Computer Science 15-410: Operating Systems
Mid-Term Exam (B), Spring 2008

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.	20		
3.	20		
4.	15		
5.	10		
	75		

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 one-paragraph explanation 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] Progress

(b) [5 points] User mode

2. [20 points] Trouble at the Warehouse

After completing 15-410, you and your partner have been hired by a small Pittsburgh fruit-juice distributor to automate their warehouse. Here is how the warehouse operates.

1. Big long-haul container trucks arrive from your suppliers. Each container truck contains either 1,000 cases of Odwalla juice or 1,000 cases of Jamba Juice. Generally each container truck is unloaded into the warehouse as soon as it arrives, though there are of course exceptions.
2. Your company operates a small fleet of delivery trucks. Each delivery truck arrives at the warehouse seeking some number of cases of Odwalla and some number of cases of Jamba (the total number of cases a delivery truck can hold is 50). Once it loads what it needs, it will leave the warehouse and deliver them. Like container trucks, delivery trucks usually access the warehouse right away, but sometimes must wait.
3. Container trucks park at one loading dock, and delivery trucks park at a different one, because their heights are different.
4. Juice must be moved into and out of the warehouse by the warehouse's one forklift (after you complete the exam you may wish to view the YouTube video on the AirTrax "Sidewinder").
5. To avoid a "denial-of-service attack" by one or the other juice manufacturer, the warehouse manager has imposed a constraint on container deliveries: if unloading a container would result in more than 80% of the warehouse being occupied by one manufacturer's product, the container must wait.
6. Because the warehouse is located in an industrial district with low land values, you may assume the neighborhood provides enough parking for your fleet of delivery trucks and as many container trucks as necessary.

Your partner was hired before you, and deployed the code depicted below. You may assume that all objects are properly initialized before use as part of system start-up.

```
/* Container loading dock */
mutex_t cd_m;    int cd_avail;     cond_t cd_released;

/* Delivery loading dock */
mutex_t dd_m;    int dd_avail;     cond_t dd_released;

/* Forklift --
 * locking simpler because both #contenders and hold
 * time are well bounded */
mutex_t fl_m;

/* Inventory management */
#define JTYPES 2            // 0=Odwalla, 1=Jamba
#define CONTAINER 1000      // cases per truck
#define CAPACITY 9000       // total cases per warehouse
int avail[JTYPES];
```

```
01 void
02 container_arrival(int type)
03 {
04   int ready;
05
06   mutex_lock(&cd_m);
07   ready = 0;
08   while (!ready) {
09     while (!cd_avail) {
10       cond_wait(&cd_released, &cd_m);
11     }
12     cd_avail = 0; // claim
13
14     // Must not overflow warehouse
15     int total = avail[0] + avail[1];
16     if ((total + CONTAINER > CAPACITY) ||
17       (avail[type] + CONTAINER > ((80*CAPACITY)/100))) {
18       // No room, let somebody else try.
19       cd_avail = 1;
20       cond_broadcast(&cd_released);
21       //  Retry when space freed up by delivery team
22       cond_wait(&dd_released, &cd_m);
23     } else {
24       ready = 1;
25     }
26   }
27   mutex_unlock(&cd_m);
28
29   // Use forklift to unload
30   mutex_lock(&fl_m);
31   operate_forklift();
32   avail[type] += CONTAINER;
33   mutex_unlock(&fl_m);
34
35   // Drive away... announce container dock is free
36   mutex_lock(&cd_m);
37   cd_avail = 1;
38   cond_broadcast(&cd_released);
39   mutex_unlock(&cd_m);
40 }
```

```
01 void
02 delivery_arrival(int demand[JTYPES])
03 {
04   int ready, t;
05
06   mutex_lock(&dd_m);
07   ready = 0;
08   while (!ready) {
09     while (!dd_avail) {
10       cond_wait(&dd_released, &dd_m);
11     }
12     dd_avail = 0; // claim
13
14     // Ensure required inventory
15     for (t = 0; t < JTYPES; ++t) {
16       if (avail[t] < demand[t]) {
17         // Insufficient stock for us.  Let other delivery trucks try.
18         dd_avail = 1;
19         cond_broadcast(&dd_released);
20         // Retry when container arrives.
21         cond_wait(&cd_released, &dd_m);
22         break;
23       } else {
24         ready = 1;
25       }
26     }
27   }
28   mutex_unlock(&dd_m);
29
30   // Use forklift to load
31   mutex_lock(&fl_m);
32   operate_forklift();
33   for (t = 0; t < JTYPES; ++t)
34     avail[t] -= demand[t];
35   mutex_unlock(&fl_m);
36
35   // Drive away... announce delivery dock is free
38   mutex_lock(&dd_m);
39   dd_avail = 1;
40   cond_broadcast(&dd_released);
41   mutex_unlock(&dd_m);
42 }
```

(a) [10 points] Something is very wrong with the way this code synchronizes the operation of the warehouse. Clearly explain what is wrong. You should probably provide an execution trace, in the format presented in class, showing the problem in action. Obvious abbreviations are ok. For example:

container(0)	deliver({10,15})
unlock(cd_m);	
	unlock(dd_m);

If you cannot find a synchronization correctness problem, you may obtain partial credit by describing *one* "interesting" synchronization correctness problem the code does *not* have and briefly arguing why your claim is true. Avoid answers of the form "Such-and-such looks wrong, but I'm not sure why," as these demonstrate understanding of the material poorly at best.

(b) [10 points] Provide corrected code for one of the truck-arrival functions. If possible, try to maintain or reduce, rather than increase, the complexity of the system.

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

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

3. [20 points] Dual-priority locking.

Different application architectures require different locking approaches. For example, later in the semester we will discuss how real-time scheduling increases the complexity of locking. This question is about a dramatically simpler situation. The application in question has threads of two different priorities, and some objects must be locked according to a priority-aware protocol. The key requirement is this: when one of these objects is unlocked, it must be acquired by a high-priority thread if any are waiting; otherwise it must be acquired by a low-priority thread if any are waiting.

Each time a thread acquires or releases one of these objects it will indicate whether it is a high-priority thread or a low-priority thread. Threads will not lie and will not change priorities.

You will provide us with both a structure definition for your dual-priority lock and the code for three functions.

- `void dlock_init(dlock_p dp)`
- `void dlock_acquire(dlock_p dp, int isHigh)`
- `void dlock_release(dlock_p dp, int isHigh)`

You need not provide us with code for `dlock_destroy()`. You are encouraged to use the Project 2 thread-library primitives. If necessary you may use other synchronization primitives, but you should strive to avoid this, as it may reduce your grade. 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.).

Note that you are *not* required to make the impossible guarantee that no high-priority thread ever waits for a low-priority thread. Furthermore, do not worry if your solution is *very slightly* imperfect in a way which is not avoidable in certain situations. In other words, the key requirement may be rephrased as follows: when an object is released, if any high-priority threads are waiting, at most one low-priority thread may obtain the object before a high-priority thread does, but the number of low-priority acquisitions in this situation should almost always be zero instead of one.

The remainder of this page is intentionally blank.

(a) [5 points] Please declare your `struct dlock` here. Also write a function `void dlock_init(dlock_p dp)` to initialize a dual-priority lock.

```
typedef struct dlock {

} *dlock_p;

void dlock_init(dlock_p dp)
{

}
```

(b) [15 points] Now please write `dlock_acquire()` and `dlock_release()`.

You may use this page as extra space for your dual-priority lock solution if you wish.

You may use this page as extra space for your dual-priority lock solution if you wish.

4. [15 points] "Dead rock"

This semester, 410 students have formed four rock bands. Each band has a (small) set of songs they have practiced; each song requires an eclectic set of instruments.

Naturally the students are familiar with Professor Dannenberg's prominence in the field of computer music and figure that instead of buying instruments they can borrow them from him. Roger's collection contains these instruments.

Accordion	Bagpipe	Cowbell	Drum
4	3	2	5

The bands, their songs, and the instruments needed to play each song are listed in the table below. Note that each instrument, each band name, and each song title can be represented unambiguously by a 1-letter or 1-digit abbreviation.

	Accordion	Bagpipe	Cowbell	Drum
Band 1: One Thread Yielding				
Entry of the Threads	1	0	1	1
Free Memory	0	0	1	3
Band 2: Two Cores				
Groove Gettid	2	0	0	2
Hit The Break Jack	1	1	0	1
Band 3: Three Spare Bits				
Justify My Yield	0	1	1	1
Kill Dash Nine	3	0	2	2
Lock Free Your Heart	1	3	0	2
Band 4: Forth				
Mutex Romance	1	1	0	1
Never Going to Halt	2	2	0	4

There was a "battle of the bands" (popularity competition) last weekend, with these four bands and no others.

Because Roger was delayed by bad weather, the 410 students broke into his lab and each band borrowed the instruments necessary to play its first song (E, G, J, and M, respectively). Roger knows that each band is stubborn and will demand to go on stage and be allowed to play all of their songs in some order before being willing to yield the stage or return any instruments to Roger's collection.

Roger is worried that the naive resource allocations made by the students plus their stubborn policy may have placed the system at risk of deadlock.

(a) [5 points] Is there a safe sequence? If so, list a sequence of bands in order. If not, explain why there is not.

At Roger's urging, the 410 students decide to adopt an improved instrument-management protocol. After finishing each song and deciding on the next, they calculate how many instruments of each type they have which they will not need for the next song, and how many they will need which they do not have. For example, Band A switching from song E to song F would need to release one accordion and acquire two drums.

After calculating the change in resource requirements, they proceed as follows:

1. Release unneeded Accordions
2. Release unneeded Bagpipes
3. Release unneeded Cowbells
4. Release unneeded Drums
5. Acquire additional Accordions
6. Acquire additional Bagpipes
7. Acquire additional Cowbells
8. Acquire additional Drums

Please note that each of the above steps is atomic, meaning "Acquire 2 drums" happens all at once (at a time when two or more drums are available).

Initially each band is playing no song, and may decide to "play no song" (i.e., take a break) at any time. As far as the management protocol is concerned, playing no song is equivalent to playing a song requiring no instruments.

(b) [5 points] Consider bands 2, 3, and 4, and their songs as listed in the table above. Assume there are three stages, so the bands can play simultaneously. Can this algorithm deadlock?

If so, list a sequence of songs leading to deadlock in the tabular format indicated below and draw a process/resource graph showing the deadlock.

Band	Song	Operation	Instrument	Count
1	F	Acquire	Cowbell	1

If not, list the four deadlock requirements and briefly indicate for each one whether the proposed management protocol has, or lacks, that ingredient.

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

(c) [5 points] Now consider this variant on the protocol of Part B: after every song, each group takes a break (thus releasing all instruments) and then acquires instruments for their next song as described in the Part B protocol. Given all four bands, can this algorithm deadlock? Justify your answer.

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

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

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

player_t protagonist;

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);
}

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()
{
    lmm_remove_free( &malloc_lmm, (void*)USER_MEM_START, -8 - USER_MEM_START );
    lmm_remove_free( &malloc_lmm, (void*)0, 0x100000 );

    handler_install(ignoreticks);
    pic_init( BASE_IRQ_MASTER_BASE, BASE_IRQ_SLAVE_BASE );

    test("Hiro");

    while (1)
        continue;

    return 0;
}
```

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.

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 );
```