Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Spring 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.	**15**		
3.	**15**		
4.	**20**		
5.	**10**		
	70		

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

Andrew ID: _____

After I leave this exam session, I will not discuss the contents of this 15-410/605 midterm with *anybody*, whether or not in this class, whether or not present in this exam session with me, for 24 hours.

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.

   (a) [5 points] When designing a body of code, at times one finds oneself thinking, "I wonder if I can assume X?" According to the 15-410 design orthodoxy, immediately upon having such a thought one is required to ask oneself two questions. Please state those questions.

(b) [5 points] List the *logical* components of an x86 32-bit protected-mode IDT entry (it is not necessary to be detailed about the *physical* layout).

2. [15 points] Consider the following critical-section protocol:

```
       boolean waiting[2] = { false, false };
       int turn = 0;

 1.    do {
 2.        waiting[i] = true;
 3.        while (waiting[j]) {
 4.            waiting[i] = false;
 5.            while (turn == j)
 6.               continue;
 7.            waiting[i] = true;
 8.        }
 9.       ...begin critical section...
10.       ...end critical section...
11.       turn = j;
12.       waiting[i] = false;
13.       ...begin remainder section...
14.       ...end remainder section...
15.   } while (1);
```

This protocol is presented in "standard form," i.e.,

1. When process 0 is running this code, $i == 0$ and $j == 1$; when process 1 is running this code, $i == 1$ and $j == 0$, so i means "me" and j means "the other process."
2. Lines 2–8 are can be thought of roughly as "acquiring a lock" and lines 11–12 can be thought of roughly as "releasing the lock."

(a) [15 points] There is a problem with this protocol. That is, it does not ensure that all three critical-section algorithm requirements are always met. Identify a requirement which is not met and lay out a scenario which demonstrates your claim. Use the format presented in class, i.e.,

P0	P1
waiting[0] = false;	
	turn = 0;

You may introduce temporary variables or other obvious notation as necessary to improve the clarity of your answer. *Be sure that the execution trace you provide us with is easy to read and conclusively demonstrates the claim you are making.* It is possible to answer this question with a brief, clear trace, so you should do what is necessary to ensure that you do.

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

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

3. [15 points] Pair matching.

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 "pair matcher." The idea is that some parallel tasks must be worked on by pairs of threads, and the threads need some (dynamic) way to pick a partner to work with. After a pair matcher is initialized, an even number of threads will invoke the `match` operation. The `match` operation involves some amount of thread synchronization, potentially including blocking, and then returns to each thread, in a reasonably timely fashion, the thread identification number of the thread it has been matched with. A match object does not know how many threads will invoke it, though it can depend on the number being even. Once a program is sure that no more threads will invoke the `match` operation on a particular object, the `destroy` operation can and should be invoked.

A small example program using a pair matcher is displayed on the next page.

The remainder of this page is intentionally blank.

```
#define NTHREADS 410
int tids[NTHREADS];
pmatch_t matcher;
void *threadbody(void *ignored);

int main(int argc, char** argv)
{
    thr_init(4096); // exam: no failures

    pmatch_init(&matcher); // 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);
    }
    printf("Done\n");
    pmatch_destroy(&matcher);
    thr_exit(0);
}

void *threadbody(void *ignored)
{
    int me = thr_getid();
    int partner;
    int done = 0, rounds = 0;

    while (!done) {
        int coolpartner;

        partner = pmatch_match(&matcher);

        printf("I am %d, my partner is %d\n", me, partner);

        coolpartner = (me&1) == (partner&1);

        if (coolpartner) {
            printf("Whee!  That was so much fun I might do it again.\n");
        }
        if ((++rounds == 10) || !coolpartner) {
            done = 1;
        }
    }
    return 0;
}
```

Your task is to implement pair matchers with the following interface:

- `int pmatch_init(pmatch_t *pmp)` — initializes a pair matcher.
- `int pmatch_match(pmatch_t *pmp)` — "Reasonably promptly" returns the thread i.d. of another thread invoking `pmatch_match()` on the same pair-matcher object. Note that `pmatch_match()`, as specified for this exam, does not return error codes.
- `void pmatch_destroy(pmatch_t *pmp)` — Deactivates a pair-matcher object. It is illegal for a program to invoke `pmatch_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 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) [3 points] Please declare your `pmatch_t` here. If you need one (or more) auxilary structures, you may declare it/them here as well.

```
typedef struct {

} pmatch_t;
```

(b) [12 points] Now please implement `int pmatch_init()`, `int pmatch_match()`, and `void pmatch_destroy()`.

. . . space for pair-matcher implementation . . .

. . . space for pair-matcher implementation . . .

4. [20 points] Ride-sharing deadlock!

You have doubtless heard the hype about "ride sharing" services, and perhaps have even used one. As you might have noticed, however, true ride *sharing*, in the sense of multiple riders who don't know each other sharing a car to a common destination, isn't activated in all cities by the major ride-sharing companies—not in Pittsburgh, for example. It turns out this is algorithmically trickier than it looks, even in the restricted case we will now consider.

Imagine a large event, such as a concert, produces a large queue of riders. When the concert is over, riders will start forming a queue and the ride-sharing company ("Ryed," perhaps you've heard of them?) will begin sending an infinite stream of cars to the concert venue, with the result that riders and drivers will be streaming through the system for a while; eventually we presume Ryed will stop sending cars, but that is not modeled in this question.

Each arriving driver invites the rider at the head of the queue into the car and then tries to fill the rest of the seats in the car with other riders who are going to the same destination. For the purposes of this problem we will assume that destinations are zip codes, and that there is a hash function which, given all zipcodes in Pennsylvania, collapses them down to a compact range of integers, from 0 to `NUM_ZIPCODES` (such hashing functions do exist—the trick is called "perfect hashing"). So once the driver acquires a destination, i.e., the destination of the rider at the head of the concert queue, the driver looks in a separate, per-zipcode, queue of other riders who are going to the same place. Once the car is full or the driver is tired of waiting, the car drives off and eventually drops off the riders. The ride-sharing company will definitely send (or re-circulate) enough drivers to get everybody home.

Below is code simulating the operation of the Ryed system.

The remainder of this page is intentionally blank.

```
V01    #include <stdlib.h>
V02    #include <syscall.h>
V03    #include <thread.h>
V04    #include <mutex.h>
V05    #include <sem.h>
V06    #include <cond.h>
V07    #include <queue.h>
V08
V09    #define NUM_ZIPCODES 2185 // http://data.mongabay.com/igapo/zip_codes/PA.htm
V10    #define WAIT_TIME 10
V11    #define DRIVE_TIME 10
V12    #define CAB_SIZE 3
V13
V14    #define WAITING 0
V15    #define RIDING 1
V16    #define AT_HOME 2
V17
V18    typedef struct {
V19        int status;
V20        int dest_code; // hash of zipcode
V21        sem_t at_home;
V22    } rider_t;
V23
V24    #define NUM_RIDERS 100
V25    rider_t riders[NUM_RIDERS];
V26
V27    mutex_t concert_lock;
V28    queue_t concert_queue;
V29
V30    mutex_t zipcode_locks[NUM_ZIPCODES];
V31    queue_t zipcode_queues[NUM_ZIPCODES];
```

```
R01    void *rider(void *arg) {
R02        rider_t *rider = (rider_t *)arg;
R03        int dest_code = perfect_ziphash(my_home_zipcode());
R04
R05        rider->status = WAITING;
R06        rider->dest_code = dest_code;
R07        sem_init(&rider->at_home, 0);
R08
R09        mutex_lock(&concert_lock);
R10        mutex_lock(&zipcode_locks[dest_code]);
R11
R12        enq(&concert_queue, rider);
R13        enq(&zipcode_queues[dest_code], rider);
R14
R15        mutex_unlock(&concert_lock);
R16        mutex_unlock(&zipcode_locks[dest_code]);
R17
R18        sem_wait(&rider->at_home);
R19
R20        return NULL; // we're home!
R21    }
```

```
D01    void *driver(void *arg) {
D02        queue_t cab_queue; // stores riders in my car now
D03        queue_init(&cab_queue);
D04
D05        mutex_lock(&concert_lock);
D06        if (queue_empty(&concert_queue)) {
D07            mutex_unlock(&concert_lock);
D08            return NULL; // everybody has left
D09        }
D10
D11        rider_t *rider = deq(&concert_queue);
D12        int dest_code = rider->dest_code;
D13
D14        // remove the rider from the corresponding zipcode queue
D15        mutex_lock(&zipcode_locks[dest_code]);
D16        remove(&zipcode_queues[dest_code], rider);
D17        mutex_unlock(&concert_lock);
D18
D19        rider->status = RIDING; enq(&cab_queue, rider); int num_riders = 1;
D20
D21        int waits = 0;
D22        while (num_riders < CAB_SIZE) {
D23            // wait for more riders to the same dest_code
D24            while (queue_empty(&zipcode_queues[dest_code])) {
D25                mutex_unlock(&zipcode_locks[dest_code]);
D26                if (++waits == WAIT_TIME)
D27                    goto drive_home;
D28                sleep(1);
D29                mutex_lock(&zipcode_locks[dest_code]);
D30            }
D31
D32            rider = deq(&zipcode_queues[dest_code]);
D33
D34            // remove the rider from the concert queue as well
D35            mutex_lock(&concert_lock);
D36            remove(&concert_queue, rider);
D37            mutex_unlock(&concert_lock);
D38
D39            rider->status = RIDING; enq(&cab_queue, rider); num_riders++;
D40        }
D41        mutex_unlock(&zipcode_locks[dest_code]);
D42
D43    drive_home:
D44        sleep(DRIVE_TIME);
D45
D46        // drop them all off!
D47        while (num_riders-- > 0) {
D48            rider = deq(&cab_queue);
D49            rider->status = AT_HOME;
D50            sem_signal(&rider->at_home);
D51        }
D52        return NULL;
D53    }
```

```
M01    int main() {
M02        int z,r,d;
M03        mutex_init(&concert_lock);
M04        queue_init(&concert_queue);
M05
M06        for (z = 0; z < NUM_ZIPCODES; z++) {
M07            mutex_init(&zipcode_locks[z]);
M08            queue_init(&zipcode_queues[z]);
M09        }
M10
M11        for (r = 0; r < NUM_RIDERS; r++) {
M12            thr_create(rider, (void *)&riders[r]);
M13        }
M14
M15        // infinite stream of Ryed drivers -- this is "road show" demo code
M16        while (1) {
M17            thr_create(driver, (void*)NULL);
M18            sleep(15);
M19        }
M20        return 0;
M21    }
```

(a) [10 points] Unfortunately, the code shown above can deadlock. Show *clear, convincing* evidence of deadlock. Begin by *describing the problem* in one or two sentences; then *clearly specify a scenario.* **If possible (i.e., for full credit), describe a deadlock involving only one type of thread (some number of riders or some number of drivers), though you may instead describe a deadlock involving multiple thread types if necessary.**

If you cannot describe a particular exact deadlock, or are having trouble describing how it would occur, you may receive partial credit by describing which deadlock ingredients are and/or are not exhibited by the code above. *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 your deadlock answer if you wish.

(b) [10 points] Explain in detail how to fix the deadlock you have identified. You should begin by providing a theoretical / conceptual rationale for how your solution works. While showing code is not *strictly* required if your solution description is very clear, we expect that most students will rewrite one to three small blocks of code. Please note that there are multiple ways to solve any given deadlock–some good, but some which are very undesirable. *Because we will assign points according to our beliefs about quality, writing down the first solution you think of may not be the best strategy.*

You may use this page for your deadlock answer if you wish.

5. [10 points] `Nuts and bolts`

You have signed up to spend Spring Break working on a one-week interdisciplinary super-agile rapid-prototyping entrepreneurship project mini class, 99-999. Your group is working on building a new operating system, similar in some ways to Plan 9, for a new processor, similar in many ways to the venerable x86-32 platform—but with extra features related to security.

Unfortunately, something is wrong with your group's project. The hardware people suspect the software, and the software people suspect the hardware. Both the hardware people and the software people suspect the compiler people. Because you are the group's most interdisciplinary and agile person (plus entrepreneurial!), you jump right in to debugging. You shrink the problem down, over and over, until you are left with one simple C function that isn't working, as shown below.

```
unsigned int offset(unsigned int a, unsigned int b) {
    return a + b;
}
```

Eventually you figure out that the new agile interdisciplinary secure processor contains a serious hardware flaw: the 32-bit add instruction computes wrong answers! In particular, the processor correctly adds the values it operates on, but as it fetches the two source operands, it mistakenly thinks the low-order 16 bits of each input is all zeroes. Here is an example.

```
MOVL $0xCAFEF00D, %eax  # %eax = 0xCAFEF00D
MOVL $0xDEADC0DE, %edx  # %edx = 0xDEADC0DE
ADDL %edx, %eax         # %eax = %eax + %edx = 0xA9AB0000, should be 0xA9ACB0EB!!!
```

The software people are stunned at such an outcome, but the hardware people muttered "inputs pulled to zero for lower 16 bits" and "bugs like this happen all the time." Unfortunately, due to your entrepreneurial development cycle, you can't stop the project while your agile virtual fabrication facility, sailing above the planet on a solar-powered glider, produces a working processor chip. A decision is made to work around the problem in software. Your agile proof-of-concept task for the current sprint is to rewrite the `offset()` function so that it produces correct answers. You are allowed to use the broken `ADDL` instruction, but presumably will need to add some other code to correct for the problem. Note that testing reveals that the `SUBL` instruction is broken in exactly the same way.

The remainder of this page is intentionally blank.

(a) [10 points] Please provide a working implementation of the arithmetic core of the `offset()` function. You may present your solution in C or in x86-32 assembly code as you prefer (but you may not mix C and assembly code). If you write assembly code, you can assume, as above, that the source operands are in `%eax` and `%edx`, i.e., you don't need to worry about x86-32 stack discipline. If you write C code, you may assume that the "+" operator relies on the broken `ADDL` instruction, but that other C operators work correctly.

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."

Andrew ID: _____

Typing Rules Cheat-Sheet

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \to \tau \mid \mu\alpha.\tau \mid \forall\alpha.\tau \\ e & ::= & x \mid \lambda x{:}\tau.e \mid e\,e \mid \mathsf{fix}(x{:}\tau.e) \mid \mathsf{fold}_{\alpha.\tau}(e) \mid \mathsf{unfold}(e) \mid \Lambda\alpha.e \mid e[\tau] \end{array}$$

$$\frac{}{\Gamma,\, \alpha\,\mathbf{type} \vdash \alpha\,\mathbf{type}}\ \text{istyp-var} \qquad \frac{\Gamma \vdash \tau_1\,\mathbf{type} \quad \Gamma \vdash \tau_2\,\mathbf{type}}{\Gamma \vdash t_1 \to t_2\,\mathbf{type}}\ \text{istyp-arrow}$$

$$\frac{\Gamma,\, \alpha\,\mathbf{type} \vdash \tau\,\mathbf{type}}{\Gamma \vdash \mu\alpha.\tau\,\mathbf{type}}\ \text{istyp-rec} \qquad \frac{\Gamma,\, \alpha\,\mathbf{type} \vdash \tau\,\mathbf{type}}{\Gamma \vdash \forall\alpha.\tau\,\mathbf{type}}\ \text{istyp-forall}$$

$$\frac{}{\Gamma,\, x : \tau \vdash x : \tau}\ \text{typ-var} \qquad \frac{\Gamma,\, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash \tau_1\,\mathbf{type}}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ \text{typ-lam} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\,e_2 : \tau_2}\ \text{typ-app}$$

$$\frac{\Gamma,\, x : \tau \vdash e : \tau \quad \Gamma \vdash \tau\,\mathbf{type}}{\Gamma \vdash \mathsf{fix}(x{:}\tau.e) : \tau}\ \text{typ-fix}$$

$$\frac{\Gamma \vdash e : [\mu\alpha.\tau/\alpha]\tau \quad \Gamma,\, \alpha\,\mathbf{type} \vdash \tau\,\mathbf{type}}{\Gamma \vdash \mathsf{fold}_{\alpha.\tau}(e) : \mu\alpha.\tau}\ \text{typ-fold} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathsf{unfold}(e) : [\mu\alpha.\tau/\alpha]\tau}\ \text{typ-unfold}$$

$$\frac{\Gamma,\, \alpha\,\mathbf{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \text{typ-tlam} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau \quad \Gamma \vdash \tau'\,\mathbf{type}}{\Gamma \vdash e[\tau'] : [\tau'/\alpha]\tau}\ \text{typ-tapp}$$

$$\frac{}{\lambda x{:}\tau.e\,\mathbf{value}}\ \text{val-lam} \qquad \frac{}{\mathsf{fold}_{\alpha.\tau}(e)\,\mathbf{value}}\ \text{val-fold} \qquad \frac{}{\Lambda\alpha.\tau\,\mathbf{value}}\ \text{val-tlam}$$

$$\frac{e_1 \;\mapsto\; e_1'}{e_1\,e_2 \;\mapsto\; e_1'\,e_2}\ \text{steps-app}_1 \qquad \frac{e_1\,\mathbf{value} \quad e_2 \;\mapsto\; e_2'}{e_1\,e_2 \;\mapsto\; e_1\,e_2'}\ \text{steps-app}_2$$

$$\frac{e_2\,\mathbf{value}}{(\lambda x{:}\tau.e_1)\,e_2 \;\mapsto\; [e_2/x]e_1}\ \text{steps-app-}\beta$$

$$\frac{}{\mathsf{fix}(x{:}\tau.e) \;\mapsto\; [\mathsf{fix}(x{:}\tau.e)/x]e}\ \text{steps-fix}$$

$$\frac{e \;\mapsto\; e'}{\mathsf{unfold}(e) \;\mapsto\; \mathsf{unfold}(e')}\ \text{steps-unfold}_1 \qquad \frac{}{\mathsf{unfold}(\mathsf{fold}_{\alpha.\tau}(e)) \;\mapsto\; e}\ \text{steps-unfold}_2$$

$$\frac{e \;\mapsto\; e'}{e[\tau] \;\mapsto\; e'[\tau]}\ \text{steps-tapp}_1 \qquad \frac{}{(\Lambda\alpha.e)[\tau] \;\mapsto\; [\tau/\alpha]e}\ \text{steps-tapp}_1$$

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.