Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (B), Fall 2017

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.	10		
3.	15		
4.	20		
5.	10		
	65		

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) $\boxed{\text{5 points}}$ Please list the necessary conditions for deadlock. For each condition, briefly define the condition, describe one approach that the system could use to prevent the condition from occuring, and describe a potential disadvantage of that approach.

You may use this page as extra space for the deadlock short-answer question if you wish.

(b) $\boxed{\text{5 points}}$ Please discuss the role of an x86 interrupt acknowledgment:

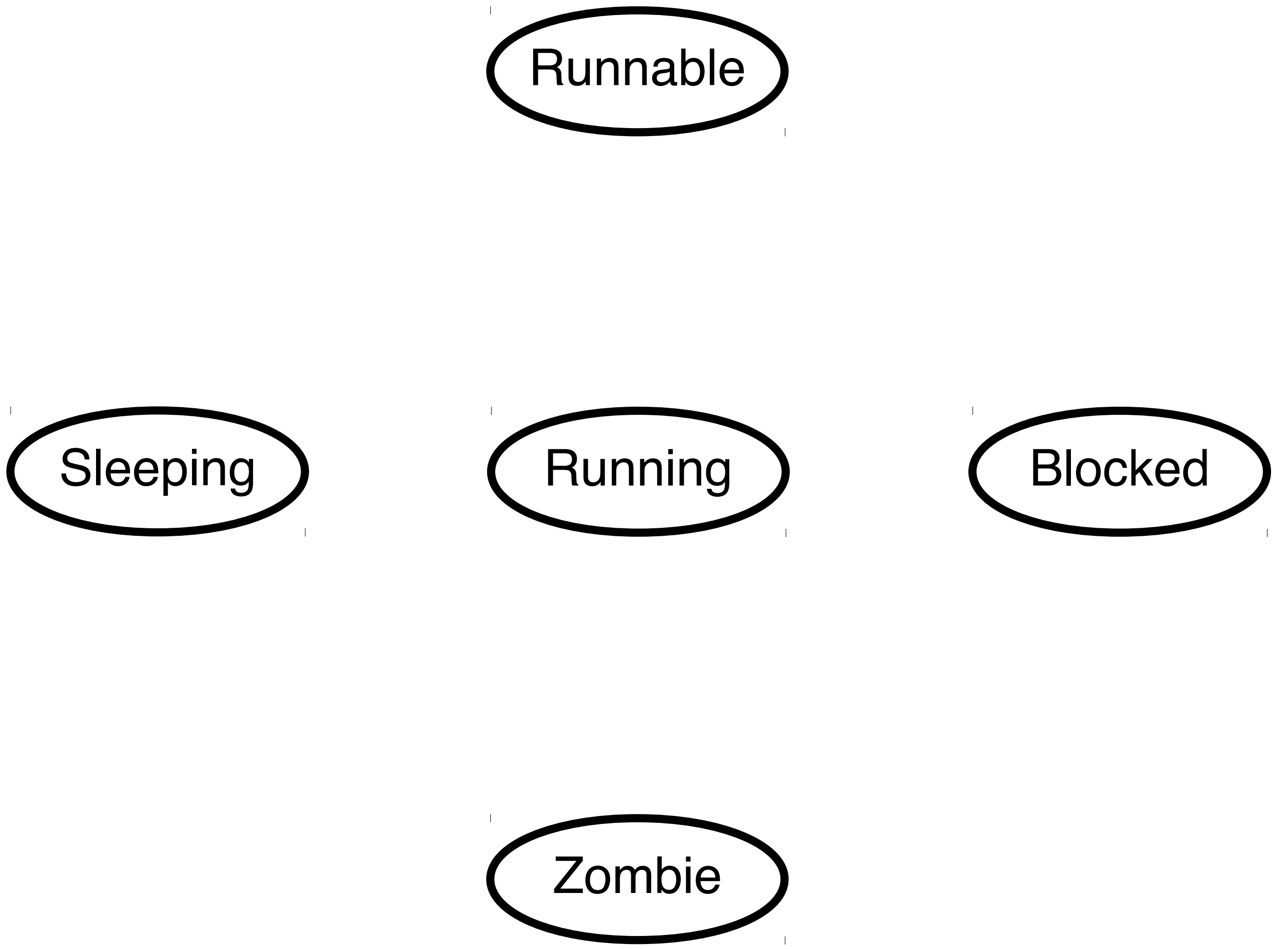
(i) What part of the system sends the acknowledgement, and what part receives it?

(ii) What does the receipt of the interrupt acknowledgement tell the receiver?

(iii) What specific x86 mechanism is used to send the acknowledgement?

Andrew ID: ____________

2. $\boxed{\text{10 points}}$ Draw a diagram showing the transitions among the following scheduling states: Running, Runnable, Sleeping, Blocked, and Zombie (Zombie may also be known as Exiting). Label the arcs with events which make sense in terms of events that happen in Pebbles kernels. If a particular transition can be caused by more than one event, show two on the diagram (try to make the two of them as different from each other as you can). If you believe some "arcs" have a number of endpoints other than two, you might be able to convince us.



3. $\boxed{\text{15 points}}$ "Nemo's Algorithm II".

Homework 1 included a critical-section protocol called "Nemo's Algorithm." The code we asked you to evaluate in Homework 1 was a variant of Dijkstra's (1965) n-process generalization of Dekker's (1965) two-process solution, as presented in a 1986 scholarly book by Michel Raynal, *Algorithms for Mutual Exclusion.* As you (hopefully!) showed in your HW1 answer, Dijkstra's solution does not provide bounded waiting. In this exam question we will ask you to consider a subtle variant of Dijkstra's solution—the author of this variant doesn't like negative numbers, so instead of using `turn == -1` to indicate "nobody," this version just sets `turn` to the i.d. of an arbitrary thread (in this case, zero, the i.d. of T0). Here is the code.

```
volatile int turn = 0;     // initialize to i.d. of an arbitrary thread
int entering[N] = {0, };   // per-thread flags (initially all zero)

int n_entering(void) {
    int n_entering = 0;
    for (int t = 0; t < N; t++)
        n_entering += entering[t];
    assert(n_entering > 0);
    return (n_entering);
}

void lock(void) {
    do {
        do {
            entering[i] = 0;
            if (turn == 0)
                turn = i;
        } while (turn != i);
        entering[i] = 1;
    } while (n_entering() != 1);
}

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

As it turns out, this variant fails to ensure *both* bounded waiting *and* some other critical property. For full credit, identify a failure *other than* a lack of bounded waiting and then present a trace which supports your claim (in an emergency, for a small amount of partial credit, you could show us a bounded-waiting-failure 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 `entering[]` as `e[]` and `n_entering()` as `n_e()`. *Be sure that the execution trace you provide us with is easy to read and conclusively demonstrates the claim you are making. When writing a trace, claims that some sequence repeats must be both precise and correct–writing "now this repeats" at the bottom of a trace is usually not precise and is often incorrect, and thus often results in significant point deductions. It is to your advantage to use scrap paper or the back of some page to experiment with draft traces, so that the answer you write below is easy for us to read.*

You may use this page for the critical-section protocol question.

Andrew ID: ______________

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

4. 20 points Multi-locks.

In lecture we talked about two fundamental operations in concurrent programming: brief mutual exclusion for atomic sequences (provided in P2 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.

In this question you will implement a synchronization object called a "multi-lock." This object is designed to help applications that must acquire locks in an order which is undefined or unpredictable by the program author. The idea is that the program delegates management of a set of locks to a single "multi-lock" object, at which point the program can request the locks managed by the multi-lock in any order. It is the job of the multi-lock to ensure that the program's threads can acquire the locks managed by the multi-lock without deadlock (obviously the multi-lock object cannot ensure that the entire program is deadlock-free).

The multi-lock object we will ask you to implement has two noteworthy features:

1. If a thread asks to acquire a lock which it already holds, this is not an error (the request merely succeeds),
2. While an application using a multi-lock may acquire the underlying locks in any order, the only unlock operation, `ml_unlock_all()`, releases all underlying locks held by the unlocking thread. So a thread may call `ml_lock()` any number of times, acquiring (or re-acquiring) locks in any order, followed by a call to `ml_unlock_all()`, after which it may go back to acquiring locks.

As an example, consider the following trace which would deadlock if the application directly used two P2 mutexes `m1` and `m2`, but must not deadlock when a multi-lock is used.

Time	Thread 1	Result	Thread 2	Result
0	ml_lock(m, 1)	Holds #1		
1			ml_lock(m, 2)	Holds #2
2	ml_lock(m, 2)	...wait...		
3		...wait...	ml_lock(m, 1)	No deadlock!
4		Holds #1,#2		...wait...
5	ml_unlock_all(m)	Holds nothing		...wait...
6				Holds #1,#2
7			ml_unlock_all(m)	Holds nothing

A small example program using a multi-lock is displayed on the next page.

```
#define NTHREADS 10
#define NBALLS 20
#define PICKUPS 30

mlock_t mlock;
int     tids[NTHREADS];
void*   threadbody(void*);

// Returns a random valid ball
// ENSURES: 0 <= result < NBALLS
int nextBall(void);

int main() {
    thr_init(4096);  // exam: no failures

    ml_init(&mlock, NBALLS);  // exam: no failures

    for (int t = 0; t < NTHREADS; t++) {
        tids[t] = thr_create(threadbody, (void*)t);  // exam: no failures
    }

    for (int t = 0; t < NTHREADS; t++) {
        thr_join(tids[t], NULL);
    }

    ml_destroy(&mlock);
    thr_exit(0);
}

void* threadbody(void* tid_arg) {
    int tid = (int)tid_arg;

    for (int p = 1; p < PICKUPS; p++) {
        int ball = nextBall();
        ml_lock(&mlock, ball);

        if ((p % (tid + 1)) == 0) { // Occasionally drop the balls
            ml_unlock_all(&mlock);
        }
        thr_yield(-1);
    }
    ml_unlock_all(&mlock);

    return NULL;
}
```

Your task is to implement multi-lock with the following interface:

- `int ml_init(mlock_t* m, int size)`
  Initializes a multi-lock object containing `size` locks. Because this is an exam, you may assume that allocating and initializing the necessary state will succeed (thus, this declaration shows the function returning a value so that the declaration matches what a non-exam implementation would declare).
- `void ml_lock(mlock_t* m, int lock_number)`
  `REQUIRES: 0 <= lock_number < size`

  Acquires the lock specified by `lock_number`. In contrast to other locking primitives, it is legal for a thread to invoke `ml_lock()` repeatedly on the same multi-lock, even specifying acquisition of an already-acquired underlying lock. Threads operating on a single multi-lock object should not deadlock as a result of `ml_lock()` operations (obviously, there are other ways for threads to deadlock).
- `void ml_unlock_all(mlock_t* m)`
  Drops all underlying locks of this multi-lock which are held by the invoking thread.
- `void ml_destroy(mlock_t *m)`
  Destroys the multi-lock object. It is illegal for a program to invoke `ml_destroy()` if any threads are operating on it.

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 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) $\boxed{\text{5 points}}$ Please declare your `mlock_t` here. If you need one (or more) auxilary structures, you may declare it/them here as well.

```
typedef struct {







} mlock_t;
```

(b) $\boxed{\text{15 points}}$ Now please implement `int ml_init()`, `void ml_lock()`, `void ml_unlock_all()`, and `void ml_destroy()`.

…space for multi-lock implementation …

. . . space for multi-lock implementation . . .

…space for multi-lock implementation …

Andrew ID: ____________

5. $\boxed{\text{10 points}}$ `Nuts & Bolts`.

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

1. Determine which "wrong register value(s)" caused the thread to run an instruction which resulted in a fatal exception.
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 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.

(Continued on next page)

(a) $\boxed{\text{5 points}}$ `Registers:`

```
eax: 0x00000000, ebx: 0x00000000, ecx: 0x0000000a,
edx: 0xffffef2c, edi: 0x00000000, esi: 0x00000000,
ebp: 0xffffefe0, esp: 0xffffefd0, eip: 0x00000000,
 ss:     0x002b,  cs:     0x0023,  ds:     0x002b,
 es:     0x002b,  fs:     0x002b,  gs:     0x002b,
eflags: 0x00000246
```

(b) $\boxed{\text{5 points}}$ `Registers:`

```
eax: 0x00000000, ebx: 0x00000000, ecx: 0x0000000a,
edx: 0xffefb70c, edi: 0x00000000, esi: 0x00000000,
ebp: 0xffefb7a8, esp: 0xffefc000, eip: 0x0100008d,
 ss:     0x002b,  cs:     0x0023,  ds:     0x002b,
 es:     0x002b,  fs:     0x002b,  gs:     0x002b,
eflags: 0x00000246
```

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: ____________

Useful-Equation Cheat-Sheet

$$\cos^2\theta + \sin^2\theta = 1$$
$$\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$$
$$\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta$$
$$\sin 2\theta = 2\sin\theta\cos\theta$$
$$\cos 2\theta = \cos^2\theta - \sin^2\theta$$
$$e^{ix} = \cos(x) + i\sin(x)$$
$$\cos(x) = \frac{e^{ix} + e^{-ix}}{2}$$
$$\sin(x) = \frac{e^{ix} - e^{-ix}}{2i}$$

$$\int \ln x\, dx = x\ln x - x + C$$
$$\int_0^\infty \sqrt{x}\, e^{-x}\, dx = \frac{1}{2}\sqrt{\pi}$$
$$\int_0^\infty e^{-ax^2}\, dx = \frac{1}{2}\sqrt{\frac{\pi}{a}}$$
$$\int_0^\infty x^2 e^{-ax^2}\, dx = \frac{1}{4}\sqrt{\frac{\pi}{a^3}} \text{ when } a > 0$$
$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\, dt$$

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},\, t) = \hat{H}\Psi(\mathbf{r}, t)$$
$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},\, t) = -\frac{\hbar^2}{2m}\nabla^2\Psi(\mathbf{r},\, t) + V(\mathbf{r})\Psi(\mathbf{r},\, t)$$
$$E = hf = \frac{h}{2\pi}(2\pi f) = \hbar\omega$$
$$p = \frac{h}{\lambda} = \frac{h}{2\pi}\frac{2\pi}{\lambda} = \hbar k$$
$$\nabla\cdot\mathbf{E} = \frac{\rho}{\varepsilon_0}$$
$$\nabla\cdot\mathbf{B} = 0$$
$$\nabla\times\mathbf{E} = -\frac{\partial\mathbf{B}}{\partial t}$$
$$\nabla\times\mathbf{B} = \mu_0\mathbf{J} + \mu_0\varepsilon_0\frac{\partial\mathbf{E}}{\partial t}$$

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.