Computer Science 15-410: Operating Systems
Mid-Term Exam (B), Spring 2011

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.	**20**		
3.	**20**		
4.	**10**		
5.	**15**		
	75		

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

I have not received advance information on the content of this 15-410 mid-term exam by discussing it with anybody who took part in the main exam session or via any other avenue.

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.

1. $\boxed{\text{10 points}}$ Short answer.

Give a one-paragraph definition/explanation of each of the following terms *as it applies to this course*. Your goal is to make it clear to your grader that you understand the concept and can apply it when necessary.

(a) $\boxed{\text{5 points}}$ stack frame

(b) $\boxed{\text{5 points}}$ thread-safe

2. $\boxed{\text{20 points}}$ “Socket locks”

You have been asked to write a simplified version of a specialized locking object which might be used in a web browser. As you may know, it is somewhat expensive to establish a connection to a web server, so many browsers “cache” connections so that once a request for some web object (HTML page or image) is complete a later request can re-use an existing connection without the overhead of establishing a new one. (For the purposes of this exam, we will assume that our browser never opens more than one connection at a time to any single server—maybe it’s running on an embedded device and needs to conserve resources).

This locking object is special for two reasons. First, once a thread acquires a lock on a socket, it will perform a potentially long sequence of socket I/O requests, each one of which may require many milliseconds, before releasing the lock (and thus the socket). Your design should be appropriate for this usage pattern. Second, it is a sad fact of life that networks fail and browsers become disconnected from servers. When this happens, it would be silly for many threads in turn to “acquire” a broken connection socket and fruitlessly issue system calls against it. Therefore, when a browser thread determines that a server connection has failed, it invokes a special operation, called “`broken()`,” on the socket lock, before unlocking it. This allows the socket-lock code to inform all relevant threads to give up and invoke higher-level policy code to figure out what to do next. For exam purposes we will assume that `slock_init()`, `slock_broken()`, `slock_unlock()`, and `slock_destroy()` cannot fail, but `slock_lock()` is explicitly allowed (i.e., expected) to return -1 to indicate that the underlying socket has *not* been locked because it was declared to be broken.

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 start to write your solution on the next page. If we cannot understand the solution you provide, your grade will suffer!

The remainder of this page is intentionally blank.

Please declare a `struct slock` and implement `slock_init()`, `slock_lock()`, `slock_broken()`, and `slock_unlock()` (you do not need to implement `slock_destroy()`). You may use standard thread-library synchronization, such as mutexes, condition variables, semaphores, and/or reader/writer locks. You may use `deschedule()`/`make_runnable()` *if you must—it's not recommended*, but may *not* use atomic instructions (`XCHG`, `LL/SC`, etc.). For the purposes of the exam you should assume an error-free environment (invocations of `slock` functions are always legal; memory allocation will always succeed; thread-library primitives will not detect internal inconsistencies or otherwise "fail," etc.). If you wish, you may assume that the mutexes provided by the underlying thread library provide bounded waiting; you may also assume, if you wish, that the underlying condition variables are "as FIFO as possible." You may wish to refer to the "cheat sheets" at the end of the exam.

```
typedef struct slock {











} slock_t;
```

...space for slock implementation...

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

3. $\boxed{\text{20 points}}$ Consider the following proposed implementation of reader/writer locks.

```
typedef struct rwlock {
    int read_count;           /* #readers holding lock (or waiting on a writer) */
    mutex_t read_count_lock;  /* protects read_count */
    int mode;                 /* mode that lock is in: RWLOCK_READ or RWLOCK_WRITE */
    mutex_t write_lock;       /* taken by writer XOR first reader; protects mode */
} rwlock_t;

void rwlock_init(rwlock_t *rwlock) {
    mutex_init(&rwlock->read_count_lock);  mutex_init(&rwlock->write_lock);
    rwlock->read_count = 0;  rwlock->mode = RWLOCK_READ;
}

int rwlock_lock(rwlock_t *rwlock, int mode) {
    switch (mode) {
        case RWLOCK_READ:
            mutex_lock(&rwlock->read_count_lock);
            if (rwlock->read_count == 0) { /* Are we the first reader? */
                mutex_lock(&rwlock->write_lock);
                rwlock->mode = RWLOCK_READ;
            }
            rwlock->read_count++;
            mutex_unlock(&rwlock->read_count_lock);
            return 0;
        case RWLOCK_WRITE:
            mutex_lock(&rwlock->write_lock);
            rwlock->mode = RWLOCK_WRITE;
            return 0;
    }
}

int rwlock_unlock(rwlock_t *rwlock) { /* write_lock is always held on entry */
    switch (rwlock->mode) {
        case RWLOCK_READ:
            mutex_lock(&rwlock->read_count_lock);
            rwlock->read_count--;
            if (rwlock->read_count == 0) { /* Were we the last reader? */
                mutex_unlock(&rwlock->write_lock);
            }
            mutex_unlock(&rwlock->read_count_lock);
            return 0;
        case RWLOCK_WRITE:
            mutex_unlock(&rwlock->write_lock);
            return 0;
    }
}
```

When answering the questions below, you may assume:

1. Mutexes ensure bounded waiting; cvars are “as FIFO as possible.”
2. Due to “exam mode,” nothing fails.
3. The code above invokes mutexes only in legal ways (i.e., does not violate any contract in the mutex spec).

   (a) $\boxed{\text{5 points}}$ Does this rwlock implementation starve readers, writers, both, or neither? Explain your answer.

As you know, one approach to the deadlock problem is to impose a total ordering on resources such as locks. When analyzing unknown code, it can be useful to compute a "lock dependency graph" in which a directed edge is drawn from node A to node B if resource B is acquired by a thread at a time when that thread already owns resource A (we say that the acquisition of B "depends on" the acquisition of A having previously happened). If the lock dependency graph of a body of code is a directed acyclic graph (DAG), this constitutes a *partial* ordering on the resources, which can easily be "flattened" to form some total ordering. However, if a lock dependency graph contains a cycle, there cannot be a total ordering of locks. Note that a lock dependency graph reflects static properties of the code, and is different from the process/resource graph notation used in class, which shows the state of a system at some point in time.

(b) $\boxed{\text{5 points}}$ Draw the lock dependency graph for the code shown above. You *must* "comment" each edge by detailing the condition or situation in which the dependency occurs.

(c) $\boxed{\text{10 points}}$ Can the implementation shown above deadlock? If so, provide an execution sequence using the tabular form shown below. If a deadlock is not possible, provide a clear and concise argument that it cannot happen; your reasoning should be convincing enough to be included with the code as documentation.

Trace format:

Thread 0	Thread 1
rwlock(WRITE)	
...	rwlock(READ)
	...
	return;
return;	

You may introduce temporary variables or other obvious notation as necessary to improve the clarity of your answer. *Be sure that any execution trace or argument you provide us with is easy to read and conclusively demonstrates the claim you are making.*

You may use this page as extra space for the reader/writer locks question if you wish.

4. $\boxed{\text{10 points}}$ Process model.

Consider the following Pebbles system calls (listed in alphabetical order):

1. `deschedule()`
2. `get_ticks()`
3. `make_runnable()`
4. `sleep()`
5. `yield()`

Assign each system call on the list above to one of three categories: "likely to block the invoking thread," "may or may not block the invoking thread," or "unlikely to block the invoking thread." Briefly (one to four sentences) justify your assignment of each system call to the category you selected. Note that "block the invoking thread" means "changes the thread from 'running' to 'blocked."' It is up to you to decide whether one (or more!) of the three categories is empty.

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

5. $\boxed{\text{15 points}}$ `Nuts & Bolts`.

In Project 2, we asked you to implement condition variables in such a way that they don't "fail" if `malloc()` fails. In this question, we'll explore an alternative way of obtaining memory for short periods of time, namely the `alloca()` "function". Below is an official definition from a man page.

```
ALLOCA(3)                BSD Library Functions Manual                ALLOCA(3)

NAME
     alloca -- memory allocator

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <alloca.h>
     or
     #include <stdlib.h>

     void *
     alloca(size_t size);

DESCRIPTION
     The alloca() function allocates size bytes of space in the stack frame
     of the caller.  This temporary space is automatically freed on return.

RETURN VALUES
     The alloca() function returns a pointer to the beginning of the
     allocated space.  If the allocation failed, a NULL pointer is returned.

SEE ALSO
     brk(2), calloc(3), getpagesize(3), malloc(3), realloc(3)

HISTORY
     The alloca() function appeared in Version 32V AT&T UNIX.

BUGS
     The alloca() function is machine and compiler dependent; its use is
     discouraged.

BSD                             June 4, 1993                             BSD
```

In short, `alloca()` provides a C-language interface to allocating a variable amount of memory on the stack. In some senses, it is much like C99's variable array allocation support—these two code sequences accomplish the same thing:

```
void fun1(void)
{
  int n = rand() % 15410;
  int foo[n];

  foo[0] = 15412;
  printf("%d\n", foo[0]); /* foo[] is not disturbed by functions fun1() calls */
  return; /* foo[] is automatically deallocated when fun1() returns */
}

void fun2(void)
{
  int n = rand() % 15410;
  int *foo = alloca(n * sizeof (int));

  foo[0] = 15412;
  printf("%d\n", foo[0]); /* foo[] is not disturbed by functions fun2() calls */
  return; /* foo[] is automatically deallocated when fun2() returns */
}
```

It turns out that compiling code that uses `alloca()` is not straightforward. To make this clear, we will ask you to hand-compile into assembly language a short C function that would be a good candidate for using `alloca()`.

The following function receives a "length/value"-coded message from a network connection and appends the message onto the end of a log. Because each message can have a different size, the function first reads the length from the network connection, then allocates a buffer, then uses that buffer to receive and store the message. Note that for exam purposes critically-important input validation and error checking have been omitted.

```
void fetchlog(int socket) {
    int len;
    unsigned char *value;

    read(socket, &len, sizeof (len));
    value = alloca(len); /* value[] is allocated on stack */
    read(socket, value, len);
    logstore(value, len);
    return; /* value[] is automatically deallocated */
}
```

It is probably beneficial for you to read all parts of the question before answering any part.

It is probably beneficial for you to read all parts of the question before answering any part.

(a) $\boxed{\text{8 points}}$ To start off with, please write the assembly code that a compiler might generate for this function.

Although `alloca()` is in section 3 ("library functions") of the man pages, it's not a symbol in libc:

```
joshua@nyus:~$ nm /usr/lib/libc.a 2> /dev/null | grep 'alloca$'
joshua@nyus:~$
```

(b) $\boxed{\text{5 points}}$ The reason why the C library contains no symbol for `alloca()` is that it *cannot* be implemented as a function callable via the standard calling convention. Explain why it can't (we are expecting two to three sentences).

The `alloca()` specification shown above is copied from Mac OS X. The corresponding Linux man page contains the following statement in the BUGS section: "On many systems alloca() cannot be used inside the list of arguments of a function call." What this means is that

```
    read(fd, alloca(32), 32)
```

won't work right.

(c) $\boxed{\text{2 points}}$ Explain briefly (again, two or three sentences should be plenty; you may show assembly code if you wish) why that kind of invocation won't work or what might go wrong if code like that were compiled by a naïve compiler.

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 */
typedef void (*swexn_handler_t)(void *arg, ureg_t *ureg);
int swexn(void *esp3, swexn_handler_t eip, void *arg, ureg_t *newureg):

/* 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."

Andrew ID: ______________

Useful-Equation Cheat-Sheet

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

If you wish, you may tear this page off and use it for scrap paper. But be sure not to write anything on this page which you want us to grade.