Computer Science 15-410: Operating Systems
Mid-Term Exam (B), Fall 2007

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.	**15**		
	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 conflict 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.

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

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] Write pipe (also known as "write buffer")

(b) [5 points] Interrupt acknowledge

2. [15 points] Monitors

One feature of monitors is that the compiler automatically adds most necessary synchronization code in a boring, mechanical way to avoid programmer mistakes. However, it is arguably possible to obtain much of the same benefit even in C without automatic compiler support by coding according to a "monitor pattern." In this approach, C macros inserted in standard places will generate the synchronization code in a regular way. As long as the programmer uses the macros correctly, the result should be correct code (at least as far as synchronization is concerned!).

Here are the basic macros for monitor setup, entry, and exit.

Macro	Purpose
`M_DECL()`	appears in monitor source file before any declarations or code
`M_INIT()`	must be called before the monitor is used
`M_ENTER`	programmer must insert this at every monitor entry point
`M_RET`	programmer must use this instead of "return;"
`M_RETURN(t,v)`	`t` is the C type of value `v`; programmer must use this instead of "return(v);"

These macros are used if condition variables are necessary.

`MC_DECL(name)`	declares a monitor condition variable
`MC_INIT(name)`	initializes a declared monitor condition variable
`MC_WAIT(name)`	waits for a monitor condition variable to be signalled
`MC_NOTIFY(name)`	signals a monitor condition variable

The code example below will further explain the meanings and usages of the macros we will consider.

```
/* Monitor for doing work in background:
 * Drop off an item for processing, pick it up
 * when you need it (hopefully it will be ready).
 *
 * Usage model:
 * 1. work_setup() MUST be called before any other function
 * 2. call work_dropoff(item_pointer);
 * 3. later call work_pickup(item_pointer)
 * 4. repeat as desired.
 */

#define TRUE 1
#define FALSE 0

M_DECL();
MC_DECL(activity);

item_t in_box, out_box;
int in_box_empty = TRUE, out_box_full = FALSE;

/* provide some work to be done in background */
void work_dropoff(item_t item)
{
    M_ENTER;

    while (!in_box_empty) MC_WAIT(activity);
    in_box = item;
    in_box_empty = FALSE;
    MC_NOTIFY(activity);

    M_RET;
}

/* get the results as soon as they are ready */
void work_pickup(item_t *item)
{
    M_ENTER;

    while (!out_box_full) MC_WAIT(activity);
    *item = out_box;
    out_box_full = FALSE;
    MC_NOTIFY(activity);

    M_RET;
}
```

```
/* worker() is designed to be run by a background
 * thread.  Because this is an exam, we won't worry
 * about how the worker thread is shut down.
 * worker() calls the external function process_an_item(),
 * which does heavy computation, and then delivers the
 * result to the out_box for pickup.
 */
static void
worker()
{
    M_ENTER;
    while (TRUE) {
        while (in_box_empty) MC_WAIT(activity);
        while (out_box_full) MC_WAIT(activity);

        out_box = process_an_item(in_box);

        in_box_empty = TRUE;
        out_box_full = TRUE;
        MC_NOTIFY(activity);
    }
    M_RET;
}
```

(a) [5 points] As you carefully read the listing, you realize that the code for the `work_setup()` function was unaccountably left out. Please fill in the missing code.

```
/* initialize our work dropoff/pickup monitor --
 * this must be called before any other calls to this monitor
 */
void work_setup(void)
{









}
```

(b) [2 points] Please show the code for M_DECL() and M_INIT(). Don't worry too much about the exact macro syntax (multi-statement macros in C are not intuitive), *but make it very clear to your grader what each macro does and that it can be done in the C language.*

(c) [3 points] Please show the code for M_ENTER, M_RET, and M_RETURN(t,v). Don't worry too much about the exact macro syntax (multi-statement macros in C are not intuitive), *but make it very clear to your grader what each macro does and that it can be done in the C language.*

Imagine that there are multiple threads calling `work_dropoff()` and `work_pickup()`, and that for some application-specific reason they aren't concerned with the *association* between items which are dropped off and the resulting computation—in other words, after a thread calls `work_dropoff()` it is just as happy if `work_pickup()` returns the result of another thread's `work_dropoff()` as it would be with its own result.

Imagine that an analysis of the work-flow for this application suggests that its performance would be improved by running multiple `worker()` threads—but that adding a queue structure to the monitor is not necessary: the existing single-entry mailbox will be adequate.

(d) [5 points] Would the system operate correctly if multiple threads were running the existing `worker()` function? Either explain why it would be *incorrect* to use the the current `worker()` function in that way, explain why the current `worker()` function is a *good design* for the job, or explain why the `worker()` function should be rewritten (and then provide the code).

3. [15 points] Critical-section Algorithm.

Imagine that a special-purpose multi-threaded application will never have more than 31 threads, which will always have thread id's ranging from 1 to 31 (inclusive). A developer for that system proposes the following lock-acquisition code, inspired by the famous Bakery Algorithm, but tuned according to the observation that the number of threads happens to fit in the number of bits in a word.

In this algorithm, the bits of one integer, `want`, indicate the set of threads attempting to acquire the lock. The thread that is granted the lock is indicated by the variable `turn`, which at any time has exactly one bit set; the position of that bit indicates the id of the thread owning the lock. This bit is rotated through the possible thread id's to ensure fair selection.

Because this system is very limited in scope, it contains exactly one critical region and thus one lock. Whenever a thread performs an operation on the single lock, it passes in its thread id (ranging from 1 to 31).

```
int want = 0; /* set of threads wanting the lock, begins empty */
int turn = 1; /* id-mask of thread that owning the lock; a value
               * of 1 indicates that no one has the lock */

#define set(x, i) ((x) |= (1 << (i)))
#define clr(x, i) ((x) &= ~((1 << (i))))
#define mybit(i) (1 << (i))

/* acquire a lock, given a thread id from 1 to 31 */
void lock_acquire(int thread_id)
{
    int self = mybit(thread_id);

    set(want, thread_id); /* indicate we are waiting for lock */

    /* spin waiting if it is someone else's turn */
    while ((turn != self) && (turn != 1))
        continue;

    if (turn == 1) /* it's no one's turn */
        turn = self; /* now it's my turn */
    /* postcondition: turn == self */
}
```

```
/* release a lock */
void lock_release(int thread_id)
{
    clr(want, thread_id);
    int new_turn = turn;
    if (want) { /* give lock to next waiting thread. Threads
                 * are chosen in a round-robin fashion by
                 * rotating the turn bit until a waiting
                 * thread is found */
        int n;
        for (n = 0; n < 31; n++) {
            new_turn <<= 1;
            if (new_turn == 0) new_turn = 2; /* wrap around */
            if (new_turn & want) { /* found waiting thread */
                turn = new_turn; /* grant entry to lock */
                return;
            }
        }
    } else {
        turn = 1; /* no one's turn -- no one wants it */
    }
}
```

For each of the critical-section algorithm requirements, either argue that it does or describe a scenario in which it does not. In the latter case, use the format presented in class, as exemplified in the table below. If you are trying to show a repeating pattern, be sure to show "enough" steps that the pattern is evident. Likewise, you do *not* need to show all 31 threads; you need to show enough to make your point but need not show any more.

T0	T1
set(want, id)	
	clr(want, id);

(a) [5 points] Does this algorithm ensure mutual exclusion? Briefly argue that it does, or show an execution trace indicating that it does not.

(b) [5 points] Does this algorithm ensure progress? Briefly argue that it does, or show an execution trace indicating that it does not.

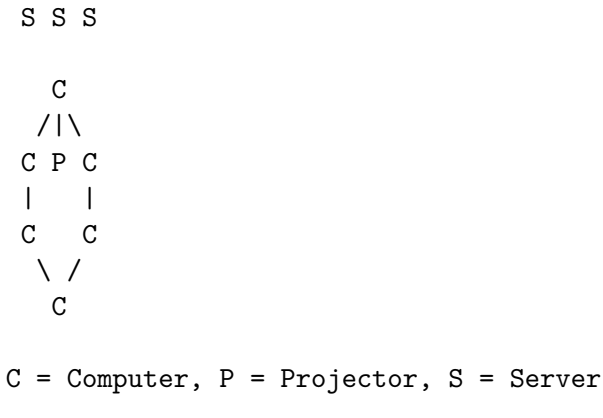
(c) [5 points] Does this algorithm ensure bounded waiting? Briefly argue that it does, or show an execution trace indicating that it does not.

You may use this page as extra space if you wish.

4. [20 points] Deadlock.

Yesterday Cluster Services began a project to construct a new Linux "collaboration mini-cluster" on campus (yay!). While they have a design in hand, they are uncertain as to how well it will work. Sensing an opportunity to make use of a room full of concurrency experts, they have asked Prof. Eckhardt to subject their design to analysis by an exam session full of 410 students.

Here is a diagram showing the layout of the proposed cluster.



Note that some resources are physically adjacent, represented by edges in the graph above. The projector is plugged into one computer and can project images from only that computer. All computers in the room are connected by a wireless network (which contributes no lines to the diagram). Each workstation computer, server, and projector can be used by only one person at a time.

Different groups utilize the cluster. OS students work in two-person teams, and require two adjacent computers so they can collaborate closely. From time to time Networks students will also come in and use the cluster. Each two-person Networks team requires one server and two workstation computers, but the computers do not need to be adjacent (the whole point of a Networks class is for computers which are not adjacent to send each other messages; in this case, each Networks group runs an IRC chat application server on the server computer they own).

The behavior of the OS students is as follows. One group member comes into the cluster and randomly selects a pair of adjacent computers, whether or not they are in use. If one of the selected computers is connected to the projector, the student will wait for that computer to open up, acquire that computer, wait for the projector to become available, watch one movie, release the projector, and then wait patiently for the partner to arrive. If neither computer in the pair selected is connected to the projector, the first student will simply wait for the left-most (or "clock-wise") computer to become available, acquire it, and wait patiently for the partner. The partner, upon arriving, will wait for the first team member's acquisition steps to complete, and then will wait for the other computer in the pair to become available, and acquire it. At this point the two students will work furiously on their OS project until their computron supply is exhausted, at which they will give up their computers and stagger out of the cluster.

The behavior of Networks students is as follows. The first partner of a team arriving at the cluster will wait until any server in the server pool is available. Once a server is acquired, the student will wait until any cluster computer is or becomes available, and will allocate it. The student will read `cmu.misc.market` until the partner arrives and acquires any other cluster computer (the computers need not be adjacent).

(a) [5 points] Assume that there is a week in the semester during which an OS assignment is in progress but no Networks assignment is.[1] Either demonstrate (through the use of a process/resource graph and an associated resource-request event trace) that OS students can deadlock, or explain why they cannot (i.e., state which deadlock condition(s) they avoid).

[1] This actually happens.

(b) [5 points] Assume that there is a week in the semester during which a Networks assignment is in progress but no OS assignment is.[2] Either demonstrate (through the use of a process/resource graph and an associated resource-request event trace) that Networks students can deadlock, or explain why they cannot (i.e., state which deadlock condition(s) they avoid).

[2] Please excuse the counter-factual assumption.

(c) [10 points] Due to a scheduling oversight, the due date for the Networks routing-daemon project and the OS thread-library project are the same. As a result, both Networks and OS students attempt to use the cluster at the same time. Either demonstrate (through the use of a process/resource graph and an associated resource-request event trace) that the mixture of OS and Networks students can deadlock, or explain why they cannot (i.e., state which deadlock condition(s) they avoid).

5. [15 points] Nuts & Bolts.

Your project partner angrily asks you for help arguing against a TA's "unfair" grading of some Project 1 code implementing an *extremely* minimal "game." Here is the code the TA marked with red ink.

```
char *
the_word(int num)
{
    char buf[8];

    switch (num % 4) {
        case 0: snprintf(buf, sizeof(buf), "zero"); break;
        case 1: snprintf(buf, sizeof(buf), "one"); break;
        case 2: snprintf(buf, sizeof(buf), "two"); break;
        case 3: snprintf(buf, sizeof(buf), "three"); break;
    }
    return (buf);
}
```

(a) [5 points] Apparently the TA marked this code as "exhibiting a conceptual misunderstanding of the C run-time environment which you should not have after passing the prerequisite for 15-410—see course staff." Explain what the TA believes to be wrong with the code.

(b) [10 points] Your partner angrily reports that, while the code in question is *theoretically* wrong, Project 1 code executes in a very stripped-down, controlled environment (see below). Therefore, even if this code "looks wrong" it will execute perfectly in this particular environment, and the TA should consider seeking employment elsewhere at CMU. Explain why your partner is wrong and the TA is right.

```
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();
    pic_init( BASE_IRQ_MASTER_BASE, BASE_IRQ_SLAVE_BASE );
    enable_interrupts();

    while (1) {
        int num, delay;

        for (num = 0; num < 16; ++num) {
            char *w, *cp;
            w = the_word(num);
            for (cp = w; *cp; cp++)
              if (*cp == 'z')
                printf("The number %d has the z characteristic.\n", num);
        }
        for (delay = 0; delay > 0; ++delay)
            continue;
    }
    return 0;
}
```

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 );
```