Computer Science 15-410: Operating Systems
Mid-Term Exam (A), Fall 2012

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.	**20**		
5.	**15**		
	70		

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

Andrew ID: _____

1. [10 points] Short answer.

Give a definition of each of the following terms *as it applies to this course*. We are expecting three to five sentences or "bullet points" for each definition. Your goal is to make it clear to your grader that you understand the concept and can apply it when necessary.

(a) [5 points] "Atomic instruction sequence"

(b) [5 points] "kernel mode"

Andrew ID: _____

2. [10 points] Consider the following critical-section protocol:

```
       boolean want[2] = { false, false };
       int turn = 0;

 1.    do {
 2.       // begin entry section
 3.       want[i] = true;
 4.       while (turn != i) {
 5.           if (!want[j]) {
 6.               turn = i;
 7.           }
 8.       }
 9.       // end entry section
10.       ...critical section...
11.       // begin exit section
12.       turn = j;
13.       want[i] = false;
14.       // end exit section
15.       ...remainder section...
16.   } 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 3–8 are can be thought of roughly as "acquiring a lock" and lines 12–13 can be thought of roughly as "releasing the lock."

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
want[0] = 1;	
	want[1] = 1;

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.

Use this page for your solution to the critical-section protocol question.

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

3. [15 points] Graders' Algorithm.

The Operating Systems course staff are grading a mid-term exam. Paul and Eric are working together to grade a particularly tough question.

Initially the grading procedure is quite simple. Eric and Paul share a large "input stack" of exams to be graded. Each grader repeatedly takes an exam off of the input stack, places it in front of himself, grades that exam's answer to their question, and then places the exam on their shared "output stack."

This procedure doesn't work for very long, however. Since the question they are grading is quite tricky, and since various students provide a wide variety of incorrect answers, sometimes one grader needs to ask the other grader to "sign off" on a proposed score. They agree to extend the grading procedure as follows: if, while grading an exam, one grader needs input from the other grader, he will write a note on the exam, place the exam next to the other grader, and then take the next exam from the input pile and continue grading. Neither Paul nor Eric wishes to appear excessively demanding of the other's time, so they agree on this rule: if one grader wishes to place a half-graded exam next to the other grader, but there is already an exam pending review next to the other grader, the grader requesting a new review will not place the new exam until the one previously waiting has been reviewed. Each time a grader pulls an exam from the shared input stack and successfully grades it solo (without consulting the other grader), he will check his personal "pending review" area to see if it contains an exam. In our model, once an exam has been examined by both graders it is definitely done, i.e., there is never a case when one exam needs to be handed back and forth.

Once they settle on this protocol, Michael decides to code it up for simulation purposes. Here is what he comes up with. Note that this simulation doesn't move exam *objects* around; instead, each exam is represented by a small integer.

```
// Do not worry about printf()/printf() races (messages are "small enough").

/* Begin: exam-grading data structures */
typedef struct grader {
    volatile int waiting;
    cond_t removed;
} grader_t;

int total_exams;
mutex_t table_lock;
volatile int table_exams;  // "input stack"
volatile int graded_exams; // "output stack"
grader_t graders[2];
/* End: exam-grading data structures */

// grader-thread "body function"
void *run_grader(void *id);

// examine_exam_number() code is not shown.
// It returns a negative number when a grader can't assign a final score.
extern int examine_exam_number(int en);
```

```
int main(int argc, char *argv[])
{
    total_exams = 54; // should come from command line
    if (thr_init(64*1024)) panic("thr_init");

    mutex_init(&table_lock);
    table_exams = total_exams; graded_exams = 0;
    graders[0].waiting = -1; cond_init(&graders[0].removed);
    graders[1].waiting = -1; cond_init(&graders[1].removed);

    if ((thr_create(run_grader, (void*) 0) < 0) ||
        (thr_create(run_grader, (void*) 1) < 0))
        panic("thr_create");
    thr_exit(0);
    exit(0); // placate compiler
}

// This can take a while, so we enter and leave with the table unlocked.
int handle_exam(int i, int j, int e)
{
    if (examine_exam_number(e) >= 0) {
        // I was able to assign a score myself!
        // Note that this is ALWAYS true if I am the second reader.
        printf("Grader %d graded exam %d.\n", i, e);
        mutex_lock(&table_lock);
        ++graded_exams;
        mutex_unlock(&table_lock);
        return 1;
    } else {
        // I wrote down some comments, now my partner will finish grading.
        printf("Grader %d perplexed by exam %d.\n", i, e);
        mutex_lock(&table_lock);
        while (graders[j].waiting > 0) {
            cond_wait(&graders[j].removed, &table_lock);
        }
        graders[j].waiting = e;
        mutex_unlock(&table_lock);
        printf("Grader %d handed off exam %d.\n", i, e);
        return 0;
    }
}
```

```
void *run_grader(void *vid)
{
    int i = (int) vid;
    int j = 1 - i;

    while (graded_exams < total_exams) {
        int e;
        int succeeded = 1;

        mutex_lock(&table_lock);

        if (table_exams > 0) {
            e = table_exams--;
            mutex_unlock(&table_lock);
            succeeded = handle_exam(i, j, e);
            mutex_lock(&table_lock);
        }

        if (succeeded && graders[i].waiting > 0) {
            e = graders[i].waiting;
            graders[i].waiting = -1;
            mutex_unlock(&table_lock);
            cond_signal(&graders[i].removed);
            handle_exam(i, j, e);
        } else {
            mutex_unlock(&table_lock);
        }
    }

    printf("Grader %d is DONE GRADING!\n", i);
    thr_exit(0);
    exit(0); // placate compiler
}
```

Suggestions for working on this problem:

1. When tracing the execution of the code, we recommend a tabular format very similar to this:

Grader 0	Grader 1
	wait;
lock;	
...	
unlock;	
signal(1)	

2. It is *strongly recommended* that you write down a draft version of any execution trace using 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!

(a) [10 points] Unfortunately, this exam-grading procedure can deadlock. Show a *clear, convincing* execution trace that yields a deadlock (missing, unclear, or unconvincing traces will result in only partial credit).

You may use this page as extra space for the first part of the exam-grading question if you wish.

(b) [5 points] Briefly describe how to fix or restructure the code so that it does not deadlock. It is not necessary for your answer to include code for it to receive full credit if it is clear and convincing (be sure to indicate how your solution addresses one or more deadlock ingredient(s)).

4. [20 points] Channels

In your Project 2 thread library, you implemented various standard synchronization primitives such as mutexes, condition variables, semaphores, reader/writer locks. It is increasingly popular for systems to provide "channels" that combine inter-thread communication with inter-thread synchronization—for example, channels are built into the definitions of two recent programming languages, Go and Rust. In this question you will implement a simplified form of Go channels using the thread-library primitives you implemented during Project 2.

In Go, each channel carries items of a single type, and type safety is enforced by the language. Because C's type system is more streamlined, the channels you implement will carry `void *` values.

When a Go channel is created, the creator specifies a buffer capacity which in turn specifies one of two synchronization policies. If the buffer capacity is zero, the channel is fully synchronous: whether the first thread to operate on a channel wishes to send or receive, it will block until a thread intending to perform the opposite operation arrives; at that point, one item will be transferred from the sender to the receiver and both threads will be unblocked. Note that when a send operation on a synchronous channel completes the sender knows that the value has been received by some receiver (who presumably starts working on the value right away).

If the buffer capacity is non-zero, the channel is mostly-asynchronous: as long as the channel buffer isn't full, a send operation can quickly store its value in the buffer and return. It is important to observe two things about sends on buffered channels: first, because the buffer may be full at any given time, sends may block instead of returning quickly; second, when a send operation completes that does *not* imply that any receiver has picked up the item (it may still be in the buffer).

The channels you will be implementing will have three methods, `chan_create()`, `chan_send()`, and `chan_receive()`. When creating a channel, a buffer capacity is specified. The send operation takes a `void *` argument and the receive operation returns one.

Because the focus of this question is thread synchronization rather than buffer design, you may assume the existence of a buffer "library" as follows:

- `int buffer_init(buffer_t *b, int size)`
- `void buffer_enqueue(buffer_t *b, void *data)`
- `void* buffer_dequeue(buffer_t *b)`
- `int buffer_items(buffer_t *b)`
- `int buffer_capacity(buffer_t *b)`

For these buffers, `buffer_init()` initializes a buffer struct to have `size` capacity and pre-allocates that much space, returning zero on success and a negative value in the case of failure. The `enqueue()` function appends some data into the buffer; it will panic if the buffer is already full, but *will not otherwise fail* (i.e., it will not be troubled by heap exhaustion). Similarly, the `dequeue()` function returns the oldest piece of data in the buffer, but will panic if the buffer is empty. The `items()` function will return the number of items currently in the buffer. Finally, the `capacity()` convenience function returns the total capacity of the buffer as it was specified when `buffer_init()` was called. Aside from `buffer_init()`, these functions do not block and they do not contain any internal locking.

Your mission

You will implement `struct chan` with the following interface:

- `int chan_init(chan_t *chan, int capacity)`
- `void chan_send(chan_t *chan, void *data)`
- `void* chan_receive(chan_t *chan)`

You may use the `buffer_t` structure described previously in your implementation.

Your solution can use Project 2 thread library mutexes, condition variables, and/or semaphores, which you may assume to be strictly-FIFO if you wish. You may use `deschedule()`/`make_runnable()` if you must, though we don't recommend it; otherwise, you may not use other atomic or thread-synchronization operations, such as, but not limited to: reader/writer locks or any atomic instructions (`XCHG`,`LL/SC`). You may use various system calls not prohibited above, e.g., `get_ticks()`, if you wish, and non-synchronization-related thread-library routines in the "`thr_xxx()` family," e.g., `thr_getid()`. **For the purposes of the exam you should assume an error-free environment (memory allocation and initialization functions will always succeed; system calls and thread-library primitives will not detect internal inconsistencies or otherwise "fail"** (unless you indicate in your comments that you have arranged, and are expecting, a particular failure), etc.). You may wish to refer to the "cheat sheets" at the end of the exam.

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

Hint: it *is* possible to implement both the synchronous and asynchronous functionality in terms of common "core code" with one or more small modifications "around the edges." Such implementations might even be more likely to be correct...

Please declare a `struct chan` and implement `chan_init()`, `chan_send()`, and `chan_receive()` (you do not need to implement `chan_destroy()`).

```
typedef struct chan {









} chan_t;
```

...space for channel implementation...

...space for channel implementation...

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

5. [15 points] `Nuts & Bolts`.

In Project 2, you implemented a traditional 1:1 thread library in which the size of every created thread was identical and specified to `thr_init()` as the thread library was initialized.

This can be inconvenient for an application developer writing multi-threaded code. The stack size must be set according to the maximum needs of *any* code path which will run in any thread. This can result in a multi-threaded application allocating much more memory for thread stacks than is actually required.

"Segmented stacks" are one solution to this problem. In a segmented-stack system, each thread begins with a relatively small stack. When stack space is running low, the thread allocates a new block of space and begins using that space for its stack. If that space is eventually no longer needed, the thread will free it and return to using its previous stack. The Go programming language uses segmented stacks and an M:N thread library to efficiently support hundreds of thousands of Go threads (called "goroutines") at once. The LLVM compiler project has experimental support for compiling C code to use segmented stacks.

A C program can manually implement segmented stacks even if the compiler and standard run-time do not provide them. In this question we will ask you to implement, using a mixture of C and assembly language, a crude segmented-stack facility called `ss_call()`.

In particular, we will ask you to implement

```
void *ss_call(size_t stack_size, void *(*func)(void *), void *arg)
```

such that invoking

```
ss_call(20*1024, my_function, foo)
```

is equivalent to calling `my_function(foo)` except that the execution of `my_function()` will take place in a new 20-kilobyte stack segment, which will be freed after that invocation of `my_function()` returns.

(a) [10 points] Implement the `ss_call()` facility. We expect most solutions to be phrased in terms of a short C function and a short assembly-language routine. **For the purposes of the exam you should assume an error-free environment (memory allocation and initialization functions will always succeed; system calls and thread-library primitives will not detect internal inconsistencies or otherwise "fail"** (unless you indicate in your comments that you have arranged, and are expecting, a particular failure), etc.).

You may use this page as space for your `ss_call()` solution if you wish.

One awkward aspect of `ss_call()` is that an application developer still must guess how much stack space the function being called will need. During development, it would be helpful if memory corruption caused by stack overflows were detected "fairly quickly" when "easily possible."

(b) [5 points] Briefly, in a few sentences, describe a way that `ss_call()` can be modified to help detect such errors.

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 );
void mutex_destroy( mutex_t *mp );
void mutex_lock( mutex_t *mp );
void mutex_unlock( mutex_t *mp );

int cond_init( cond_t *cv );
void cond_destroy( cond_t *cv );
void cond_wait( cond_t *cv, mutex_t *mp );
void cond_signal( cond_t *cv );
void 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 );
void sem_wait( sem_t *sem );
void sem_signal( sem_t *sem );
void sem_destroy( sem_t *sem );

int rwlock_init( rwlock_t *rwlock );
void rwlock_lock( rwlock_t *rwlock, int type );
void rwlock_unlock( rwlock_t *rwlock );
void rwlock_destroy( rwlock_t *rwlock );
void rwlock_downgrade( 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: _____

Typing Rules Cheat-Sheet

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \to \tau \mid \mu\alpha.\tau \mid \forall\alpha.\tau \\ e & ::= & x \mid \lambda x{:}\tau.e \mid e\,e \mid \mathsf{fix}(x{:}\tau.e) \mid \mathsf{fold}_{\alpha.\tau}(e) \mid \mathsf{unfold}(e) \mid \Lambda\alpha.e \mid e[\tau] \end{array}$$

$$\frac{}{\Gamma,\, \alpha\,\mathbf{type} \vdash \alpha\,\mathbf{type}}\ \text{istyp-var} \qquad \frac{\Gamma \vdash \tau_1\,\mathbf{type} \quad \Gamma \vdash \tau_2\,\mathbf{type}}{\Gamma \vdash t_1 \to t_2\,\mathbf{type}}\ \text{istyp-arrow}$$

$$\frac{\Gamma,\, \alpha\,\mathbf{type} \vdash \tau\,\mathbf{type}}{\Gamma \vdash \mu\alpha.\tau\,\mathbf{type}}\ \text{istyp-rec} \qquad \frac{\Gamma,\, \alpha\,\mathbf{type} \vdash \tau\,\mathbf{type}}{\Gamma \vdash \forall\alpha.\tau\,\mathbf{type}}\ \text{istyp-forall}$$

$$\frac{}{\Gamma,\, x : \tau \vdash x : \tau}\ \text{typ-var} \qquad \frac{\Gamma,\, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash \tau_1\,\mathbf{type}}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ \text{typ-lam} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\,e_2 : \tau_2}\ \text{typ-app}$$

$$\frac{\Gamma,\, x : \tau \vdash e : \tau \quad \Gamma \vdash \tau\,\mathbf{type}}{\Gamma \vdash \mathsf{fix}(x{:}\tau.e) : \tau}\ \text{typ-fix}$$

$$\frac{\Gamma \vdash e : [\mu\alpha.\tau/\alpha]\tau \quad \Gamma,\, \alpha\,\mathbf{type} \vdash \tau\,\mathbf{type}}{\Gamma \vdash \mathsf{fold}_{\alpha.\tau}(e) : \mu\alpha.\tau}\ \text{typ-fold} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathsf{unfold}(e) : [\mu\alpha.\tau/\alpha]\tau}\ \text{typ-unfold}$$

$$\frac{\Gamma,\, \alpha\,\mathbf{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \text{typ-tlam} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau \quad \Gamma \vdash \tau'\,\mathbf{type}}{\Gamma \vdash e[\tau'] : [\tau'/\alpha]\tau}\ \text{typ-tapp}$$

$$\frac{}{\lambda x{:}\tau.e\,\mathbf{value}}\ \text{val-lam} \qquad \frac{}{\mathsf{fold}_{\alpha.\tau}(e)\,\mathbf{value}}\ \text{val-fold} \qquad \frac{}{\Lambda\alpha.\tau\,\mathbf{value}}\ \text{val-tlam}$$

$$\frac{e_1 \;\mapsto\; e_1'}{e_1\,e_2 \;\mapsto\; e_1'\,e_2}\ \text{steps-app}_1 \qquad \frac{e_1\,\mathbf{value} \quad e_2 \;\mapsto\; e_2'}{e_1\,e_2 \;\mapsto\; e_1\,e_2'}\ \text{steps-app}_2$$

$$\frac{e_2\,\mathbf{value}}{(\lambda x{:}\tau.e_1)\,e_2 \;\mapsto\; [e_2/x]e_1}\ \text{steps-app-}\beta$$

$$\frac{}{\mathsf{fix}(x{:}\tau.e) \;\mapsto\; [\mathsf{fix}(x{:}\tau.e)/x]e}\ \text{steps-fix}$$

$$\frac{e \;\mapsto\; e'}{\mathsf{unfold}(e) \;\mapsto\; \mathsf{unfold}(e')}\ \text{steps-unfold}_1 \qquad \frac{}{\mathsf{unfold}(\mathsf{fold}_{\alpha.\tau}(e)) \;\mapsto\; e}\ \text{steps-unfold}_2$$

$$\frac{e \;\mapsto\; e'}{e[\tau] \;\mapsto\; e'[\tau]}\ \text{steps-tapp}_1 \qquad \frac{}{(\Lambda\alpha.e)[\tau] \;\mapsto\; [\tau/\alpha]e}\ \text{steps-tapp}_1$$

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.