Computer Science 15-410: Operating Systems
Mid-Term Exam (B), Spring 2009

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: _____

I have not received advance information on the content of this 15-410 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. [10 points] Short answer.

Give a three-to-five sentence definition of each of the following terms *as it applies to this course*. Your goal is to make it clear to your grader that you understand the concept and can apply it when necessary.

(a) [5 points] Blocked

(b) [5 points] Asynchronous thread cancellation

2. [15 points] Interrupts

Consider interrupt handlers in the Project 1 run-time environment. Shown below is the wrapper code for two handlers (they might be for the keyboard and timer, or they might be for other handlers, such as a mouse or disk drive). Assume the relevant IDT entries are configured as trap gates.

```
.text
.globl i1handler, i2handler

.globl g1wrapper
g1wrapper:
  pusha
  call i1handler
  popa
  iret

.globl g2wrapper
g2wrapper:
  pusha
  call i2handler
  popa
  iret
```

When you and your Project 3 partner compare notes on how you implemented interrupt handler wrappers in Project 1, controversy breaks out. Your partner wants to know what happens if the processor is in the middle of executing the PUSHA instruction in `g1wrapper` when the interrupt handled by `g2wrapper` is asserted. In detail, the concern is that `g1wrapper`'s PUSHA may have pushed some, but not all, of the registers it's supposed to when suddenly control could transfer to `g2wrapper`'s PUSHA.

(a) [6 points] Given the description above, explain either why this condition *cannot* occur, or why it *can occur but is not a threat.*

Meanwhile, your partner expresses a second concern. Since PUSHA pushes quite a few registers on the stack, your partner is worried that the pushing could cross a page boundary and cause a page fault. Your partner proposes the following fix, to be applied to all interrupt handler wrappers (only two are shown).

```
.text
.globl i1handler, i2handler

.globl g1wrapper
g1wrapper:
  cli            # disable interrupts
  pusha
  call i1handler
  popa
  sti            # enable interrupts
  iret

.globl g2wrapper
g2wrapper:
  cli            # disable interrupts
  pusha
  call i2handler
  popa
  sti            # enable interrupts
  iret
```

(b) [8 points] Is this change necessary and sufficient to solve the problem as described? If yes, explain how. If no, explain why this change does *not* help and also explain any necessary steps which must be taken.

3. [20 points] Semaphore problem

Having heard that one of the 410 course staff very much enjoys semaphores, your partner has implemented a "small extension" to the standard semaphore design which increases the similarity between semaphores and condition variables. In particular, your partner's code tracks how many threads are waiting on a semaphore and provides a primitive, `sem_broadcast()`, which wakes up all threads waiting on a semaphore.

```
typedef struct {
    int count;
    int waiters;
    mutex_t count_lock;
    mutex_t waiters_lock;
    cond_t cv;
} sem_t;

int sem_init(sem_t *sem, int count); /* code omitted for exam purposes */
int sem_destroy(sem_t *sem);         /* code omitted for exam purposes */

int sem_wait(sem_t *sem) {
    mutex_lock(&sem->count_lock);
    if (sem->count > 0) {
        sem->count--;
        mutex_unlock(&sem->count_lock);
        return 0;
    } else {
        mutex_lock(&sem->waiters_lock);
        sem->waiters++;
        mutex_unlock(&sem->waiters_lock);
    }
    cond_wait(&sem->cv, &sem->count_lock);
    sem->count--;
    mutex_unlock(&sem->count_lock);
    mutex_lock(&sem->waiters_lock);
    sem->waiters--;
    mutex_unlock(&sem->waiters_lock);
    return 0;
}

int sem_signal(sem_t *sem) {
    mutex_lock(&sem->count_lock);
    sem->count++;
    cond_signal(&sem->cv);
    mutex_unlock(&sem->count_lock);
    return 0;
}
```

```
int sem_broadcast(sem_t *sem) {
    int i, w;
    // lock: we want only *existing* waiters
    mutex_lock(&sem->waiters_lock);
    w = sem->waiters;
    for (i = 0; i < w; i++)
        sem_signal(sem);
    mutex_unlock(&sem->waiters_lock);
    return 0;
}
```

There are (at least) two synchronization problems found in the code presented above. To receive full credit, identify two *different* problems (rather than two instances of the same mistake). Do not present more than two. The problems are found in the code presented to you (in other words, we are not looking for claims that `sem_init()` might be implemented incorrectly). You should assume that invocations of thread-library primitives (e.g., `mutex_lock()`) succeed rather than detecting inconsistency or otherwise failing. You may wish to refer to the "cheat sheets" at the end of the exam.

If you have correctly identified a sychronization problem, you will be able to *briefly and clearly* summarize it and show a *clear and compelling* execution trace using the tabular execution format from the lectures and homework assignment. Confusing descriptions and unclear execution traces will be read as evidence of incomplete understanding and will be graded as such. This means that it is to your advantage to *think your answers through before beginning to write.*

(a) [12 points] *Briefly and clearly* describe the first synchronization problem you have identified with the code above. Show a *clear and compelling* execution trace.

(b) [8 points] *Briefly and clearly* describe a second synchronization problem with the code above, which should be "a different kind of problem." Show a *clear and compelling* execution trace.

4. [20 points] Rendezvous

Concurrent programs use a variety of synchronization objects beyond the standard mutex and condition variable. For Project 2 you implemented semaphores and readers-writers locks, but applications use other synchronization objects as well.

One such object is the "rendezvous," which combines synchronization with data communication. Two threads wishing to synchronize "arrive" at the same rendezvous, each with a data item they wish to convey to the other thread. The first thread to arrive suspends its execution until the second arrives; when the second arrives, the two values are exchanged; the "arrive" operation then returns to each thread indicating the value which was provided by the other thread.

Here is some toy code demonstrating the usage of a rendezvous.

```
rendezvous_t r;

void *worker(void *data)
{
    int initial, other;

    initial = (int)data;                // get our value from main()

    other = rend_arrive(&r, initial);   // now, swap values with someone else
    assert(initial == !other);          // assumes main() used 0 and 1!

    other = rend_arrive(&r, other);     // now, just swap them back
    assert(initial == other);

    return NULL;
}

int main(void)
{
    int thr1, thr2;

    thr_init(PAGE_SIZE);
    rend_init(&r);

    thr1 = thr_create(worker, (void *) 0);
    thr2 = thr_create(worker, (void *) 1);

    thr_join(thr1, NULL); thr_join(thr2, NULL);

    thr_exit(NULL);

    return 0;
}
```

The Plan 9 operating system exports rendezvous as a kernel primitive serving essentially the same purpose as `cas2i_runflag()` does in Pebbles. You will implement a userland variant of this mechanism on top of the other synchronization objects provided by the thread library.

You will provide us with both a structure definition for your rendezvous and the code for two functions

- `typedef struct rendezvous { ... } rendezvous_t;`
- `void rend_init(rendezvous_t *rp);`
- `int rend_arrive(rendezvous_t *rp, int value);`

We will not worry about `rend_destroy()`. Your solution can use Project 2 thread library primitives. 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. You may not use assembly code, inline or otherwise. For the purposes of the exam you should assume an error-free environment (memory allocation will always succeed; thread-library primitives will not detect internal inconsistencies or otherwise "fail," etc.). You may wish to refer to the "cheat sheets" at the end of the exam.

The remainder of this page is intentionally blank.

(a) [0 points] The best way to get a correct solution is to analyze the problem and form an invariant or other conceptual description which will let you check your solution. A good way to get in trouble is to write down code for one case, and then discover another case... We suggest that you present a brief argument that your solution addresses all relevant hazards.

Answering this part of the question is optional but *highly recommended*. If your grader cannot understand why your code is correct, you will not receive full credit.

(b) [5 points] Please declare your `struct rendezvous` here. Also write a function `void rend_init(rendezvous_t *rp)` to initialize a rendezvous.

```
typedef struct rendezvous {

} rendezvous_t;

void rend_init(rendezvous_t *rp)
{

}
```

(c) [15 points] Now please write `rend_arrive(rendezvous_t *rp, int value)`.

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

5. [10 points] Process model.

Consider the following odd program.

```
const char rodata[] = "Can't touch this!";

int main(int argc, char *argv[])
{
  return (rodata[0]);
}
```

The "rodata" entity is "self-descriptive": it is named "rodata" and it exists in the rodata region (or rodata segment, or rodata section) of the program.

(a) [5 points] Please write a short C program which defines another "self-descriptive" entity. Also state one property of this region which differentiates it from other regions.

(b) [5 points] Please write a short C program which defines yet another "self-descriptive" entity. Also state one property of this region which differentiates it from other regions.

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 cas2i_runflag(int tid, int *oldp, int ev1, int nv1, int ev2, int nv2);
int get_ticks();
int sleep(int ticks); /* 100 ticks/sec */

/* 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 ls(int size, char *buf);

/* "Special" */
void misbehave(int mode);
```

Thread-Library Cheat-Sheet

```
int mutex_init( mutex_t *mp );
int mutex_destroy( mutex_t *mp );
int mutex_lock( mutex_t *mp );
int mutex_unlock( mutex_t *mp );

int cond_init( cond_t *cv );
int cond_destroy( cond_t *cv );
int cond_wait( cond_t *cv, mutex_t *mp );
int cond_signal( cond_t *cv );
int 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 );
int sem_wait( sem_t *sem );
int sem_signal( sem_t *sem );
int sem_destroy( sem_t *sem );

int rwlock_init( rwlock_t *rwlock );
int rwlock_lock( rwlock_t *rwlock, int type );
int rwlock_unlock( rwlock_t *rwlock );
int rwlock_destroy( rwlock_t *rwlock );
```

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.