Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (B), Spring 2015

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

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

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] "BSS"

(b) [5 points] "Three kinds of error"

2. [15 points] Faulty Mutex

In this question you will examine some mutex code submitted for your consideration by your project partner. You should assume that `atomic_exchange()` works correctly (as described below), that the `gettid()` system call never "fails," that all queue functions behave in a strictly FIFO fashion, and that they will never fail. You should also assume "traditional x86-32 memory semantics," i.e., *not* "wacky modern memory."

```
/* queue.h */
int  q_init(queue_t *q);            // initializes a queue (to empty)
void q_enqueue(queue_t *q, int i);  // adds integer to queue at the end
int  q_dequeue(queue_t *q);         // removes+returns first integer in queue (else -1)
void q_destroy(queue_t *q);         // frees any resources used by queue
/* For the purposes of this exam, assume queue functions cannot fail.  */

/* atomic.h */
/**
 * Atomically:
 * (1) fetches the old value of the memory location pointed to by "target"
 * (2) places the value "source" into the memory location pointed to by "target"
 * Then: returns the old value (that was atomically fetched)
 *
 * Equivalently:
 *  ATOMICALLY {
 *    int previous_target = *target;
 *    *target = source;
 *  }
 *  return previous_target;
 */
extern int atomic_exchange(int *target, int source);
```

The remainder of this page is intentionally blank.

```
typedef struct {
    int turn;
    int locked;
    int queue_locked;
    queue_t waiting;
} mutex_t;

int mutex_init(mutex_t *m) {
    m->turn = -1;
    m->locked = 0;
    m->queue_locked = 0;
    return q_init(&m->waiting);
}

void mutex_lock(mutex_t *m) {
    if (atomic_exchange(&m->locked, 1)) {
        // someone else has the lock
        int tid = gettid();
        while (atomic_exchange(&m->queue_locked, 1))
            continue;
        q_enqueue(&m->waiting, tid);
        m->queue_locked = 0;
        while (m->turn != tid)
            continue;
    }
}

void mutex_unlock(mutex_t *m) {
    int tid;
    while (atomic_exchange(&m->queue_locked, 1))
        continue;
    if ((tid = q_dequeue(&m->waiting)) > 0) {
        m->turn = tid;
    } else {
        m->locked = 0;
    }
    m->queue_locked = 0;
}
/* ... remainder omitted, e.g., mutex_destroy() ... */
```

There is a problem with the mutex code shown above. 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.,

T1	T2
tid = gettid();	
	continue;

...

Be sure that the execution trace you provide us with is easy to read and conclusively demonstrates the claim you are making. You may introduce temporary variables or other obvious notation as necessary to improve the clarity of your answer. *You should report a problem with code that is visible to you rather than assuming a problem in code that you have not been shown.* It is possible to answer this question with a brief, clear trace, so you should do what is necessary to ensure that you do. 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 begin to write your solution on the next page. If we cannot understand the solution you provide, your grade will suffer!

This page is for your faulty-mutex solution.

3. [10 points] Deadlock.

Below you will find simple, prototype-quality code for a game system of some kind. Perhaps the code is used for the game's board, or perhaps the squares are pixels in a display—we don't care. The usage of the code is as follows.

- As you will see, the space being managed is large.
- Allocation and deallocation is according to lists of points; the points in a list may or may not be rectangular, may or may not be contiguous, etc. Each point list may be large or small.
- When the system is running, many areas will be allocated and deallocated by multiple threads. The designers of the system hope to optimize the throughput in terms of allocations/deallocations per second rather than the latency of single requests.
- *However*, the designers of the system obviously want to avoid *infinite* latency–meaning deadlock–for any allocations.

The remainder of this page is intentionally blank.

```
#include <stdlib.h>
#include <mutex.h>
#include <thread.h>
#include <cond.h>

typedef struct point {
  int x;
  int y;
} point_t;

typedef struct square {
  int available;
  mutex_t m;
  cond_t want;
} square_t;

#define COLS 10000
#define ROWS 10000

square_t array[ROWS][COLS];

void init_array(void) {
  for (int i = 0; i < ROWS; ++i) {
    for (int j = 0; j < COLS; ++j) {
      array[i][j].available = 1;
      mutex_init(&array[i][j].m);
      cond_init(&array[i][j].want);
    }
  }
}

void alloc_area(point_t points[], int npoints) {
  for (int p = 0; p < npoints; ++p) {
    square_t *sp = &array[points[p].x][points[p].y];
    mutex_lock(&sp->m);
    while (!sp->available) {
      cond_wait(&sp->want, &sp->m);
    }
    sp->available = 0;
    mutex_unlock(&sp->m);
  }
}

void free_area(point_t points[], int npoints) {
  for (int p = 0; p < npoints; ++p) {
    square_t *sp = &array[points[p].x][points[p].y];
    mutex_lock(&sp->m);
    sp->available = 1;
    cond_broadcast(&sp->want);
    mutex_unlock(&sp->m);
  }
}
```

(a) 5 points Unfortunately, the code shown above *can* deadlock. Show *clear, convincing* evidence of deadlock. Begin by *describing the problem* in one or two sentences; then *specify a scenario* (e.g., describe one or more function invocations), and finally *show a a "tabular execution trace"*. Missing, unclear, or unconvincing traces will result in only partial credit.

You may use this page for your deadlock trace if you wish.

(b) [3 points] Explain how to address the problem you described above via *deadlock prevention*. If your approach is correct and clear in your mind it should be possible for you to make it clear to us in two or three sentences or maybe a small scrap of code.

(c) [2 points] Is *deadlock avoidance* a good approach for solving this problem? Explain.

4. [20 points] Master-slave message passing

In your Project 2 thread library, you implemented various standard synchronization primitives such as mutexes, condition variables, semaphores, and reader/writer locks. On some systems, however, synchronization is done not by using these kinds of structures, but instead by sending messages between threads or processes.

In this question, you will implement a simplified version of some Open MPI functions that are used for inter-process communication. (Since your P2's dealt with threads and not processes, you'll write functions that communicate between threads.)

Specifically, you will implement functionality to allow sending a broadcast message from a master thread to many slave threads. Both of these functions act as "barriers." That is, the master thread (sending the message) and the slave threads (who want the message) can invoke their respective functions in any order; once all threads have "arrived," the information transfer takes place and all the threads are released to go about their business.

Your mission: You will implement `msm`s with the following interface:

- `void msm_init(msm_t *m, int num_slaves)` - Initializes an `msm_t`. The calling process becomes the master.
- `void msm_broadcast_send(msm_t *m, void *msg, size_t len)` - Send the message `msg`, which is `len` bytes, to each of the `num_slaves` slave threads that call `msm_broadcast_recv`.

  After this function returns, the master is free to do whatever it wants to `msg`, including `free()`ing it. Thus this function must not return while any slave might still examine the master's buffer.

  Must be called by the master.
- `int msm_broadcast_recv(msm_t *m, void *msg, size_t len)` - Receives a message from the master thread, blocking if necessary, storing it in `msg`, which has `len` bytes of space available.

  The slave should always receive as much of the message as possible. Returns the length of the message received if it was received in its entirety (and hence was not longer than `len` bytes), otherwise an integer less than 0.

  Must be called by exactly `num_slaves` slaves.

A minimal usage example follows on the next page.

The remainder of this page is intentionally blank.

```
#define BUFLEN 512
#define QUIT_STR "quit\n"
#define NUM_THREADS 32

// Mine bitcoins based on user input (tries different hashes based on id).
extern void bitcoin_mine(char *buf, int len, int id);

msm_t m;

void *slave(void *arg) {
    char buf[BUFLEN];
    int my_id = (int) arg;

    buf[0] = '\0';
    int res = msm_broadcast_recv(&m, buf, BUFLEN);
    assert(res >= 0);

    while (strcmp(buf, QUIT_STR)) {
        bitcoin_mine(buf, res, my_id);
        int res = msm_broadcast_recv(&m, buf, BUFLEN);
        assert(res >= 0);
    }
    return NULL;
}

int main(int argc, char **argv) {
    char buf[BUFLEN];
    msm_init(&m, NUM_THREADS);
    thr_init(PAGE_SIZE);

    int i;
    for (i = 0; i < NUM_THREADS; i++) {
        thr_create(&slave, (void *) i);
    }

    // Send all messages (including quit) to the slave threads
    do {
        int input_len = readline(BUFLEN, buf);
        msm_broadcast_send(&m, buf, input_len);
    } while (strcmp(buf, QUIT_STR));
    return 0;
}
```

Assumptions:

1. You may use regular Project 2 thread-library primitives: mutexes, condition variables, semaphores, reader/writer locks, etc.
2. You may *not* use other atomic or thread-synchronization synchronization operations, such as, but not limited to: `deschedule()`/`make_runnable()`, or any atomic instructions (`XCHG`, `LL/SC`).
3. You must comply with the published interfaces of synchronization primitives, i.e., you cannot inspect or modify the internals of any thread-library data objects.
4. You may not use assembly code, inline or otherwise.
5. If you wish, you may assume that the standard Project 2 thread-library primitives (mutex, condition variable, ...) are "as FIFO as possible."
6. **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).
7. You may use non-synchronization-related thread-library routines in the "`thr_xxx()` family," e.g., `thr_getid()`. You may wish to refer to the "cheat sheets" at the end of the exam. If you wish, you may assume that `thr_getid()` is "very efficient" (for example, it invokes no system calls).
8. If you wish, you may use the queue library described below. This queue *is not thread safe*, but because this is an exam you may assume that calls to it never fail.

- `void q_init(queue_t* q)` - Initializes the queue.
- `void q_enqueue(queue_t* q, void* data)` - Inserts `data` into the queue.
- `void* q_dequeue(queue_t* q)` - Removes the next item in the queue, returning the value of the removed item. Returns NULL if the queue is empty.
- `void q_destroy(queue_t* q)` - Destroys the queue, freeing any resources allocated by the queue.

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, your grade will suffer!

(a) [5 points] Please declare your `msm_t` here. If you need one (or more) auxilary structures, you may declare it/them here as well.

```
typedef struct {

} msm_t;
```

(b) [15 points] Now please implement `msm_init()`, `msm_broadcast_send()`, and `msm_broadcast_recv()`. You do not need to implement `msm_destroy()`.

. . . space for msm implementation . . .

. . . space for msm implementation . . .

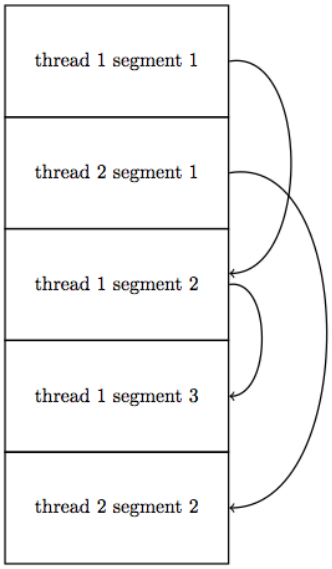
5. [20 points] `Nuts & Bolts`.

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.

One potential fix is to start the stacks a certain distance apart from each other and grow them through an autostack handler. However, this still puts an upper bound on the stack size, and attempting to increase the stack size may cause the thread stacks to run into the heap or other memory due to the large amount of space they take up.

"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. Note that this stack is not necessarily contiguous with the previous stack, so extra care must be taken to maintain the illusion of one stack. As they are allocated, the stack segments are tracked in a linked list so that they can be freed when the thread exits. Below is a diagram of how this might appear in memory:



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 typical way to implement segmented stacks is to change the compiler's code generator so that it inserts a "hook" at the beginning of each function; in this case the compiler will insert "`call segment_leap`" in the body of the function right after the prologue. The `segment_leap` code checks to see if the current value of `%esp` is "close" to the bottom of the segment. If so, it allocates a new segment and adjusts the stack so that the function which invoked it runs using the new segment for its stack space.

```
/* void swap(int *xp, int *yp) */
swap:
    /* prologue */
    pushl %ebp
    mov %esp, %ebp

    /* hook */
    call segment_leap

    /* function body */
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    popl %ebx

    /* epilogue */
    mov %ebp, %esp
    pop %ebp
    ret
```

Note that because `segment_leap` is an infrastructure hook rather than a regular function it runs in a situation which is not in accordance with the regular stack discipline. For example, the "real function" (here, `swap()`) doesn't save any registers before calling the hook (`segment_leap`).

For this question, you will be implementing `segment_leap`. You will be able to rely on the following three facilities.

```
/*
 * @return Returns the lowest accessible address in the current segment.
 */
void *segment_bottom(void);

/*
 * @brief When the current stack pointer is fewer than SEGMENT_THRESHOLD bytes
 *        above segment_bottom(), the current function should be run in
 *        the next segment.
 *        SEGMENT_THRESHOLD is large enough to store segment_leap's stacks,
 *        so long as it is "reasonably" small.
 */
#define SEGMENT_THRESHOLD ...
```

```
/*
 * @brief When called with %esp within one segment of a thread's stack,
 *        this function returns a pointer to the top word of the "next"
 *        segment of this thread's stack.  If the current segment is the
 *        most-recently-allocated one for this thread, a new segment will
 *        be allocated and returned; if the current segment is somewhere
 *        "in the middle" of the segments previously allocated for this
 *        thread's stack, the segment returned will be the one that was
 *        allocated just after the current segment was allocated.
 *
 *        You may assume that this function does not fail.
 *        You are not responsible for freeing segments.
 */
void *segment_next(void);
```

As an implementation detail, you may assume that the compiler has been adjusted so that "somehow" it uses only *positive* offsets to `%ebp` (as in the example above), and will address local variables, if any, using some other technique.

(a) [5 points] Draw a careful picture of the stack situation at the time when somebody called `swap(&zip1,&zip2)`, `segment_leap` decided to switch to a new segment, and the first instruction of `swap()`'s body code is about to run.

(b) [15 points] Now provide the code for `segment_leap`. You may use any tasteful mixture of C and x86-32 assembly code. Please make sure your intent is clear! It is likely that you should write down a draft version of your code using the scrap paper provided at the end of the exam, or on the back of some other page, *before* you begin to write your "official" solution. If we cannot understand the solution you provide, your grade will suffer!

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

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 readfile(char *filename, char *buf, int count, int offset);

/* "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.