Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Fall 2013

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.	**15**		
4.	**20**		
5.	**15**		
	75		

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.

   (a) [5 points] Explain either "Paradise Lost" or "TOCTTOU" *as the term 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.

(b) [5 points] Explain something that is dangerous about mixing `fprintf()` with Unix/Linux signal handlers. Be sure to describe both a scenario which would lead to trouble and what sort of trouble would plausibly occur.

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) and that the `gettid()` system call never "fails." You should also assume "traditional x86-32 memory semantics," i.e., *not* "wacky modern memory."

```
/**
 * 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:
 *  /* START ATOMIC SEQUENCE */
 *  int previous_target = *target;
 *  *target = source;
 *  /* END ATOMIC SEQUENCE */
 *  return previous_target;
 */
extern int atomic_exchange(int *target, int source);
```

The remainder of this page is intentionally blank.

```
typedef struct {
    int last_requested;  // tid of thread who most recently requested this lock
    int last_owner;      // tid of thread who most recently acquired this lock
    int locked;          // flag indicating lock is currently held
} lock_t;

int mutex_init(lock_t* lock) {
    lock->last_requested = -1;
    lock->last_owner = -1;
    lock->locked = 0;
    return 0;
}

void mutex_lock(lock_t* lock) {
    int me = gettid();
    int before_me = atomic_exchange(&(lock->last_requested), me);

    while (lock->last_owner != before_me) {
        continue;
    }

    while (lock->locked) {
        continue;
    }

    lock->locked = 1;
    lock->last_owner = me;
}

void mutex_unlock(lock_t* lock) {
    lock->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
me = 1;	
	me = 2;

...

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.

This page is for your faulty-mutex solution.

You may use this page as extra space for the faulty-mutex question if you wish.

3. [15 points] Deadlock.

Imagine that for fun you and your partner decide to port your Project 1 Battleship implementation so that it runs on the reference kernel you used to implement your Project 2 thread library. Furthermore, you and your partner decide to improve the code that the artificial intelligence (AI) uses when placing its ships on the board: you will speed up that phase of the game by using multiple threads to place the multiple ships (this is one way that computers are inherently superior to humans).

You and your partner agree that AI ship placement will use a new `place_ship()` interface as follows:

- The size of the ship is specified by the `length` parameter.
- The desired location of the ship is specified by a *starting coordinate* (`start_x`, `start_y`) and a `direction` along which the remainder of the ship is oriented (either `UP`, `DOWN`, `LEFT`, or `RIGHT`), where location (0,0) is in the top left-hand corner of the board. For example, if a ship of `length` 3 has a starting coordinate of (2,3) and a `direction` of `RIGHT`, then the desired location of that ship is at coordinates (2,3), (3,3), and (4,3). If a ship of `length` 4 has a starting coordinate of (7,7) and a `direction` of `UP`, then the desired location of that ship is at coordinates (7,7), (7,6), (7,5), and (7,4).
- Because each thread in the AI engine is choosing potential ship locations at random while other threads are also placing ships, there is a chance that multiple target locations might overlap each other, which would be a problem. Hence the `place_ship()` interface *returns a boolean value* indicating whether it successfully placed the entire ship without overlapping any part of it with another ship that had already been placed on the board. If an attempt to place a ship is unsuccessful, the thread placing that ship will select another random location and retry `place_ship()`.

Your partner on this project (Project P1$\frac{1}{2}$) has implemented an initial draft of the `place_ship()` interface, as shown on the next page. Here are some details regarding this implementation:

- The `ship_position_occupied[i][j]` 2-D boolean array is used to record which locations on the game board already contain legally-placed ships.
- To help prevent race conditions, a 2-D array of mutex variables (`ship_position_locks[i][j]`) is also used.
- Assume that the fact that the `place_remainder_of_ship()` routine executes recursively does not create a problem (because we have ported our P1 Battleship to user space and our threads have been created using sufficiently-large stacks, we do not need to worry about stack overflow).

The remainder of this page is intentionally blank.

```
typedef int bool;
#define TRUE 1
#define FALSE 0
enum ship_direction_t {UP, DOWN, LEFT, RIGHT};

bool index_within_board(int index) {
   return ((index >= 0) && (index < BOARD_SIZE));
}

bool place_ship(int length, int start_x, int start_y, ship_direction_t direction) {
   int delta_x, delta_y;
   switch (direction) {
      case UP:    delta_x = 0;  delta_y = -1; break;
      case DOWN:  delta_x = 0;  delta_y = 1;  break;
      case LEFT:  delta_x = -1; delta_y = 0;  break;
      case RIGHT: delta_x = 1;  delta_y = 0;  break;
      default: assert(FALSE);
   }
   assert((length >= MIN_SHIP_LENGTH) && (length <= MAX_SHIP_LENGTH));

   int end_x = start_x + (length-1)*delta_x;
   int end_y = start_y + (length-1)*delta_y;
   assert(index_within_board(start_x));
   assert(index_within_board(start_y));
   assert(index_within_board(end_x));
   assert(index_within_board(end_y));

   return place_remainder_of_ship(length, start_x, start_y, delta_x, delta_y);
}

bool place_remainder_of_ship(int length, int x, int y, int delta_x, int delta_y) {
   bool placed_successfully = TRUE;    /* value to return when length decreases to zero */
   if (length > 0) {
      /* more parts of the ship to place, grab a lock to avoid race conditions */
      mutex_lock(&ship_position_locks[x][y]);

      /* check whether this position is already occupied by another ship */
      if (ship_position_occupied[x][y]) {
         /* if so, then skip the rest of the ship */
         placed_successfully = FALSE;
      } else {
         /* OK so far; attempt to place the rest of the ship (via recursion) */
         placed_successfully = place_remainder_of_ship((length-1), (x+delta_x), (y+delta_y),
                                                       delta_x, delta_y);
         if (placed_successfully) {
            /* if we were successful, then mark this position as occupied */
            ship_position_occupied[x][y] = TRUE;
         }
      }
      mutex_unlock(&ship_position_locks[x][y]);
   }
   return placed_successfully;
}
```

(a) [8 points] Unfortunately, this implementation of `place_ship()` can deadlock. Show a *clear, convincing* execution trace that yields a deadlock. (Missing, unclear, or unconvincing traces will result in only partial credit).

(b) [7 points] *While preserving the functionality of the* `place_ship()` *interface*, briefly describe how to fix or restructure the code that implements that interface so that it does not deadlock. We expect that most successful solutions will be in the form of a small "patch" to the code. However, solutions which are clear and convincing without code will receive full credit. Be sure to explain how your solution addresses one or more deadlock ingredient(s).

4. [20 points] Master-Slave Mutex

In this question, we will consider a lock which could be useful in certain situations. Specifically, imagine a system where one master thread begins a large data processing task, during which it spawns many worker threads. In this situation, it is plausible that the master's task is "more important" than the workers' tasks, so a strictly-fair lock might slow the whole system down by making the master thread wait in line like everybody else. For example, perhaps the master needs to periodically check the system state to see if the slave threads have computed an acceptable answer to some scientific problem, in which case the application should quickly display the answer and start working on a new problem.

To improve the system, you will implement a "master-slave mutex." This object's behavior is basically the same as that of a regular mutex, except that the thread which initialized the mutex (the "master") has higher priority when handling the lock than any other thread. This has two consequences with respect to the lock's behavior. First, when the object is unlocked, if the master is waiting on the lock, then the master must acquire the lock before any slave thread. Second, only the master can destroy the lock, **but it can do so at any point at which it holds the lock**. When the master thread destroys the lock, **any outstanding attempts to acquire the lock must fail with an error code**.

Assumptions:

1. You may use regular Project 2 thread-library primitives: mutexes, condition variables, semaphores, 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. If you wish, you may assume that the standard Project 2 thread-library primitives (mutex, condition variable, ...) are "as FIFO as possible."
4. **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).
5. You may **not** rely on any data-structure libraries such as splay trees, red-black trees, queues, stacks, or skip lists, lock-free or otherwise, that you do not implement as part of your solution.
6. You may use non-synchronization-related thread-library routines in the "`thr_xxx()` family," e.g., `thr_getid()`.

API notes:

1. You should assume that by the time the master thread begins to destroy the lock it has somehow ensured that no more slave threads will begin attempts to acquire the lock.
2. You should assume that the master thread never does anything illegal, such as destroy an uninitialized or already-destroyed lock, acquire a lock when it already holds it, destroy a lock when it doesn't already hold the lock, etc.
3. You should also assume that the slave threads behave properly, e.g., they will not try to acquire the lock before it is initialized, after it is destroyed, or when they already hold it; they will not attempt to destroy the lock, etc.

4. Once the master thread finishes destroying the lock, it is allowed to free the memory, use the memory for something else, or re-initialize the lock.

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!

Serious hint: before writing any "real code," write a "rough draft" of `msmutex_destroy()`.

You will provide us with both a structure definition for a "master-slave mutex" and the code for the following four functions:

```
void msmutex_init(msmutex_p msm);
int msmutex_lock(msmutex_p msm); // returns error in destroy case, else 0
void msmutex_unlock(msmutex_p msm);
void msmutex_destroy(msmutex_p msm);
```

The remainder of this page is intentionally blank.

Please declare a `struct msmutex` and implement `msmutex_init()`, `msmutex_lock()`, `msmutex_unlock()`, and `msmutex_destroy()`.

```
typedef struct msmutex {









} *msmutex_p;
```

...space for master-slave mutex implementation...

...space for master-slave mutex implementation...

You may use this page as extra space for your master-slave mutex solution if you wish.

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

Your friend, Grave O'Danger, is excited about a new idea for running C functions. He believes that performance will be improved by running certain functions inside their own independent stack regions (which we could call "stacklets")—for some reason he believes that "the cache will be less polluted" if some functions run "far away from" other functions. Instead of running `f(x)` the usual way, he wants to invoke `far_call(f,x)`; the `far_call()` function will allocate some memory for a new stacklet, and "somehow" invoke `f()` so that it runs on the new stacklet. Once `f()` returns, regular execution will resume on whatever stack was used before `far_call()` was invoked.

Grave has begun hacking together a crude prototype of `far_call()` as follows.

```
typedef (*intfun)(int);

extern wrapper(char *stack_high, intfun f, int arg); // TBD

#define FAR_SIZE 4096

int far_call(intfun f, int arg) {
    int result;
    unsigned char *stack_low, *stack_high;

    stack_low = malloc(FAR_SIZE); // Guaranteed to be 8-byte aligned

    if (stack_low) {
        stack_high = stack_low + FAR_SIZE - 8;
        result = wrapper(stack_high, f, arg);
        free(stack_low);
    } else {
        // We encountered a "Hmm..." error; try to work around it.
        result = f(arg);
    }
    return (result);
}
```

Grave understands that the `wrapper()` function will need to be written in assembly code, but so far all of his attempts have resulted in program crashes. Because you are a helpful stack expert, your mission will be to write the wrapper for him. You should assume the standard "Linux x86-32" stack convention that we have been using.

Please write code for the `wrapper()` function. In addition to the code, you may provide documentation, such as a *clear* supporting diagram, which we will try to use to better understand your code. However, your score will be derived from your code as we understand it (i.e., diagrams without code will not score very highly).

You may use this page for your `wrapper()` solution 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.