. Weights of question *parts* are estimates which may be revised during the grading process and are for your guidance only.
6. Please be concise in your answers. You will receive partial credit for partially correct answers, but truly extraneous remarks may count against your grade.
7. **Write legibly even if you must slow down to do so!** If you spend some time to *think clearly* about a problem, you will probably have time to write your answer legibly.

Andrew Username	
Full Name	

Question	Max	Points	Grader
1.	**10**		
2.	**10**		
3.	**15**		
4.	**15**		
5.	**15**		
6.	**10**		
	75		

Please note that there are system-call and thread-library "cheat sheets" at the end of the exam.

Andrew ID: _____

After I leave this exam session, I will not discuss the contents of this 15-410 midterm with *anybody*, whether or not in this class, whether or not present in this exam session with me, before 19:00 on Wednesday, October 6th.

Signature: _____ Date _____

Please note that there are system-call and thread-library "cheat sheets" at the end of the exam.

If we cannot read your writing, we will be unable to assign a high score to your work.

Andrew ID: _____

1. [10 points] Short answer.

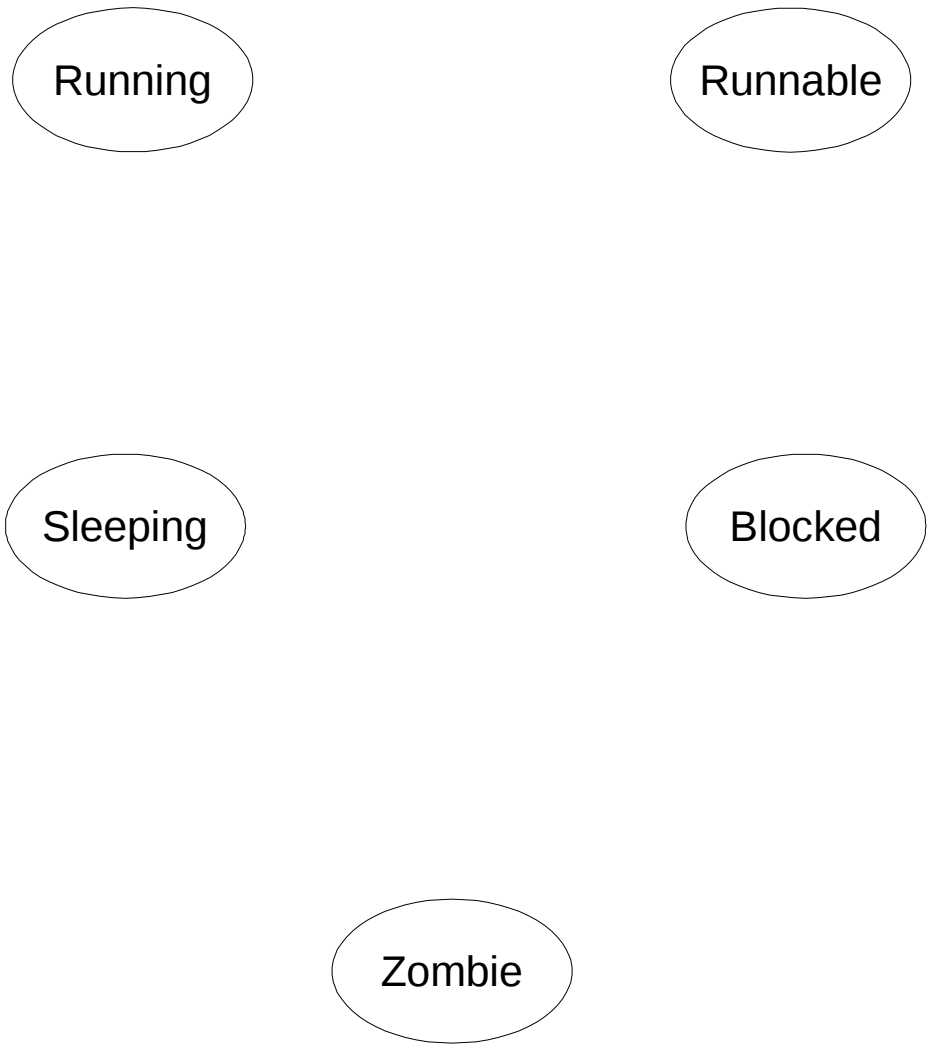
We are expecting each part of this question to be answered by three to six sentences. Your goal is to make it clear to your grader that you understand the concept *as it applies to this course* and can apply it when necessary.

(a) [6 points] List three "kinds" of register. For each kind, name one register of that "kind" and briefly explain what is special about it or its particular purpose.

(b) [4 points] What is a "kernel stack"? When is one used and why are they important?

Andrew ID: _____

2. [10 points] Draw a diagram showing the transitions among the following scheduling states: RUNNING, RUNNABLE, SLEEPING, BLOCKED, ZOMBIE (ZOMBIE may also be known as EXITING). Label the arcs with events which make sense in terms of the 15-410 "Pebbles" run-time environment. If more than one event can cause a particular transition, list two, and try to make the two of them as different from each other as you can. If it seems appropriate to you, you are authorized to draw some "arcs" which have one endpoint instead of two.



3. [15 points] Condition variables.

The "condition variable" is an important concurrency primitive that encapsulates the notion of "stop running, so that other threads may use the processor, until the world changes in a way which probably enables me to continue working." A necessary part of implementing condition variabls is solving the "atomic unlock-and-deschedule" problem: a waiting thread must release a lock and ask the kernel to block it; if another thread is trying to awaken the waiting thread, the "awaken" operation must not be lost just because its execution is interleaved in a troublesome way with the "release, then block" sequence.

The Fall 2010 Pebbles kernel specification provides user code with two system calls which can be combined to solve the atomic-blocking problem, `deschedule()` and `make_runnable()`. Other systems provide other primitives, some of which are quite different. Linux provides "futexes" and signals; Plan 9 provides a primitive called "rendezvous."

> `void *rendezvous(void *tag, void *value)` - Synchronize, and exchange values between, two threads in the same task. Two threads wishing to synchronize invoke `rendezvous` specifying the same *tag* parameter and arbitrary *value* parameters. The first thread to arrive at the rendezvous will suspend execution until the second thread arrives. When another thread invokes `rendezvous()` with the same *tag* parameter, each one receives the `value` parameter specified by the other one. After the exchange, both threads are runnable. The return value of the system call is the *value* parameter specified by the other thread. The kernel places no interpretation on the values of the *tag* and *value* parameters except when it performs equality testing on the *tag* parameters.

In this problem you will implement condition variables using mutexes and `rendezvous()`. You may not use other atomic or thread-synchronization synchronization operations, such as, but not limited to: semaphores, reader/writer locks, `deschedule()`/`make_runnable()`, or any atomic instructions (`XCHG`, `LL/SC`).

For the purposes of the exam, you may assume that library routines and system calls don't "fail" (unless you indicate in your comments that you have arranged, and are expecting, a particular failure).

(Continued on next page)

Please declare a `struct cond` and implement `cond_init()`, `cond_wait()`, and `cond_signal()` (you do not need to implement either `cond_broadcast()` or `cond_destroy()`). If you wish, you may also declare an auxiliary structure, `struct aux`, *but this is strictly optional.*

It is strongly recommended that you rough out an implementation on the scrap paper provided at the end of the exam, or on the back of some other page, before you write anything here. If we cannot understand the solution you provide on this page, your grade will suffer!

```
typedef struct cond {









} cond_t;


typedef struct aux {




} aux_t;
```

...space for cvar implementation...

You may use this page as extra space for your cvar solution if you wish.

4. [15 points] List locking.

Your friend Barry Hovik is writing some concurrent code using a shared linked list, and comes to you wondering about concurrency primitives. His code currently looks like this:

```
void list_insert_sorted(list_t *list, list_node_t *new)
{
    list_node_t **p;

    /* STEP 0: Prevent concurrent access. */
    rwlock_lock(&list->lock, RWLOCK_WRITE);

    /* STEP 1: Iterate down the list. */
    for (p = &list->head; *p != NULL; p = &(*p)->next) {
        if (list->compare((*p)->data, new->data) >= 0) {
           /* Entry should be inserted after the current one. */
           break;
        }
    }

    /* STEP 2: Insert node. */
    new->next = *p;
    *p = new;

    rwlock_unlock(&list->lock);
}
```

You should imagine that other list operations, such as `list_delete()`, `list_count_nodes()`, etc., exist and are coded in sensible (correct) fashions. Naturally, operations that examine the list but do not modify it lock the list in `RWLOCK_READ` mode.

Barry is worried that `list_insert_sorted()` will perform poorly because walking down the list takes $O(n)$ time and his code will hold a `RWLOCK_WRITE` (exclusive) lock during the entire operation. He explains that he briefly considered rewriting `list_insert_sorted()` as follows.

```
void list_insert_sorted(list_t *list, list_node_t *new)
{
    list_node_t **p;

    /* STEP 0: "Declare" that list shouldn't change while we walk it. */
    rwlock_lock(&list->lock, RWLOCK_READ);

    /* STEP 1: Iterate down the list. */
    for (p = &list->head; *p != NULL; p = &(*p)->next) {
        int cmp;
        if ((cmp = list->compare((*p)->data, new->data)) == 0) {
           /* We already have one of these */
           rwlock_unlock(&list->lock);
           free(new);
           return;
        } else if (cmp > 0) {
           /* Entry should be inserted after the current one. */
           break;
        }
    }

    /* STEP 2: Promote READ lock to WRITE lock before modification. */
    rwlock_unlock(&list->lock);
    rwlock_lock(&list->lock, RWLOCK_WRITE);

    /* STEP 3: Insert node. */
    new->next = *p;
    *p = new;

    rwlock_unlock(&list->lock);
}
```

(a) [5 points] Of course, you and Barry both know this proposed revision of `list_insert_sorted()` is totally unusable. Briefly but compellingly explain why. Your answer may consist of a few sentences or a short paragraph, and *may* include a brief execution trace, though one is *not* required. For full credit, explain not just what is "theoretically wrong" with this approach but also a particular bad outcome which could be observed.

Now that you and Barry are thinking along the same lines, he unveils a novel proposal. He has invented a new operation for readers/writers locks to support, called `rwlock_promote()`, defined as follows.

> `void rwlock_promote(rwlock_t *rwlock)` - If the invoking thread does not already hold a `RWLOCK_READ` lock on the specified rwlock, this constitutes a fatal error and the program immediately ends. Otherwise, the invoking thread is blocked until the `RWLOCK_READ` lock can be "atomically upgraded" to an `RWLOCK_WRITE` lock. While a thread is blocked in `rwlock_promote()`, no other thread will obtain an `RWLOCK_WRITE` or `RWLOCK_READ` lock on the specified rwlock before the promotion to `RWLOCK_WRITE` completes.

Barry proposes replacing the two lines of bad "STEP 2: Promote..." code above with one line of code, namely the invocation of `rwlock_promote(&list->lock);`.

As Barry explains his proposal, you have a sinking feeling. Though Barry assures you that he has figured out how to rewrite his rwlock implementation to provide an `rwlock_promote()` that fully complies with the specification above, you regretfully explain to him that this approach has a fatal flaw.

(b) [10 points] Summarize the flaw in a sentence or two and also provide an execution trace which *clearly and compellingly* illustrates how this flaw will appear. Assume his `rwlock_promote()` (and his other readers/writers lock code) works exactly according to specification, and also assume, for exam purposes, that necessary library and system call invocations aren't forced to fail by unfortunate circumstances.

You may use this page as extra space for the rwlock question if you wish.

5. [15 points] Consider the following critical-section protocol:

```
        volatile int stall[2] = { 1, 1 };

1.      do {
2.          ...remainder section...
3.          do {
4.              stall[i] = 1 - stall[j]; // I stall iff other thread isn't.
5.          } while (!stall[j]);
6.          ...critical section...
7.          stall[i] = 1;
8.      } while (1);
```

This protocol is presented in "standard form," i.e.,

1. When process 0 is running this code, $i == 0$ and $j == 1$; when process 1 is running this code, $i == 1$ and $j == 0$, so i means "me" and j means "the other process."
2. Lines 3–5 are can be thought of roughly as "acquiring a lock" and line 7 can be thought of roughly as "releasing the lock."

(a) [10 points] There is a problem with this protocol. That is, it does not ensure that all three critical-section algorithm requirements are always met. Identify a requirement which is not met and lay out a scenario which demonstrates your claim. Use the format presented in class, i.e.,

P0	P1
stall[0] = 99;	
	stall[1] = 103;

You may introduce temporary variables or other obvious notation as necessary to improve the clarity of your answer. *Be sure that the execution trace you provide us with is easy to read and conclusively demonstrates the claim you are making.* It is possible to answer this question with a brief, clear trace, so you should do what is necessary to ensure that you do.

You may use this page as extra space for the critical-section protocol question if you wish.

(b) [5 points] Show, *with a clear and convincing trace*, that a *different* critical-section algorithm requirement also does not hold.

You may use this page as extra space for the critical-section protocol question if you wish.

6. [10 points] `Nuts & Bolts`.

Below are some register dumps produced by the "pathos" P2 reference kernel when it decided to kill a thread. Your job is to carefully consider each register dump and:

1. Determine which register's "wrong value" caused the thread to run an instruction which resulted in a fatal exception.
2. Briefly state the most plausible way you think that register could have taken on that value (i.e., try to describe a bug which could have this effect).
3. Then write a *small* piece of code which would cause the thread to die in the fashion indicated by the register dump. Most answers will probably be in assembly language, but C is acceptable as well. Your code should assume execution begins in `main()`, which has been passed the typical two parameters in the typical fashion.

Your code doesn't need to reproduce the register dump *exactly*—for example, we won't require you to pad your code so that a faulting instruction has exactly the same address as what appears in the register dump.

Please be sure that your description of the fatality and the code, taken together, clearly support your diagnosis.

(Continued on next page)

(a)
```
Registers:
eax 00000000   ebx 00000000   ecx 00000000   edx 00000004
edi 00000000   esi 00000000   ebp ffffffdd
esp ffffffd5    ss 0000002b   eip ffefffdd    cs 00000023
 ds     002b    es     002b    fs     002b    gs     002b
eflags 00210246
```

(b)
```
Registers:
eax 00000001   ebx 010000af   ecx ffffffd1   edx 00000000
edi 00000000   esi 00000000   ebp ffffffdd
esp 00002fbb    ss 0000002b   eip 0100000c    cs 00000023
 ds     002b    es     002b    fs     002b    gs     002b
eflags 00210282
```

System-Call Cheat-Sheet

```
/* Life cycle */
int fork(void);
int exec(char *execname, char *argvec[]);
void set_status(int status);
void vanish(void) NORETURN;
int wait(int *status_ptr);
void task_vanish(int status) NORETURN;

/* Thread management */
int thread_fork(void); /* Prototype for exam reference, not for C calling!!! */
int gettid(void);
int yield(int pid);
int deschedule(int *flag);
int make_runnable(int pid);
int get_ticks();
int sleep(int ticks); /* 100 ticks/sec */

/* Memory management */
int new_pages(void * addr, int len);
int remove_pages(void * addr);

/* Console I/O */
char getchar(void);
int readline(int size, char *buf);
int print(int size, char *buf);
int set_term_color(int color);
int set_cursor_pos(int row, int col);
int get_cursor_pos(int *row, int *col);

/* Miscellaneous */
void halt();
int ls(int size, char *buf);

/* "Special" */
void misbehave(int mode);
```

If a particular exam question forbids the use of a system call or class of system calls, the presence of a particular call on this list does not mean it is "always ok to use."

Thread-Library Cheat-Sheet

```
int mutex_init( mutex_t *mp );
int mutex_destroy( mutex_t *mp );
int mutex_lock( mutex_t *mp );
int mutex_unlock( mutex_t *mp );

int cond_init( cond_t *cv );
int cond_destroy( cond_t *cv );
int cond_wait( cond_t *cv, mutex_t *mp );
int cond_signal( cond_t *cv );
int cond_broadcast( cond_t *cv );

int thr_init( unsigned int size );
int thr_create( void *(*func)(void *), void *arg );
int thr_join( int tid, void **statusp );
void thr_exit( void *status );
int thr_getid( void );
int thr_yield( int tid );

int sem_init( sem_t *sem, int count );
int sem_wait( sem_t *sem );
int sem_signal( sem_t *sem );
int sem_destroy( sem_t *sem );

int rwlock_init( rwlock_t *rwlock );
int rwlock_lock( rwlock_t *rwlock, int type );
int rwlock_unlock( rwlock_t *rwlock );
int rwlock_destroy( rwlock_t *rwlock );
```

If a particular exam question forbids the use of a library routine or class of library routines, the presence of a particular routine on this list does not mean it is "always ok to use."

Useful-Equation Cheat-Sheet

$$\cos^2\theta + \sin^2\theta = 1$$

$$\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$$

$$\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta$$

$$\sin 2\theta = 2\sin\theta\cos\theta$$

$$\cos 2\theta = \cos^2\theta - \sin^2\theta$$

$$e^{ix} = \cos(x) + i\sin(x)$$

$$\cos(x) = \frac{e^{ix} + e^{-ix}}{2}$$

$$\sin(x) = \frac{e^{ix} - e^{-ix}}{2i}$$

$$\int \ln x\, dx = x\ln x - x + C$$

$$\int_0^\infty \sqrt{x}\, e^{-x}\, dx = \frac{1}{2}\sqrt{\pi}$$

$$\int_0^\infty e^{-ax^2}\, dx = \frac{1}{2}\sqrt{\frac{\pi}{a}}$$

$$\int_0^\infty x^2 e^{-ax^2}\, dx = \frac{1}{4}\sqrt{\frac{\pi}{a^3}} \text{ when } a > 0$$

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\, dt$$

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},\, t) = \hat{H}\Psi(\mathbf{r}, t)$$

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},\, t) = -\frac{\hbar^2}{2m}\nabla^2\Psi(\mathbf{r},\, t) + V(\mathbf{r})\Psi(\mathbf{r},\, t)$$

$$E = hf = \frac{h}{2\pi}(2\pi f) = \hbar\omega$$

$$p = \frac{h}{\lambda} = \frac{h}{2\pi}\frac{2\pi}{\lambda} = \hbar k$$

$$\nabla\cdot\mathbf{E} = \frac{\rho}{\varepsilon_0}$$

$$\nabla\cdot\mathbf{B} = 0$$

$$\nabla\times\mathbf{E} = -\frac{\partial\mathbf{B}}{\partial t}$$

$$\nabla\times\mathbf{B} = \mu_0\mathbf{J} + \mu_0\varepsilon_0\frac{\partial\mathbf{E}}{\partial t}$$