Computer Science 15-410: Operating Systems
Mid-Term Exam (A), Spring 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.	20		
4.	15		
5.	20		
	80		

I will not discuss the contents of this 15-410 midterm with *anybody*, whether or not in this class, whether or not present in this exam session with me, before 19:30 Friday, March 2nd.

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 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 and can apply the concept when necessary.

(a) [5 points] Starvation

(b) [5 points] Thread-safe

2. [15 points] Process model.

During the "long" weekend between finishing up the thread library project and beginning the kernel project, you wondered if you could crash the reference kernel by violating the boundaries of your task's address space. You decided to write three very short tests, each one probing whether some part of your address space could be read from or written to. In order to avoid messy test program crashes, you decided to use system calls instead of pointer accesses. Finally, you imposed diversity requirements on yourself.

1. Each test would probe a different region/segment/area of the address space.
2. At least one test would probe readability, and at least one would probe writability.
3. At least one test should be "positive" (the read or write should be legal and should occur) and at least one test should be "negative" (the read or write should be illegal and should be rejected by the kernel).

You should be able to write each probe program in under 15 lines of code. Each should print the string "PASS" or "FAIL", indicating whether the kernel's response to your system call was correct or incorrect.

(a) [5 points] This test probes the _____ region/segment/area for readability / writability (circle one), which the kernel should allow / disallow (circle one).

(b) [5 points] This test probes the _____ region/segment/area for readability / writability (circle one), which the kernel should allow / disallow (circle one).

(c) [5 points] This test probes the _____ region/segment/area for readability / writability (circle one), which the kernel should allow / disallow (circle one).

3. [20 points] Conditions.

Suddenly, you wake up. There's a message in your inbox urgently demanding to know where the new test code is. Then you remember... after a challenging but rewarding semester in 15-410, you were asked to serve as a teaching assistant. When you made the mistake of criticizing the quality of the thread-library test suite, Professor Eckhardt responded by asking you to write some condition variable test code.

In particular, you are asked to write a program which tests whether the following property is true of a condition-variable implementation: when two threads are awaiting a condition and the condition is signaled, one thread is awakened but the other thread is not. Your program should, with very high probability, print the string "PASS" if the condition variable implementation you exercise obeys this condition and "FAIL" if it does not. Your program should make its decision in less than ten seconds.

Because you are a fan of the dystopian science fiction thriller "Bladerunner," you decide to package the thread-related and housekeeping data structures (e.g., condition variables, mutexes, ints, chars, doubles, ...) used by your test in a `struct nexus` and to limit the struct to containing six fields.

(a) [5 points] Please declare your `struct nexus` here. Also write a function `void nexus_init(nexus_p np)` to initialize a nexus.

```
typedef struct nexus {

} *nexus_p;

void nexus_init(nexus_p np)
{

}
```

(b) [15 points] Now please write your test code. The suggested structure is one to three small helper functions and a `main()` function. Note that, as an interface-compliant test program, you cannot inspect or modify the internals of any thread-library data objects.

You may use this page as extra space for your "one-two test" solution if you wish.

4. [15 points] Inscrutable code.

Another part of your duties as a 15-410 teaching assistant is evaluating claims made by students in your class. For example, it is not unusual for a student to claim that the Project 2 reference kernel contains a bug. Sometimes the bug is in the kernel, but other time it is in the student's code. Consider the following example, which the student wrote in assembly language "so there is no question about what my code does."

```
#define ASSEMBLER
#include <syscall_int.h>

# syntax is "INSTRUCTION SOURCE, DEST"

    .text
.globl main
    .type   main, @function

main:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %esi
    pushl   $8
    call    _malloc
    movl    $0x80000000, (%eax)
    addl    $4, %eax
    movl    $4096, (%eax)
    movl    %eax, %esi
    int     $NEW_PAGES_INT
    movl    $0x80000000, %eax
    subl    $4092, %eax
    movl    %eax, %esi
    int     $THREAD_FORK_INT
    movl    %eax, %esi
    int     $SET_STATUS_INT
    movl    %eax, %esi
    int     $VANISH_INT
    popl    %esi
    movl    %ebp, %esp
    popl    %ebp
    ret
    .ident  "bmm: (angry) 3.14"
```

Before going to sleep, the student mailed you the code along with a declaration that the kernel is incorrectly executing it and a request to discuss the matter before class the next day. This means that you will need to figure out, on your own, how a Pebbles kernel *should* execute this code.

What *should* be the result of running this program?

1. Should it complete, hang, or crash?
2. Should the kernel or the shell print any diagnostics? If so, what?
3. If there are multiple alternative outcomes, briefly outline them.

Be sure to briefly but convincingly support your claims. Note that it is neither necessary nor desirable to provide a blow-by-blow description of the effect of each instruction on every register—your answer should focus on the questions listed above as well as describing *significant* other effects.

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

5. [20 points] Deadlock

The CMU Computer Club operates a machine room in the subbasement of Cyert Hall. The computers in this machine room provide a variety of services for the campus community, as summarized in the table below.

Machine	Purpose
aluminum	AFS File Server
astatine	Contrib Web Server
cadmium	MySQL Database
magnesium	Mail Exchanger
sodium	Kerberos KDC

In addition, the machine room is staffed by a number of former and future 15-410 students who are very conscientious about data reliability. Thus, six months ago, two of the club operators (known informally as zero_cool and crash_override) developed two backup systems to preserve data.

Unfortunately a problem was recently encountered with the backup systems that neither zero_cool nor crash_override can diagnose, and you have been asked to debug the system and determine the problem.

The main component categories of both backup systems are as follows.

Tape Drives Both backup systems use one or more tape drives to write data to magnetic tape for storage. There are exactly two tape drives (labeled A & B), connected to the same machine.

Tape Robot The tape robot is a device that mechanically fetches tapes from the tape library and inserts them into one of the tape drives. It can also remove tapes from the drives and place them back in the library.

Machine Data Both backup systems back up data from one or more machines.

Previously the tape devices were accessed by the two backup systems without coordination, but one day an unexpected sequence of conflicting commands resulted in a bearing in the robot's arm overheating and failing. Since then, a locking protocol has been established to ensure correct operation of each device.

1. Each tape drive must be locked before use to prevent multiple processes from issuing contradictory requests (which could Result in the tape drives eating tapes).
2. The tape robot must be locked before use to prevent conflicting commands from resulting in the tape robot fighting itself.
3. Each machine must be locked before backup in order to guarantee a consistent snapshot of machine state.

This space intentionally left nearly blank.

Primary backup system

The primary backup system, developed by zero_cool, serves to make daily backups of user file data and administrative configuration information (the per-machine user account list, software package configuration, log files, etc.) from each machine in the club's server farm (as described above).

The primary backup system is executed as a daily cron (batch) job, starting at 03:00, and the entire session typically takes about two hours. The resource allocation procedure for this backup system is as follows.

```
Lock Tape Robot.

For each machine m ...
    Randomly pick a tape drive t to lock.
    Lock tape drive t.

    Lock machine m.
    Dump the data on machine m to tape drive t.
    Unlock machine m.

    Unlock tape drive t.
... repeat for all machines.

Unlock Tape Robot.
```

Note that primary backups are made to a random tape drive to avoid uneven wear.

Even though the resource allocation protocol involving multiple machines in a distributed environment is complicated, it turns out that the locking semantics can be modeled by mutexes similar to those used in the Project 2 thread library.

This space intentionally left nearly blank.

(a) [2 points] Add code to the following template to allocate resources and initiate backups as described by the primary backup procedure above.

```
#define NUM_MACHINES 5

typedef enum {TAPE_DRIVE_A, TAPE_DRIVE_B} tape_drive_t;

/* Available functions (none of these can fail). */
void backup_machine(int machine, tape_drive_t tape_drive);
tape_drive_t choose_random_tape_drive(void); /* Returns TAPE_DRIVE_A or TAPE_DRIVE_B. */
void mutex_lock(mutex_t *mp);
void mutex_unlock(mutex_t *mp);

/* Available global variables. */
mutex_t *machine[NUM_MACHINES]; /* Includes mutexes for all machines. */
mutex_t *tape_drive_a, *tape_drive_b;
mutex_t *tape_robot;

/* The primary backup system. */
void primary_backup(void)
{ /* Fill in code here. */









}
```

(b) [3 points] Draw a process/resource graph illustrating the state of the system when data from one machine (e.g., aluminum), is being written to tape drive A.

Database backup system

Because the MySQL database files are very large, and dumping them requires special interactions with the MySQL server application, crash_override has developed a separate database backup application. This system complements the primary backup system by making archival backups of the MySQL files on the database machine each time the database tables reach a threshold size. Since this system is triggered by a data-specific threshold, it can execute at any time of day. However, the application guarantees that a backup will not be triggered until a previously-running backup has completed. In practice, a database backup is triggered around once a month and takes about twelve hours to complete. The resource allocation procedure for this backup system is as follows.

```
Lock Database Machine.
Lock Tape Robot.

Lock Tape Drive A.
Dump the database contents to drive A.
Unlock Tape Drive A.

Lock Tape Drive B.
Dump the database contents to drive B.
Unlock Tape Drive B.

Unlock Tape Robot.
Unlock Database Machine.
```

Note that database backups are written to two tapes, using both tape drives. This approach was chosen by crash_override as a defense against one tape drive eating a tape or a tape becoming unreadable. Since the tape drives are backing up the same content, the writes must be sequential.

(c) [2 points] Add code to the following template to allocate resources and initiate backups as described by the database backup procedure above.

```
#define NUM_MACHINES 5
#define DB_MACHINE 2 /* cadmium, from table */

typedef enum {TAPE_DRIVE_A, TAPE_DRIVE_B} tape_drive_t;

/* Available functions (none of these can fail). */
void database_backup(tape_drive_t tape_drive);
void mutex_lock(mutex_t *mp);
void mutex_unlock(mutex_t *mp);

/* Available global variables. */
mutex_t *machine[NUM_MACHINES]; /* Includes mutexes for all machines. */
mutex_t *tape_drive_a, *tape_drive_b;
mutex_t *tape_robot;

/* The database backup system. */
void database_backup(void)
{ /* Fill in code here. */









}
```

(d) [3 points] Draw a process/resource graph illustrating the state of the system when a database dump is being written to tape drive B.

The plot thickens

One day, a Computer Club user sends mail stating that he accidentally trashed his home directory and needs it restored from backup. Computer Club operator zero_cool fetches the most recent backup tape and discovers that the most recent backup was made five months ago!

Performing a cursory equipment scan, zero_cool determines that Tape Drive A has a tape loaded in it and all the equipment appears to be functioning correctly. However, neither primary backups nor database backups have been performed, for some unknown reason, for the past five months.

Of course, zero_cool assumes that the problem must be due to a bug in crash_override's database backup script that caused the tape robot to malfunction. Meanwhile, crash_override also discovers the backup failure, and assumes the problem must be due to a bug in zero_cool's primary backup script. They begin an argument with each other and defend their respective code bases by showing their process/resource graphs to prove that each one's code contains no bug that could have resulted in this condition.

(e) [3 points] Who is right? Is there a bug in one of the two backup systems that would have resulted in the described condition? If not, how did this situation arise?

(f) [2 points] Draw the process/resource graph of the overall system that illustrates problem encountered. Use the graph and a brief textual or pseudo-code explanation to convincingly explain what went wrong.

(g) [5 points] What is the simplest change you could suggest making to the system as a whole which would eliminate this problem?

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 cas_runflag(int tid, int *oldp, int *expectp, int *newp);
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 );
```