Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (B), Spring 2016

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.	10		
	65		

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

I have not received advance information on the content of this 15-410/605 mid-term exam by discussing it with anybody who took part in the main exam session or via any other avenue.

Signature: ____________________________________ Date ________________________

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

If we cannot read your writing, we will be unable to assign a high score to your work.

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

   (a) $\boxed{\text{5 points}}$ If we compare a runnable thread to a blocked thread, typically they entered the kernel in different ways, and typically they will leave the kernel for different reasons. Explain.

(b) $\boxed{\text{5 points}}$ Briefly describe the two typical kinds of thread cancellation, and explain a serious negative feature of each one.

2. $\boxed{\text{10 points}}$ Consider the following critical-section protocol:

```
#define NTHREAD 32 // a global static limit on the number of threads!
int tid(); // returns thread id from 0..(NTHREAD-1)
int lock_available = 1;
int waiting[NTHREAD] = { 0, };
int atomic_exchange(int *ip, int val); // behaves as expected

void lock() {
    int i = tid(), got_it = 0;

    waiting[i] = 1;
    while (waiting[i] && !got_it) {
      got_it = atomic_exchange(&lock_available, 0);
    }
    waiting[i] = 0;
}

void unlock() {
    int i;

    for (i = 0; i < NTHREAD; ++i) {
        if (waiting[i]) {
            waiting[i] = 0;
            return;
        }
    }
    atomic_exchange(&lock_available, 1);
}
```

(a) $\boxed{\text{7 points}}$ 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
w[0] = 1;	
	w[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.

Please don't accidentally skip the second part of this problem.

You may use this page for the critical-section protocol question.

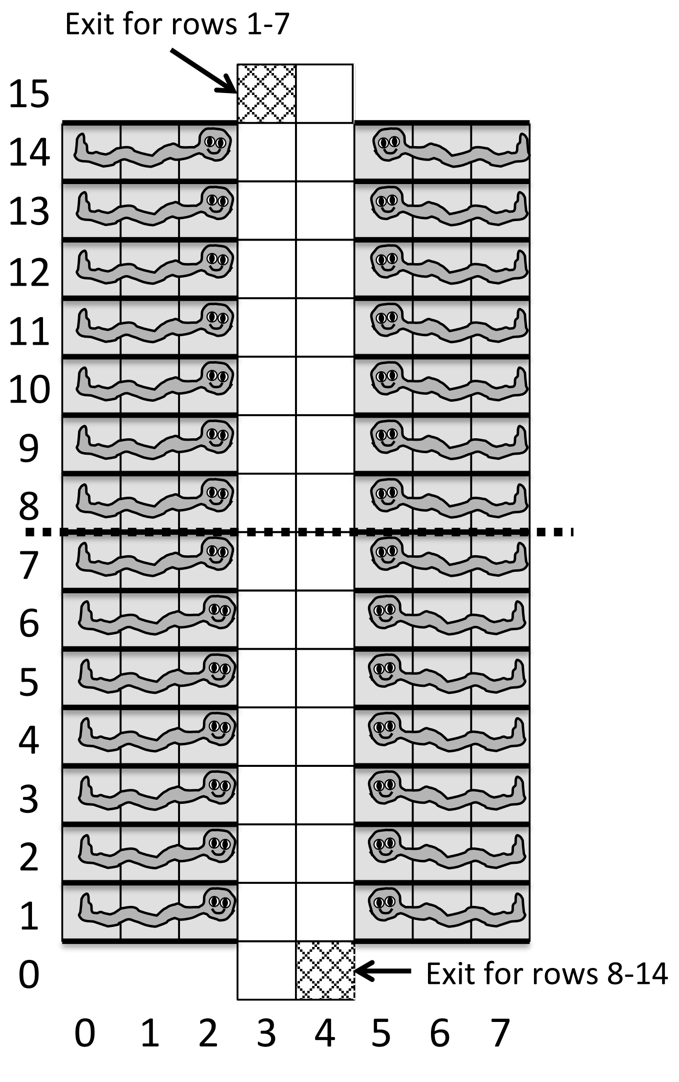
Andrew ID: ______________

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

(b) $\boxed{\text{3 points}}$ Please *clearly* demonstrate (probably using code, though *very clear* explanatory text might work also) how to handle the problem you described above.

3. $\boxed{\text{15 points}}$ Snakes on a Plane!

You have been hired as a summer intern at a movie studio that is making a sequel to "Snakes on a Plane." Instead of using computer-generated imagery ("CGI") for the special effects, however, your team has created robotic snakes. Your mission is to write the software that controls the snakes. In particular, when Samuel L. Jackson says, "I have had it with these m***** snakes on this m***** plane!" your team is counting on you to make the robo-snakes move in a quick and orderly fashion toward their designated exits without causing deadlock. The figure below shows the layout of the airplane.



The remainder of this page is intentionally blank.

Here are some more details regarding the robo-snakes and how they are programmed:

Coordinate system of the airplane: As illustrated in the figure on the previous page, the airplane is represented by a 2D coordinate system that includes 16 rows (0-15) and 8 columns (0-7). Columns 3 and 4 are the aisle, and the other columns are the seats. Seats exist only in rows 1 through 14 of the grid; rows 0 and 15 in the grid contain only aisle spaces, and this includes the exit areas for the snakes.

Occupying space: Each robo-snake consists of *three 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) or *vertically* (i.e. along the Y dimension). Hence each 3-segment snake will either be in a straight line (either horizontally or vertically) or it can be in an L-shaped right angle.

Enforcing mutual exclusion: Multiple robo-snakes can operate within the 2D space of the airplane at the same time, as long as no part of them occupies the same grid coordinate. To enforce this invariant, the robo-snake control software uses a 2D array of mutex locks (called "`grid_occupied_lock`").

Robo-snake motion: A robo-snake moves by advancing its head forward *one position*—either horizontally or vertically (but not diagonally)—into an unoccupied aisle 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 three 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.

Initial placement of robo-snakes: As illustrated in the figure on the previous page, each snake is initially placed in a set of seats with its head pointing toward the aisle. Each snake has successfully locked the three `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 the snakes that are initially in rows 1-7, their exit coordinate is (3,15), and for the snakes that are initially in rows 8-14, their exit coordinate is (4,0) (as illustrated in the figure). Hence each snake must cross to the opposite half of the airplane in order to exit (this is a strict requirement). In addition, snakes can advance only into aisle coordinates (i.e. columns 3 and 4); they cannot advance into seats (i.e. columns 0-2 and 5-7) once they start to move. 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 3   /* Number of segments in a snake */
#define GRID_ROWS 16     /* Number of rows in the 2D airplane grid */
#define GRID_COLUMNS 8   /* Number of columns in the 2D airplane grid */
#define NUM_SNAKES (2*(GRID_ROWS-2)) /* One snake per seat */
mutex_t grid_occupied_lock[GRID_COLUMNS][GRID_ROWS];
int     grid_occupied_flag[GRID_COLUMNS][GRID_ROWS];

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;       /* (3,15) for snakes in rows 1-7, (4,0) for snakes in rows 8-14 */
} 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_state[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. */
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);
   if (*delta_x == 0)   /* Only move in the y axis we aren't moving in the x axis. */
      calculate_greedy_delta(target_y, head_y, delta_y);
   else *delta_y = 0;
}

void calculate_greedy_delta(int target, int current, int *delta) {
   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;
}
```

(a) $\boxed{\text{2 points}}$ Unfortunately, this implementation of `move_snake_to_target()` can deadlock. Using the four necessary conditions for deadlock, explain why this implementation meets all of those conditions.

(b) $\boxed{\text{8 points}}$ **Deadlock Avoidance.** To prevent your snakes from deadlocking, you consider using the *deadlock avoidance* technique that you learned about in 15-410.

i. $\boxed{\text{2 points}}$ Recall that with deadlock avoidance, each process (in this case, a particular snake) needs to pre-declare its maximal resource usage to the resource manager before it can begin to request resources.

α) What would the specific resource pre-declaration be for snake on the right-hand side of row 13 (i.e. the one whose head is at coordinate (5,13))? (Hint: you may want to give your answer as a set of coordinates.)

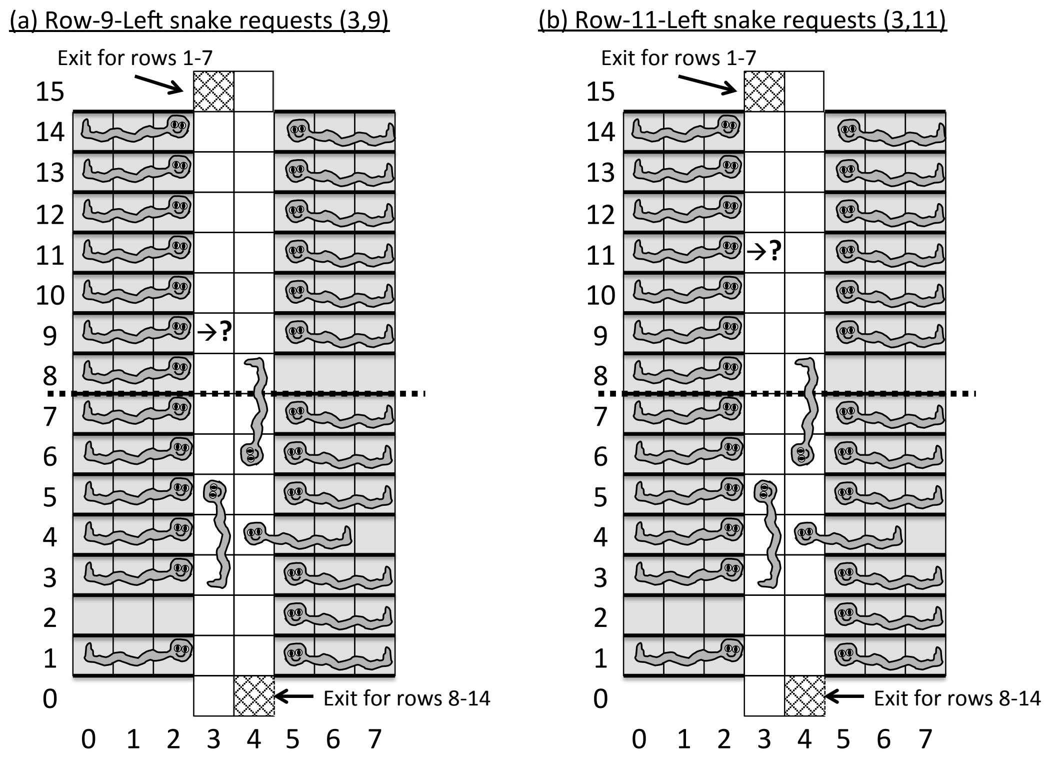
β) What would the specific resource pre-declaration be for snake on the right-hand side of row 7 (i.e. the one whose head is at coordinate (5,7))?

ii. $\boxed{\text{6 points}}$ We will now look at specific scenarios regarding resource requests and the current state of the system, and you will be asked to explain whether the resource manager would accept or reject the given resource request. In both of these scenarios, the current state of the resources is identical, and it is illustrated in the figure on the next page: three snakes have already started to move, and have been granted the following resources:

- **Row-2-Left ("2L") snake:** head = (3,5), middle = (3,4), tail = (3,3)
- **Row-8-Right ("8R") snake:** head = (4,6), middle = (4,7), tail = (4,8)
- **Row-4-Right ("4R") snake:** head = (4,4), middle = (5,4), tail = (6,4)

α) The system is currently in a safe state. What is a *safe sequence* that proves this? (You can refer to the specific snakes using the "2L", "8R", "4R" notation shown above.)

The figure below illustrates two separate scenarios for resource requests.



β) **Scenario (a):** The snake that was initially placed on the lefthand side of row 9 ("9L") requests coordinate (3,9) (illustrated on the lefthand side of the figure above). If the given resource request was granted by the resource manager, would the system still be in a *safe state*? If your answer is "yes", then prove it by showing a *safe sequence*. If your answer is "no", then explain exactly why there is no safe sequence. Also, if your answer is "no", then answer the following question: would deadlock necessarily occur if the request was granted?

γ) **Scenario (b):** Alternatively (i.e. instead of the request described above), the snake that was initially placed on the lefthand side of row 11 ("11L") requests coordinate (3,11) (illustrated on the righthand side of the figure on the previous page). If the given resource request was granted by the resource manager, would the system still be in a *safe state*? If your answer is "yes", then prove it by showing a *safe sequence.* If your answer is "no", then explain exactly why there is no safe sequence. Also, if your answer is "no", then answer the following question: would deadlock necessarily occur if the request was granted?

(c) $\boxed{\text{5 points}}$ **Deadlock Prevention.** If the `calculate_delta_xy()` procedure is modified to use a different algorithm for controlling the motion of the snakes, can *deadlock prevention* be successfully applied in this case? Note that in a modified version of `calculate_delta_xy()`, (i) the values of `delta_x` and `delta_y` must still be either 0, 1, or -1; (ii) the snakes can advance their heads only within the two aisle columns (i.e. columns 3 and 4); and (iii) all other constraints described on the previous pages regarding snake motion and positioning must be preserved. Assume that the `move_snake_one_step()` should still perform at least the same steps, and that you are free to add additional sychronization (or other state and computation) 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.

4. $\boxed{\text{20 points}}$ Countdown latches.

A CountDownLatch is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

Each CountDownLatch is initialized with a specified *count*. The `await()` method blocks the calling thread until the latch's count value reaches zero due to `count` invocations of the `countDown()` by other threads. Once the `count` reaches zero, all threads waiting in `await()` are released, and any subsequent invocations of `await()` return immediately. Once a particular CountDownLatch has counted down to zero, it is "used up": there is no way to increase or reset the `count` value.

A CountDownLatch is a versatile synchronization tool and can be used for a variety of purposes. A CountDownLatch initialized with a count of 1 serves as a simple on/off latch, or gate: all threads invoking `await()` wait at the gate until it is opened by a thread invoking `countDown()`. A CountDownLatch initialized to some positive value N can be used to make one thread wait until N threads have completed some action, or some action has been completed N times.

A useful property of a CountDownLatch is that it doesn't require that threads calling `countDown()` wait for the count to reach zero before proceeding – it simply prevents any thread from proceeding past an `await()` until all threads could pass.

The CountDownLatch we will ask you to implement for this exam differs from a more-typical CountDownLatch such as `java.util.concurrent.CountDownLatch` by being "abortable." That is, if one thread decides that, for some application-specific reason, it will not be the case that eventually enough invocations of `countDown()` will occur, that thread can invoke the CDL's `abort()` method. After the abort, an aborted CountDownLatch behaves like one that has had its count reach zero through natural means.

Note that once a thread invokes a CDL's `abort()` method and the `abort()` method returns, the caller is entitled to destroy the CDL "right away"—in other words, as soon as the logic of the application ensures that more threads won't invoke the `await()` method.

Your task is to implement CountDownLatch with the following interface:

- `int cdl_init(cdlatch_t *latch, int n)` — initialize a CountDownLatch with non-negative count n.
- `void cdl_await(cdlatch_t *latch)` — if the CountDownLatch's count is non-zero, the calling thread blocks on the CountDownLatch until the count reaches zero and `cdl_await()` returns "promptly" thereafter. If the count was already zero when `cdl_await()` is called, the calling thread should return "very promptly."
- `void cdl_countdown(cdlatch_t *latch)` — Subtracts one from" latch's `count` value and releases waiting threads if the count is zero.
- `void cdl_abort(cdlatch_t *latch)` — All threads waiting on the CountDownLatch return "promptly." After `cdl_abort()` is called, threads invoking `cdl_await()` will return "very promptly."
- `void cdl_destroy(cdlatch_t *latch)` — Destroys the CountDownLatch. It is illegal to attempt to destroy it before the count has reached zero. After `cdl_destroy()` has begun executing, it is illegal for any thread to invoke `cdl_await()`, `cdl_countdown()`, or `cdl_abort()` on the corresponding CountDownLatch.

A minimal usage example follows.

```
#define NUM_THREADS 4
cdlatch_t start_latch, done_latch;

void* worker(void* args) {
    cdlatch_await(&start_latch);
    printf("Thread %d got signaled by the start latch\n", thr_getid());
    printf("Thread %d starting to countdown on the done latch\n", thr_getid());
    cdlatch_countdown(&done_latch);
    return NULL;
}

int main() {
    int i;

    thr_init(4096);

    cdlatch_init(&start_latch, 1);
    cdlatch_init(&done_latch, NUM_THREADS);

    for (i = 0; i < NUM_THREADS; i++) {
        thr_create(worker, NULL);
    }

    printf("Thread %d starting to countdown on start-latch\n", thr_getid());
    cdlatch_countdown(&start_latch);
    cdlatch_await(&done_latch);
    printf("Thread %d signaled by the done latch\n", thr_getid());
    thr_exit(0);
    return 0;
}
```

The remainder of this page is intentionally blank.

Assumptions:

1. You may use regular Project 2 thread-library primitives: mutexes, condition variables, semaphores, reader/writer locks, etc.
2. You may assume that callers of your CountDownLatch routines will obey the rules—for example, nobody will try to destroy a latch while threads might be blocked on it. **But you must be careful that you obey the rules as well!**
3. 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`).
4. 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.
5. You may not use assembly code, inline or otherwise.
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 **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.
8. 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).

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) $\boxed{\text{5 points}}$ Please declare your `cdlatch_t` here. If you need one (or more) auxilary structures, you may declare it/them here as well.

```
typedef struct {








} cdlatch_t;
```

(b) $\boxed{\text{15 points}}$ Now please implement `int cdl_init()`, `void cdl_await()`, `void cdl_countdown()`, `void cdl_abort()`, and `void cdl_destroy()`.

. . . space for CountDownLatch implementation . . .

. . . space for CountDownLatch implementation . . .

5. $\boxed{\text{10 points}}$ `Nuts & Bolts`. Below please find an abbreviated description of the `swexn()` ("software exception") facility of Pebbles kernels.

```
typedef struct ureg_t {
    unsigned int cause;
    unsigned int cr2;   /* Or else zero. */

    unsigned int ds;
    ...some registers omitted...
    unsigned int edi;
    unsigned int esi;
    ...some registers omitted...
    unsigned int error_code;
    unsigned int eip;
    unsigned int cs;
    unsigned int eflags;
    unsigned int esp;
    unsigned int ss;
} ureg_t;
```

- `typedef struct ureg_t { // ...see figure... } ureg_t`
  `typedef void (*swexn_handler_t)(void *arg, ureg_t *ureg)`
  `int swexn(void *esp3, swexn_handler_t eip, void *arg, ureg_t *newureg)` -

  If `esp3` and/or `eip` are zero, de-register an exception handler if one is currently registered.

  If both `esp3` and `eip` are non-zero, attempt to register a software exception handler. The parameter `esp3` specifies an exception stack; it points to an address one word higher than the first address that the kernel should use to push values onto the exception stack. The parameter `eip` points to the first instruction of the handler function.

  Whether or not a handler is being registered or de-registered, if `newureg` is non-zero, the kernel is requested to adopt the specified register values (including `%EIP`!) before the `swexn()` system call returns to user space (see `syscall.h` for a description of the `ureg_t` structure and the register values it contains).

  [...]

  If the invocation is invalid [...] an error code less than zero is returned (and no change is made to handler registration or register values). The kernel **MUST** ensure that it does not allow a thread to assume register values which are unsafe ... The kernel should also reject `eip` and `esp3` values which are "clearly wrong" at the time of invocation [...]

  It is not an error to register a new handler if one was previously registered or to de-register a handler when one was not registered.

  When a software exception handler function (`swexn_handler_t`) begins running, it will be invoked via a stack frame which specifies two parameters, an opaque `void*` which was specified when the handler was registered, and a pointer to a `ureg` area. The return address of the function will be some invalid address. Before the first instruction of the handler is run, the handler is automatically de-registered by the kernel.

  [...]

While working on P2 you might have been frustrated by, or curious about, one feature of `swexn()`'s behavior. In particular, the kernel arranges that, when a user-specified software exception handler function begins running, it has an invalid return address—in other words, if the handler function returns, the thread will receive some sort of exception. This is very different from the behavior of the Unix `signal()` facility: in Unix, if a signal handler returns, the program generally resumes execution at the exact place where it was when the signal handler was invoked.

(a) $\boxed{\text{6 points}}$ Since the kernel is all-powerful, obviously it could choose to invoke each `swexn()` handler with a *valid* address. In the Pebbles environment, *every* `swexn()`-handler invocation is due to a problem encountered when running some particular instruction, so the kernel could very easily set things up so that the handler's return address is the address of the problematic instruction, so if the handler returned the problematic instruction would be re-executed. At first, that seems like an obvious improvement over the current behavior. Explain why that would be a bad idea, or not a useful change to make. If you completely understand the problem, it should be possible for you to clearly and convincingly demonstrate that in one or two sentences.

(b) $\boxed{\text{4 points}}$ Explain how the problem you identified above is handled by (or in) the standard Pebbles environment. You may wish to step through an example of how `swexn()` is used in practice.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.