Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Fall 2019

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.	10		
	70		

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) $\boxed{\text{6 points}}$ When designing a body of code, at times one finds oneself thinking, "I wonder if I should use Approach A or Approach B?" According to the 15-410 design orthodoxy, you should follow a specific process to resolve your question. Please document a design decision you made while working on your Project 2 thread library. For the purposes of this question we will be scoring based on your *description* of the decision, not whether we agree or disagree with what you chose. Begin with a brief description of the problem (one to three sentences) and then show us your design decision—answers that correctly use the 15-410 approved data structure will receive higher scores.

You may use this page as extra space for the P2-design question if you wish.

(b) $\boxed{\text{4 points}}$ Register dump.

Below is a register dump produced by the "Pathos" P2 reference kernel when it decided to kill a user-space thread. Your job is to carefully consider the register dump and:

1. Determine which "wrong register value(s)" caused the thread to run an instruction which resulted in a fatal exception. You should say why/how the wrong value led to an exception, i.e., merely claiming a register has a "wrong" value will not receive full credit.
2. Briefly state the most plausible way you think that register could have taken on that value (i.e., try to describe a bug which could have this effect).
3. Then write a *small* piece of code which would plausibly cause the thread to die in the fashion indicated by the register dump. *This code does not need to implement exactly the set of steps that you identified as "most plausible" above, or result in the same register values; you should aim to achieve "basically the same effect."* Most answers will probably be in assembly language, but C is acceptable as well. Your code should assume execution begins in `main()`, which has been passed the typical two parameters in the typical fashion.

Please be sure that your description of the fatality and the code, taken together, clearly support your diagnosis.

```
Registers:
eax: 0xffffd000, ebx: 0x0100707c, ecx: 0x00000001,
edx: 0x01000154, edi: 0x00000000, esi: 0x00000000,
ebp: 0xffffefa4, esp: 0xffffefa8, eip: 0xffffefc8,
ss:     0x002b,  cs:     0x0023,  ds:     0x002b,
es:     0x002b,  fs:     0x002b,  gs:     0x002b,
eflags: 0x00000246
```

You may use this page for the register-dump question.

2. $\boxed{\text{15 points}}$ As many of you noticed in Homework 1, Professor Eckhardt's "elegant proposal" for a three-thread critical-section algorithm was a disaster. One issue arose from the fact that, when Professor Eckhardt was cribbing from Dr. Peterson's excellent work, he left out the `turn` variable. That omission has been remedied, resulting in the following proposal, also elegant.

   Peterson's solution was presented in "standard form," i.e., when thread 0 is running the code, $i == 0$ and $j == 1$; when thread 1 is running the code, $i == 1$ and $j == 0$, so i means "me" and j means "the other thread." While this may seem odd, it is a tradition, and you might someday be expected to answer an exam question phrased in that form. The code below, titled "Trinary II," uses the same thread-local variables as the "Trinary" protocol in the homework problem. Metaphorically, as is the case in the Dining Philosophers problem, the three threads are seated around a table, so each has a left neighbor, `l`, who is "-1" modulo 3, and a right neighbor, `r`, who is "+1" modulo 3, assigned as follows.

Thread	`l`	`i`	`r`
T0	2	0	1
T1	0	1	2
T2	1	2	0

```
volatile int turn = 0;         // privileges one thread
volatile int want[3] = {0, };  // per-thread flags (initially all zero)

void lock(void) {
    turn = r; // rotate right
    want[i] = 1;
    while ((turn != i) && (want[l] || want[r]))
      continue;
}

void unlock(void) {
    want[i] = 0;
}
```

   (a) $\boxed{\text{10 points}}$ 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 present a trace which demonstrates your claim. Use the format presented in class as shown below. You may use more or fewer columns or lines in your trace. You may introduce temporary variables or other obvious notation as necessary to improve the clarity of your answer. If you wish, you may abbreviate `want[]` as `w[]` and `turn` as `t`.

   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.

Execution Trace

Time	T 0	T 1	T 2
0	w[3]==0		
1	w[2]=1		
2			

Use this page for the critical-section protocol question.

(b) $\boxed{\text{5 points}}$ Show, *with a clear and convincing trace*, that a *different* critical-section algorithm requirement also does not hold. **Please note that it may make sense for you to work on other exam questions before finding a second requirement failure in this code.**

Andrew ID: ______________

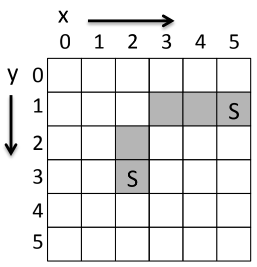
You may use this page as extra space for the critical-section protocol question if you wish.

3. $\boxed{\text{15 points}}$ Battleship Deadlock.

Battleship is a classic two-player board game. Each player has their own `BOARD_SIZE x BOARD_SIZE` board. To begin the game, each player places a number of ships on their board at locations of their choosing. For this problem, we will assume that one player is a human and the other player is an AI that you and your P2 partner are writing in your copious spare time. In order to speed things up, you have decided that the AI will use multiple threads to do its initial placement of ships, and that each thread will use the following `place_ship()` interface:

```
bool place_ship(int length, int start_x, int start_y, ship_direction_t direction);
```

- The size of the ship is specified by the `length` parameter, where length is the number of board squares.
- 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, the board below has two ships, one of `length` 2 starting at (2,3) with a `direction` of `UP`, and the other of `length` 3 starting at (5,1) with a `direction` of `LEFT`.



- 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 ships might overlap each other, which would be a problem. For example, in our example board, placing a ship of `length` 2 at (4,0) with a `direction` of `DOWN` would overlap another ship at (4,1) and should not be allowed. 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()`.

Suppose your partner 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.
- You may assume that the fact that the `place_remainder_of_ship()` routine is recursive does not create a problem. The depth of recursion is bounded by the maximum ship length. Since this is a constant, you can give each thread a stack that is large enough to perform the recursion safely.

```
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) $\boxed{\text{6 points}}$ 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) $\boxed{\text{2 points}}$ Clearly explain how the trace you showed in part (a) has all of the necessary deadlock ingredients.

(c) $\boxed{\text{7 points}}$ *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 cannot 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).

You may use this page as extra space for your `place_ship()` restructuring if you wish.

4. $\boxed{\text{20 points}}$ Boolean Cyclic Barriers

In lecture we talked about two fundamental operations in concurrent programming: brief mutual exclusion for atomic sequences (provided in Project 2 by mutexes) and long-term voluntary descheduling (provided by condition variables). As you know, these can be combined to produce higher-level objects such as semaphores or readers/writers locks, which you implemented in P2. But concurrent programs may use a variety of other synchronization objects.

Some parallel programs are organized so that computation occurs in stages or phases: each thread examines some or all of the stage-0 data and computes a stage-1 item, then each thread examines some or all of the stage-1 data and computes a stage-2 item, etc. The end of each stage and the start of the next are synchronized via an object called a "barrier." The operation of the barrier is as follows:

1. First, the barrier is initialized to expect t threads;
2. As each thread finishes its computation for a particular phase P, it arrives at the barrier and is blocked.
3. The arrival of the last thread done with phase P causes all threads done with phase P to "pass through" the barrier and begin working on phase $P + 1$; the barrier prepares itself for the next arrival of each of the t threads when they eventually complete phase $P + 1$.

You will be implementing a specific kind of barrier called a "boolean cyclic barrier." The operations are as follows:

- `int bcb_init(bcb_t *bp, int n);` — initialize a barrier with non-negative count `n`. Returns a negative error code if initialization fails, otherwise 0 on success.
- `bool bcb_arrive(bcb_t *bp, bool success);` — waits until `n` threads have invoked `bcb_arrive` on the barrier. Once the last thread in the current phase has called `bcb_arrive`, all threads blocked in the current phase should return "promptly." The return value is `true` if all threads passed in `success == true`, and otherwise `false`.
- `void bcb_destroy(bcb_t *bp);` — destroys the barrier. It is illegal to attempt to destroy it while any threads are in `bcb_arrive`. After `bcb_destroy` has begun executing, it is illegal for any thread to invoke `bcb_arrive` on the corresponding barrier.

Here is some code demonstrating the usage of the object. In this program a barrier stops being used at the first "failure" event, but that is a property of the program, not a rule for using these barriers.

The remainder of this page is intentionally blank.

```
#define NTHR 100
bcb_t barrier;

int A[NTHR];
int B[NTHR];

/* compute_item(int slot) somehow computes a value based
 * on some or all of A[] and B[] and modifies either
 * A[slot] or B[slot] depending on which phase it is in.
 * If it believes it has found an acceptable answer, it
 * prints it out somehow and returns false; otherwise, it
 * returns true.  The program as a whole may print multiple
 * answers before it exits, which is fine.
 */
bool compute_slot(int slot);

void *worker(void *arg)
{
  int idx = (int)arg;
  bool keepgoing;

  do {
    keepgoing = compute_slot(idx);
  } while (!bcb_arrive(&barrier, keepgoing));

  return NULL;
}

int main(int argc, char *argv[])
{
  int t, tids[NTHR];

  thr_init(4096); // exam: cannot fail
  bcb_init(&barrier, NTHR); // exam: cannot fail

  for (t = 0; t < NTHR; ++t)
    tids[t] = thr_create(worker, (void *)t); // exam: cannot fail

  for (t = 0; t < NTHR; ++t)
    thr_join(tids[t], NULL); // exam: cannot fail

  bcb_destroy(&barrier);
  thr_exit(0);
}
```

Assumptions:

1. You may use regular Project 2 thread-library primitives: mutexes, condition variables, semaphores, readers/writer locks, etc.
2. You may assume that callers of your routines will obey the rules. **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). You may also assume that condition variables are strictly FIFO if you wish.

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) $\boxed{\text{6 points}}$ Please declare your `struct bcb` here and write the initialization function `int bcb_init(bcb_t *bp, int n)`. If you need one (or more) auxilary structures, you may declare it/them here as well.

```
typedef struct bcb {










} bcb_t;
```

You may use this page as extra space for your barrier synchronization initialization if you wish.

(b) $\boxed{\text{14 points}}$ Now please write your code for `bool bcb_arrive(bcb_t *bp, bool success)` and `void bcb_destroy(bcb_t *bp)`.

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

Andrew ID: ______________

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

5. 10 points Nuts & Bolts. Consider the trivial Project 1 "game" kernel below.

```
static void ignoreticks(unsigned int numTicks)
{
    (void) numTicks;
    return;
}

typedef struct player {
  char *name;
  int role;
  int x, y;
  int score;
} player_t, *player_p;

static player_t protagonist;

static void output(player_p p)
{
    MAGIC_BREAK; // Assume: does not affect stack
    printf("%s @ (%d,%d): %d\n", p->name, p->x, p->y, p->score);
}

static void test(char *s)
{
    player_t initial;

    initial.name = s;
    initial.role = 0;
    initial.x = initial.y = 0;
    initial.score = 0;

    output(&initial);
    protagonist = initial;
}

int kernel_main(mbinfo_t *mbinfo, int argc, char **argv, char **envp)
{
    // placate compiler
    (void)mbinfo; (void)argc; (void)argv; (void)envp;

    handler_install(ignoreticks);

    test("Hiro");

    return 0; // Not a good idea, but we don't plan to get here.
}
```

Draw a picture of the stack as it will look when the `MAGIC_BREAK` statement causes Simics (if present) to drop into the debugger. Label each memory location you are filling with its address (you may "draw" memory in bytes or words as you see fit). You may choose any "plausible" addresses you wish, as long as they make sense. Assume a 32-bit machine with any plausible byte order. You need not show the complete frame of the invoker of `kernel_main()` or any part of any earlier stack frame. If you wish, you may assume that the invoker of `kernel_main()` is named `mb_entry()`.

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."

Ureg Cheat-Sheet

```
#define SWEXN_CAUSE_DIVIDE       0x00  /* Very clever, Intel */
#define SWEXN_CAUSE_DEBUG        0x01
#define SWEXN_CAUSE_BREAKPOINT   0x03
#define SWEXN_CAUSE_OVERFLOW     0x04
#define SWEXN_CAUSE_BOUNDCHECK   0x05
#define SWEXN_CAUSE_OPCODE       0x06  /* SIGILL */
#define SWEXN_CAUSE_NOFPU        0x07  /* FPU missing/disabled/busy */
#define SWEXN_CAUSE_SEGFAULT     0x0B  /* segment not present */
#define SWEXN_CAUSE_STACKFAULT   0x0C  /* ouch */
#define SWEXN_CAUSE_PROTFAULT    0x0D  /* aka GPF */
#define SWEXN_CAUSE_PAGEFAULT    0x0E  /* cr2 is valid! */
#define SWEXN_CAUSE_FPUFAULT     0x10  /* old x87 FPU is angry */
#define SWEXN_CAUSE_ALIGNFAULT   0x11
#define SWEXN_CAUSE_SIMDFAULT    0x13  /* SSE/SSE2 FPU is angry */

#ifndef ASSEMBLER

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

    unsigned int ds;
    unsigned int es;
    unsigned int fs;
    unsigned int gs;

    unsigned int edi;
    unsigned int esi;
    unsigned int ebp;
    unsigned int zero;  /* Dummy %esp, set to zero */
    unsigned int ebx;
    unsigned int edx;
    unsigned int ecx;
    unsigned int eax;

    unsigned int error_code;
    unsigned int eip;
    unsigned int cs;
    unsigned int eflags;
    unsigned int esp;
    unsigned int ss;
} ureg_t;

#endif /* ASSEMBLER */
```

Andrew ID: ____________

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.