Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Spring 2013

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		

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 definition of each of the following terms *as it applies to this course*. We are expecting three to five sentences or "bullet points" for each definition. Your goal is to make it clear to your grader that you understand the concept and can apply it when necessary.

(a) [5 points] "runnable"

(b) [5 points] "starvation"

2. [15 points] Exceptional Throwing.

While students are working away on an OS exam, quite a lot of time may pass. One day, while waiting for students to finish an exam, Eric and Paul are bored and decide some physical activity would stimulate their minds, so they grab a ball and go out into the hallway to toss it back and forth. Eric is outside the exam-room door, so he can very quickly (in $O(1)$ time!) check whether the exam is over, but Paul, who is further down the hall, must rely on Eric "posting" the news of this happy event. The basic idea for their "exam ball-toss protocol" is that except when the ball is briefly sailing through the air, exactly one of them has possession of it; once it lands in somebody's hands, the other person observes this and calls out "Throw!", at which point it goes sailing through the air in the other direction.

In theory, once the exam is done the ball-players should finish up their game, return to the exam room, and help Professor Eckhardt carry the exams back to his office. But sometimes this doesn't happen: the game ends up "stuck" in the sense that not all the TA's return to the exam room. In order to figure out what's going on, Alex L has carefully observed how Paul and Eric play their game and turned his observations into the code appearing below.

```
typedef struct {
    volatile int has_ball;
    cond_t block;
    mutex_t lock;
} thrower;

static thrower eric;            void *run_eric(void *arg);
static thrower paul;            void *run_paul(void *arg);
static volatile int exam_over;  extern int check_exam_room();

int main(int argc, char *argv[]) {
    int eric_tid, paul_tid;

    eric.has_ball = 0;  cond_init(&eric.block);  mutex_init(&eric.lock);

    paul.has_ball = 0;  cond_init(&paul.block);  mutex_init(&paul.lock);

    exam_over = 0;

    if (thr_init(4*PAGE_SIZE)) panic("thr_init");

    eric_tid = thr_create(run_eric, NULL);
    paul_tid = thr_create(run_paul, NULL);

    if (thr_join(eric_tid, NULL) || thr_join(paul_tid, NULL)) panic("join");

    thr_exit(0);
    exit(0); // placate compiler
}
```

```
void *run_eric(void *arg) {

    mutex_lock(&eric.lock);
    eric.has_ball = 1;
    cond_signal(&eric.block);
    mutex_unlock(&eric.lock);

    while(!exam_over) {

        mutex_lock(&paul.lock);
        while(!paul.has_ball) {
            cond_wait(&paul.block, &paul.lock);
        }
        mutex_lock(&eric.lock);
        eric.has_ball = 1;
        cond_signal(&eric.block);
        mutex_unlock(&eric.lock);

        paul.has_ball = 0;
        mutex_unlock(&paul.lock);

        exam_over = check_exam_room();
    }

    return (0);
}

void *run_paul(void *arg) {

    while(!exam_over) {

        mutex_lock(&eric.lock);
        while(!eric.has_ball) {
            cond_wait(&eric.block, &eric.lock);
        }
        mutex_lock(&paul.lock);
        paul.has_ball = 1;
        cond_signal(&paul.block);
        mutex_unlock(&paul.lock);

        eric.has_ball = 0;
        mutex_unlock(&eric.lock);
    }

    return (0);
}
```

There is a problem with this code. Briefly state (e.g., one to three sentences), which TA(s) can "get stuck" and why. Then write an execution trace which backs up your claim.

Suggestions for working on this problem:

1. When tracing the execution of the code, we recommend a tabular format very similar to this:

Eric	Paul
	wait(E)
lock(E)	
...	
signal(E)	
unlock(E)	

2. It is *strongly recommended* that you write down a draft version of any execution trace using the scrap paper provided at the end of the exam, or on the back of some other page, before you start to write your solution on the next page. If we cannot understand the solution you provide, your grade will suffer!
3. Be sure that your trace can actually happen, and that it convincingly shows the phenomenon you wish to demonstrate. For example, if you are claiming that the system can indefinitely repeat a series of states, you should probably show the series twice, and you should *definitely* make it clear exactly which steps are the ones that repeat.

Use this page for your solution to the throwing question.

You may use this page as extra space for the throwing question if you wish.

3. [15 points] Deadlock.

Cluster Services wants to know how students are using the clusters for their classes. They are particularly interested in OS students, as students in OS use the clusters often, and often remain for many hours in a row. So they conduct a brief survey of OS in which they ask OS students about their work patterns. Here is what they found.

OS students work closely together in pairs. As such, they must find two adjacent seats. But often the clusters are packed with all sorts of people and it's hard to find two adjacents seats that are open. So the OS students do the following. Each pair will come to the cluster and will try to find an open seat. If one is available, one student of the pair will go and sit down in the seat. Now the other student of the pair will wait until a seat which is adjacent to the first seat becomes available. Once one does, that student will sit down in the seat adjacent to his or her partner, and the pair will resume working on their kernel.

The Cluster Services staff naturally wish to avoid deadlock conditions in their clusters! Because they're not sure how to analyze all the possibilities, you have been given this exam question so you can help out.

(a) [10 points] For this part of the question, assume each cluster has exactly one table, which is circular and has at least three seats. This means that any given seat has two adjacent seats: the one to its left and the one to its right. Assume further that the cluster being analyzed is used by *only* OS students, who always follow the procedure described above. Can the OS students deadlock?

If so, provide either an execution trace (using the tabular trace format described in the previous problem) or a drawing of a process/resource graph (with sufficient annotation to make the situation clear). If deadlock is *not* possible, provide a clear and concise argument that it cannot happen; your reasoning should be convincing enough that it could be included with the code as documentation, and should probably mention one or more of the four "deadlock ingredients" by name.

You may use this page for the deadlock question.

Now assume that the OS students employ a different procedure. After one student of a pair acquires an initial seat, the second seat the other student tries to acquire is *not* adjacent to the first seat. Instead, the second student will acquire *any* open seat in the cluster. Once a pair has acquired two seats (which are now not guaranteed to be adjacent), they will ask people in the cluster to move around so that the pair end up with seats that are adjacent. Assume that there are an even number of seats for any table in any given cluster, and that it is always possible to rearrange everyone so that all OS pairs who have two seats will be in adjacent seats. Also assume everyone in the cluster is nice enough to comply to this request.

(b) [5 points] Given this updated procedure, and still with the assumption that *only* OS students are using the clusters, can OS students deadlock? As above, be sure that your answer is clear and convincing.

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

4. [20 points] Banking

Banks automate a large portion of their systems, and often must deal with concurrency when doing so. For this question, you will explore the synchronization for various banking-related operations. Accounts at this bank support, between when they are set up and when they are closed, the standard "deposit" and "withdraw" operations and the standard "transfer money between accounts" operation. These operations "quickly" succeed or fail: while they may need to wait briefly to acquire locks, they complete (one way or another) in a brief period of time. On an experimental basis, this bank also supports an unusual operation, "stubbornly withdraw." If this operation finds that the account doesn't contain enough money, instead of failing right away it takes additional actions (which may require quite a lot of time!) to wait for enough money to arrive in the account. This operation could take "quite a while"—in fact, it could take so long that the account in question is closed. Because money will never be added to a closed account, "stubbornly withdraw" does give up and return an error code in this situation.

The documentation for these operations appears below. For exam purposes, all amounts may be represented in whole dollars (no cents or fractional cents). Also for exam purposes, *the operations below may all assume that account pointers which are passed in point to valid memory; all operations other than* `account_init()` *may assume that account pointers point to valid memory containing an account struct that has been initialized exactly once and never closed.* Finally, you may assume that once `account_close()` returns, the memory holding the former account struct will not be re-used until the system reboots.

```
/** @brief Initializes a new account (to contain $0). */
void account_init(account_t *account);

/**
 * @brief Deposits a sum of money into an account
 *
 * The amount of money specified is added to this account's balance.
 */
void deposit(account_t *a, int amt);

/**
 * @brief Withdraws a sum of money from an account
 *
 * If the specified account has enough money in it, then the money is withdrawn.
 * Otherwise, the function returns an error code without waiting to see if new
 * money might become available.
 *
 * @return 0 if the money was successfully withdrawn, -1 if the money could not
 *         be withdrawn at this time
 */
int withdraw(account_t *a, int amt);
```

```
/**
 * @brief Transfers money between accounts
 *
 * Transfers the specified amount of money from the account 'a' into the account
 * 'b'.  If 'a' does not have sufficient funds for the transfer, the function
 * will fail (it will *not* block waiting for more money to appear in 'a').
 *
 * @return 0 if the money was transferred successfully, or -1 if there are
 *         insufficient funds.
 */
int transfer(account_t *a, account_t *b, int amt);

/**
 * @brief Withdraws some money from an account, blocking until it's available.
 *
 * If there is not enough money in the account at the time of withdrawal, this
 * function will block until the money becomes available.  It should never be
 * the case that a thread is blocked indefinitely in stubbornly_withdraw() if
 * there is enough money in the account for the thread to withdraw.
 *
 * When this operation is invoked, the account pointer points to a valid account
 * which has not been closed.  However, the account might become closed while a
 * thread is running this operation.
 *
 * @return 0 if the funds were withdrawn, -1 if the account was closed before
 *         the funds could be withdrawn
 */
int stubbornly_withdraw(account_t *a, int amt);

/**
 * @brief Closes an account
 *
 * After an account is closed, it may no longer be used by any of the previous
 * functions.  Each account will be closed no more than once.
 *
 * @return the amount of money left in the account before it was closed.
 */
int account_close(account_t *a);
```

Your mission

You will implement `struct account` with the previous six methods. They are listed here again to refer back to:

```
void account_init(account_t *account);
void deposit(account_t *a, int amt);
int withdraw(account_t *a, int amt);
int transfer(account_t *a, account_t *b, int amt);
int stubbornly_withdraw(account_t *a, int amt);
int account_close(account_t *a);
```

Your solution can use Project 2 thread library mutexes, condition variables, and/or semaphores, which you may assume to be strictly-FIFO if you wish. You may use `deschedule()`/`make_runnable()` if you must, though we don't recommend it; otherwise, you may not use other atomic or thread-synchronization operations, such as, but not limited to: reader/writer locks or any atomic instructions (`XCHG`,`LL/SC`). You may use various system calls not prohibited above, e.g., `get_ticks()`, if you wish, and non-synchronization-related thread-library routines in the "`thr_xxx()` family," e.g., `thr_getid()`. **For the purposes of the exam you should assume an error-free environment (memory allocation and initialization functions will always succeed; system calls and thread-library primitives will not detect internal inconsistencies or otherwise "fail"** (unless you indicate in your comments that you have arranged, and are expecting, a particular failure), etc.). You may wish to refer to the "cheat sheets" at the end of the exam.

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 on the next page, your grade will suffer!

Hint: the design and synchronization primitives necessary for implementing just `deposit()` and `withdraw()` may be different than what you need for implementing the other operations, so it is probably unwise to start writing code for the simple operations until you have designed the harder ones!

Please declare a `struct account` and implement `account_init()`, `deposit()`, `withdraw()`, `transfer()`, `stubbornly_withdraw()`, and `account_close()`.

```
typedef struct account {









} account_t;
```

...space for account implementation...

...space for account implementation...

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

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

5. [15 points] `Nuts & Bolts`.

Your OS partner is excited to begin the kernel project and is also excited about the many language-extension features provided by the GNU C compiler, gcc. Your partner wants to use as many of the gcc extensions as possible, and believes that context switch could be a place to use a whole bunch of them, especially ones that enable getting and setting CPU registers in the middle of a C function. You believe that, within the body of a C function, the compiler defines and manages the meaning of each of the registers, but your partner's attitude is "What could possibly go wrong?"

You decide to write down the simplest example you can, which uses just one register and doesn't even use any weird compiler features. You begin by writing a fairly legitimate function, in assembly language, that fetches the value of the stack pointer.

```
.global get_esp
get_esp:
    movl %esp, %eax
    ret
```

You then write a sample program which uses `get_esp()`.

```
void print_int(int n) {
    printf("%d\n", n);
}

int delta(void) {
    int x = get_esp();
    print_int(4);
    int y = get_esp();
    return (x - y);
}

int main(int argc, char *argv[]) {
  printf("Delta value = %d\n", delta());
  return (0);
}
```

According to your OS partner, "obviously" the delta value printed should always be zero. But you believe that by playing around with compiler flags or trying different versions of the compiler it should be possible to observe other delta values; you hope that, by proving that the values of registers are unpredictable, your partner will realize it's unwise to examine (let alone change!) them. In this exam question, your mission will be to write down assembly code for two versions of the `delta()` function which are both plausible ways for a compiler to write the function but which will result in different delta values being printed.

(a) [10 points] Write one plausible version of the `delta()` function in assembly language. State the delta value that will be printed. You should *not* write code for `print_int()` or `main()`.

(b) [5 points] Now write a second plausible version of the `delta()` function in assembly language. This one should produce a different (plausible) delta value. State the delta value that will be printed. You should *not* write code for `print_int()` or `main()`.

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.