Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Fall 2014

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.	**20**		
4.	**20**		
5.	**10**		
	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.

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] "South Bridge"

2. [15 points] Concurrency

As your first project at your new internship at the hot new startup "Yo!", you are given the responsibility of verifying the correctness of Yo's backend server code—in particular, a library responsible for allowing one master thread to generate work to be completed by a fixed-size collection of worker threads. To your surprise, you find the code quality not up to the standards you have come to expect based on your studies at this highly esteemed institution of higher education: the work-distribution code is broken!

When reading the following code, you should assume that the pointers passed to `master()` and `worker()` are valid; the `workqueue_t` struct has been properly initialized; mutexes and condition variables have been initialized before use; `generate_some_work()` and `do_important_work()` return in finite time without tromping on work-distribution state variables, etc. *In general, 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.* At any given time, it is guaranteed that there will be at most one instance of `master()` running and at most (`WORK_QUEUE_SIZE`−1) instances of `worker()` running.

```
#define WORK_QUEUE_SIZE 64
#define WRAPPED_NEXT(i) (((i) + 1) % (WORK_QUEUE_SIZE))

typedef int work_t; // for exam purposes:  in real life, "work" is more exotic

typedef struct {
    work_t queue[WORK_QUEUE_SIZE];
    int next_write; // initialized to 0
    int next_read;  // initialized to 0
    mutex_t lock;
    cond_t writer_cv;
    cond_t reader_cv;
} workqueue_t;

/* Guaranteed to only be one of these running. */
int master(workqueue_t *work) {
    while (1) {
        work_t new_work = generate_some_work();
        mutex_lock(&work->lock);
        if (WRAPPED_NEXT(work->next_write) == work->next_read) {
            cond_wait(&work->writer_cv, &work->lock);
        }
        work->queue[work->next_write] = new_work;
        work->next_write = WRAPPED_NEXT(work->next_write);
        if (work->next_write == WRAPPED_NEXT(work->next_read)) {
            cond_broadcast(&work->reader_cv);
        }
        mutex_unlock(&work->lock);
    }
}
```

```
/* Guaranteed to be strictly fewer than WORK_QUEUE_SIZE running. */
int worker(workqueue_t *work) {
    while (1) {
        mutex_lock(&work->lock);
        int next_read_index = work->next_read;
        while (work->next_write == next_read_index) {
            cond_wait(&work->reader_cv, &work->lock);
        }
        int was_full = (WRAPPED_NEXT(work->next_write) == work->next_read);
        work_t to_do = work->queue[work->next_read];
        work->next_read = WRAPPED_NEXT(work->next_read);
        if (was_full) {
            cond_signal(&work->writer_cv);
        }
        mutex_unlock(&work->lock);
        do_important_work(to_do);
    }
}
```

Unfortunately, this code contains a concurrency bug. We will ask you to briefly explain (in 1 to 3 sentences) how a concurrency bug occurs; we will also ask you to provide a *clear, convincing* execution trace that demonstrates the error (missing, unclear, or unconvincing traces will result in significant point deductions). Finally, we will ask you to sketch out a solution to the problem you identified.

Suggestions for working on this problem:

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

master	worker 1
wait(writer)	
	...
	signal(writer)
	unlock(lock)
...	

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 begin to write your solution on the next page. If we cannot understand the solution you provide, your grade will suffer!
3. Be sure that your trace can actually happen, and that it convincingly shows the phenomenon you wish to demonstrate. For example, if you claim that the work queue data structure can become corrupted in some way, you should show a concrete sequence of statements and values which lead to a specific corrupt state.

(a) [3 points] Briefly describe (1 to 3 sentences) the concurrency bug you have identified. If you find multiple concurrency bugs, describe one that is relatively easy to trace (below), as long as it's "at least reasonably interesting."

(b) [10 points] Provide a *clear, convincing* execution trace that demonstrates the error you identified.

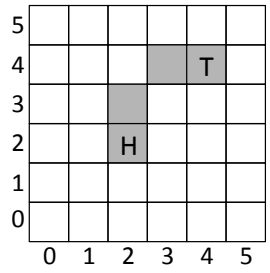
You may use this page as extra space for the first part of the work-distribution question if you wish.

(c) [2 points] Briefly describe how to fix or restructure the code so that it does not contain the bug you identified in parts (a) and (b). Full-credit answers which are clear and convincing don't necessarily need to show code, though we expect many answers will involve at least a bit of code.

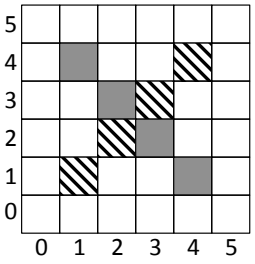
3. [20 points] Deadlock.

During a summer internship, you have been hired to write system software for a company that sells snake-like robots. Here are some details regarding the robo-snakes and how they are programmed:

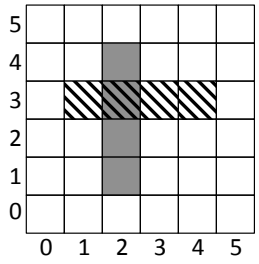
Occupying space: The robotic snakes move along the ground, which is represented by a 2D grid. Each robo-snake consists of *four contiguous segments*, where each segment occupies its own coordinate in the 2D space (no two segments can share the same position). As one follows the chain of segments from head to tail, contiguous segments must be *adjacent* to each other in the 2D space either *horizontally* (i.e. along the X dimension), *vertically* (i.e. along the Y dimension), or *diagonally* (i.e. along both X and Y). For example, a given robo-snake might occupy the following coordinates (from head to tail): $\{(2,2),(2,3),(3,4),(4,4)\}$.



Enforcing mutual exclusion: Multiple robo-snakes can operate within the same 2D space at the same time, as long as no part of them occupies the same grid coordinate. It is okay for two snakes that are arranged diagonally to cross over each other, as long as no part of them is in the same coordinate (as shown below). To enforce this invariant, the robo-snake control software uses 2D arrays of (i) mutex locks (called "`grid_occupied_lock`") and (ii) flags (called "`grid_occupied_flag`").



Diagonal crossing okay.
(Not in same coordinate.)



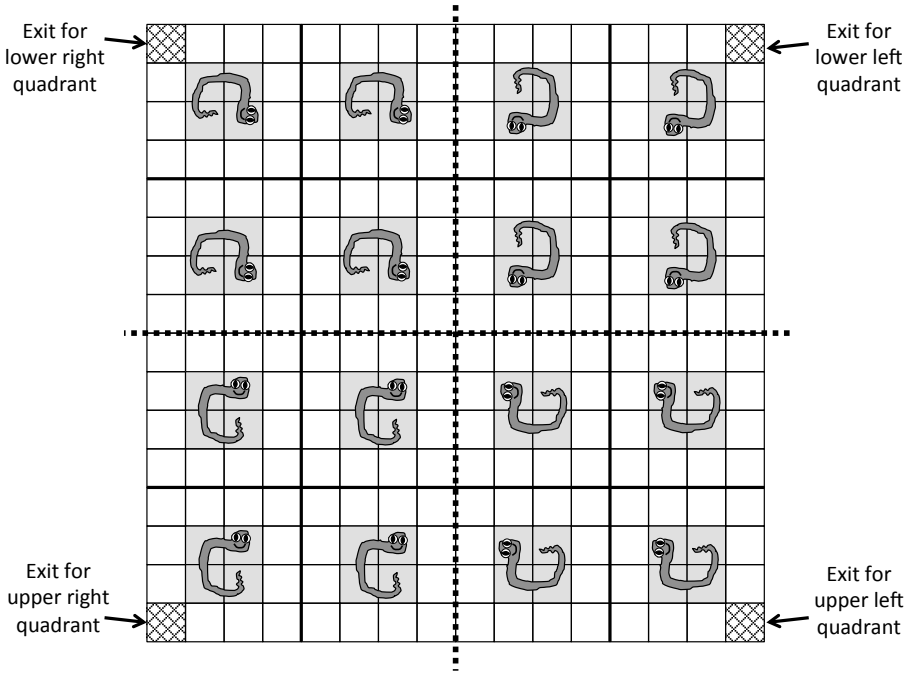
Not okay! Both
occupy (2,3).

Robo-snake motion: A robo-snake moves by advancing its head forward *one position*—either horizontally, vertically, or diagonally—into an unoccupied space in the 2D grid. When this happens, the last tail segment of the snake is retracted from the space that it previously occupied; hence the snake will always occupy four adjacent positions on the grid as it moves. The appropriate elements of the `grid_occupied_lock` array are locked and unlocked as this occurs, as shown in the `move_snake_one_step()` procedure on a subsequent page.

(Continued on the next page.)

Initial placement of robo-snakes: Before your robo-snake software begins, the system is initialized as illustrated below. (Note that this step is beyond your control.) You should *assume the following about the initial placement* of the robo-snakes (in addition to the "occupying space" discussion above):

- The 2D space is evenly divided into 4x4 sub-grids, where the four segments of each snake are placed within the 2x2 square in the center of its sub-grid, leaving the outer squares of the 4x4 sub-grid initially empty. (Note that this condition does not necessarily hold once the snakes start to move.) Although this picture shows a 16x16 grid, the code must operate on larger grid sizes also (that are even multiples of sub-grid size).
- Within the 2x2 square where each snake is initially located, the snake's *head* segment is in the square closest to the center of the overall 2D space (as illustrated below).
- Each snake has successfully locked the four `grid_occupied_lock` elements corresponding to its location, and set the corresponding `grid_occupied_flag` elements to 1.



Robo-snakes move concurrently to exits: After the snakes are successfully placed in their initial locations, each snake is given a target *exit coordinate*, which is a specific square where it can exit the 2D space. For each snake, its exit coordinate is the corner of the 2D space that is farthest away from its initial location. For example, snakes located in the top left quadrant must exit through the bottom right corner of the 2D space, etc. Each snake must move concurrently through the 2D space such that its head arrives at the its target exit coordinate while its body trails along intact, such that none of its segments occupy the same square as another snake along the way. Assume that once a snake's head reaches its exit, the snake disappears (releasing its locks). The initial draft of the code that attempts to accomplish this is the "`move_snake_to_target()`" procedure, which is called by concurrently running processes that control each snake. An important metric of success is getting the snakes to their exits as quickly as possible.

The next two pages show the current implementation of the robo-snake control software, which uses a simple greedy algorithm to move each snake toward its target.

```
#define SNAKE_LENGTH 4   /* Number of segments in a snake */
#define SUBGRID_SIZE 4   /* Size of sub-grid, used for initial placement */
#define GRID_SIZE 64     /* Overall size of 2D space (larger than illustration) */
#define NUM_SNAKES ((GRID_SIZE/SUBGRID_SIZE)*(GRID_SIZE/SUBGRID_SIZE))
mutex grid_occupied_lock[GRID_SIZE][GRID_SIZE];
int   grid_occupied_flag[GRID_SIZE][GRID_SIZE];

typedef struct {   /* 2D coordinate */
   int x;
   int y;
} location;

struct snake_info_struct {
   int head_index;         /* Circular buffer representing the snake's location */
   location segment[SNAKE_LENGTH];
   location my_exit;       /* assume initialized to farthest-away corner */
} snake_state[NUM_SNAKES];

/* Moves a given snake robot from current to target location. */
void move_snake_to_target(int which_snake) {
   struct snake_info_struct *this_snake = &snake_info[which_snake];
   int head   = this_snake->head_index;
   int head_x = this_snake->segment[head].x;
   int head_y = this_snake->segment[head].y;
   int target_x = this_snake->my_exit.x;
   int target_y = this_snake->my_exit.y;

   /* Repeatedly move snake one step at a time until it reaches target. */
   while ((head_x != target_x) || (head_y != target_y)) {
      int delta_x, delta_y;
      calculate_delta_xy(target_x, target_y, head_x, head_y, &delta_x, &delta_y);
      move_snake_one_step(which_snake, delta_x, delta_y);
      head_x += delta_x;
      head_y += delta_y;
   };
   return;
}

/* Head of snake moves by only one grid position at a time (including diagonally). */
void calculate_delta_xy(int target_x, int target_y, int head_x, int head_y,
                        int *delta_x, int *delta_y) {
   /* Simple greedy algorithm: move one step closer to target. */
   calculate_greedy_delta(target_x, head_x, delta_x);
   calculate_greedy_delta(target_y, head_y, delta_y);
}

int calculate_greedy_delta(int target, int current, int *delta) {
   assert((target >= 0) && (target < GRID_SIZE));
   if (target > current) *delta = 1;
   else if (target < current) *delta = -1;
   else *delta = 0;
}
```

```
/* Move snake robot one step.  The "delta" values are either 0, 1, or -1. */
void move_snake_one_step(int which_snake, int delta_x, int delta_y) {
   struct snake_info_struct *this_snake = &snake_info[which_snake];
   int old_head   = this_snake->head_index;
   int old_head_x = this_snake->segment[old_head].x;
   int old_head_y = this_snake->segment[old_head].y;
   int new_head_x = old_head_x + delta_x;          /* Calculate new head location. */
   int new_head_y = old_head_y + delta_y;
   int old_tail   = (old_head + 1) % SNAKE_LENGTH;  /* Vacating old tail location. */
   int old_tail_x = this_snake->segment[old_tail].x;
   int old_tail_y = this_snake->segment[old_tail].y;

   mutex_lock(&grid_occupied_lock[new_head_x][new_head_y]);  /* acquire new head */
   grid_occupied_flag[new_head_x][new_head_y] = 1;      /* mark space as occupied */
   grid_occupied_flag[old_tail_x][old_tail_y] = 0;      /* mark space as unoccupied */
   mutex_unlock(&grid_occupied_lock[old_tail_x][old_tail_y]);  /* release old tail */

   /* update circular buffer that stores snake segment locations */
   int new_head_index = old_tail;
   this_snake->segment[new_head_index].x = new_head_x;
   this_snake->segment[new_head_index].y = new_head_y;
   this_snake->head_index = new_head_index;
}
```

Unfortunately, this implementation of `move_snake_to_target()` can deadlock.

(a) [3 points] Using the four necessary conditions for deadlock, explain why this implementation meets all of those conditions.

(b) [9 points] If the `calculate_delta_xy()` procedure is modified to use a different algorithm for controlling the motion of the snakes (along with other necessary code changes), can *deadlock prevention* be successfully applied in this case? (Note that the values of `delta_x` and `delta_y` must still be either 0, 1, or -1.) Assume that the `move_snake_one_step()` should still perform at least the same steps, and that you are free to add additional synchronization (or other state and computation) throughout the code as necessary. If so, then show code (or at least detailed pseudo-code) that implements *deadlock prevention*, and explain why it works. If not, then provide a convincing explanation of why this cannot work.

...space for deadlock implementation...

(c) [8 points] Rather than using deadlock prevention, your boss has asked you to instead use *deadlock avoidance* to fix the original robo-snake control code (shown on the previous pages). Can you successfully implement *deadlock avoidance* for this case? (Keep in mind that a key metric of success is how quickly you can remove all snakes from the 2D space.) If so, show what changes you would make to the original code (show either real code or detailed pseudo-code), and describe the algorithm that the new *resource manager* would use to decide whether to grant or defer requests. If not, then provide a convincing explanation of why deadlock avoidance is not possible for this scenario.

...space for deadlock implementation...

4. [20 points] "Select variables"

The `select()` system call allows a Unix process to wait on a set of file descriptors (typically network sockets) until at least one of them becomes "ready for I/O", meaning that a `read()` or `write()` system call would return promptly instead of blocking. In a naive server architecture the programmer must dedicate an entire thread to each network connection to serve requests for just that connection. The `select()` system call enables a programmer to set up one thread, or perhaps a pool of threads, to repeatedly receive, and then process, the next request regardless of which connection it's from.

In this problem you will work on a new kind of condition variable, called a "select variable" or "svar", which exhibits behavior similar to that of `select()`. In particular, each select variable will be initialized to contain "wait channels"; a thread waiting on a select variable will provide a list of channels to wait on, and will stop waiting when any of those channels is signaled.

This object will be initialized with an integer, N, which specifies how many channels it shall have available to wait on. Its behavior is generally identical to a condition variable, except that the waiter may specify any subset of the N channels to wait on. When any one of those channels is signaled the next waiter for that channel must be awakened. As is the case with a condition variable, waiting threads should be blocked when appropriate and unblocked when appropriate.

You will provide us with both a structure definition for a "select variable" and the code for the following four functions:

- `void svar_init(svar_t* sv, int n)` - Initializes a select variable with `n` channels.
- `int svar_wait(svar_t* sv, int* idxs, int nidxs)` - Waits on a subset of the `n` channels and returns the index of whichever one was signaled. This subset is specified in the array `idxs` of length `nidxs`.
- `void svar_signal(svar_t* sv, int chn)` - Signals a waiter on channel `chn` or does nothing if there are no such waiters.
- `void svar_destroy(svar_t* sv)` - Destroys a select variable.

The remainder of this page is intentionally blank.

As a usage example consider the following trace:

```
// Thread 0
int t0_chans[] = {0, 1};
int t0_n = sizeof (t0_chans) / sizeof (t0_chans[0]);
int index;
svar_init(&sv, 3);
index = svar_wait(&sv, t0_chans, t0_n);

// Thread 1
int t1_chans[] = {1, 2};
int t1_n = sizeof (t1_chans) / sizeof (t1_chans[0]);
int index;
index = svar_wait(&sv, t1_chans, t1_n);

// Thread 2
int t2_chans[] = {0, 2};
int t2_n = sizeof (t2_chans) / sizeof (t2_chans[0]);
int index;
index = svar_wait(&sv, t2_chans, t2_n);

// Thread 3, after some time
svar_signal(&sv, 1); // awaken T0 with index == 1
svar_signal(&sv, 1); // awaken T1 with index == 0
svar_signal(&sv, 1); // awaken nobody
```

The remainder of this page is intentionally blank.

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 use non-synchronization-related thread-library routines in the "`thr_xxx()` family," e.g., `thr_getid()`.
6. You may use the queue library described below.
7. You may assume callers are well-behaved (e.g., specify channel numbers and indices that are non-negative and strictly less than N).

You may further assume the existence of a basic queue data structure. 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.
- `int q_remove(queue_t* q, void* data)` - Removes the first instance of `data` from the queue. Returns 0 on success and -1 if the provided value is not in the queue.
- `void* q_head(queue_t* q)` - Returns the next item to be dequeued, without removing it.
- `void* q_tail(queue_t* q)` - Returns the latest item enqueued, without removing it.
- `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 on the next page, your grade will suffer!

The remainder of this page is intentionally blank.

Please declare your `struct svar` here. If you need one (or more) auxilary structures, you may declare it/them here as well. Then please implement `svar_init()`, `svar_wait()`, `svar_signal()`, and `svar_destroy()`.

```
typedef struct svar {









} svar_t;
```

...space for select-variable implementation...

...space for select-variable implementation...

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

5. [10 points] Process model.

Imagine that you and your partner are working away on your Project 2 thread library and that you are watching your partner single-step through some user-space code with Simics. Imagine further that the user-space code is about to execute a `PUSHL` instruction when you are momentarily distracted by an incoming tweet, InstaSnap, etc.

When you look back at the screen, you see that once again the instruction Simics is about to single-step through is a `PUSHL`, but you notice that the instruction's address looks a little odd, and your partner tells you that "for some reason" you are now in kernel mode.

Because you like puzzles more than you like scrolling backward, you decide to brainstorm to come up with reasons why execution entered the kernel.

> Please list three plausible reasons or paths by which execution might have transferred into the kernel. Explain each in sufficient detail to convince your grader that you understand the key concepts well. You are allowed to specify that particular instructions or conditions occurred between the two `PUSHL` instructions if you wish. Some points will be assigned for "plausible novelty," meaning that we are looking for reasons/paths that are as *conceptually different* from each other as possible, while not unduly sacrificing plausibility.

You may use this page as extra space for your process model 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.