Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Spring 2019

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.

If we cannot read your writing, we will be unable to assign a high score to your work.

1. [10 points] Short answer.

   (a) [6 points] Explain the "three kinds of error." For each, provide a name and/or brief description and describe in a general high-level sense what should be done in response to that kind of error. Explain why that is the thing that should be done about that kind of error. We are expecting approximately two sentences for each kind.

(b) [4 points] Register dump.

Below is a register dump produced by the "Pathos" P2 reference kernel when it decided to kill a user-space thread. Your job is to carefully consider the 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 plausibly 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.

```
Registers:
eax: 0x00000001, ebx: 0x00000000, ecx: 0x00000000,
edx: 0x00000000, edi: 0x00000000, esi: 0x00000000,
ebp: 0xffffefec, esp: 0xffffefec, eip: 0x0010003c,
 ss:     0x002b,  cs:     0x0023,  ds:     0x002b,
 es:     0x002b,  fs:     0x002b,  gs:     0x002b,
eflags: 0x00010282
```

You may use this page for the register-dump question.

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 thread 0 is running this code, $i == 0$ and $j == 1$; when thread 1 is running this code, $i == 1$ and $j == 0$, so i means "me" and j means "the other thread."
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."

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.,

T0	T1
w[0] = 1;	
	w[1] = 1;

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.

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] "Mockchain" deadlock

After hearing a constant stream of arguments about how blockchain is the greatest technical innovation since the invention of the Internet, the 15-410 course staff has decided to build something similar, called Mockchain™—perhaps because it isn't exactly the same as more-popular blockchains, or perhaps for some other reason. Here's how it works.

- The Mockchain™ Consensus Network contains two types of participants: "miners" and "clients." Miners are responsible for reaching a consensus about which transactions are to be recorded on the mockchain. Clients generate the transactions and send them off to miners to be appended to the chain.
- When a miner receives a new transaction from a client, it packages it into a block and sends the block to other miners to be "validated." In a real blockchain, a block would likely contain many transactions, but in the interest of simplicity, Mockchain blocks contain just a single transaction. A block is considered "validated" once a majority of miners on the network have approved the block. This means that the main responsibilities of the miner are twofold:
  - Reacting to new transactions that clients have sent to the miner. Each transaction will be packed into a block and proposed to all other miners on the network for validation.
  - Validating the blocks that other miners on the network have proposed. For the sake of this problem, you won't need to worry much about how blocks are actually validated. Assume that a `seems_legit()` function handles the validation step appropriately.
- When a client sends a transaction request to a miner, it blocks until the request has been processed. At this point, the transaction has either been accepted or rejected. In other blockchain systems, a client may choose to retry a transaction if it was rejected, but in this problem all that happens is that the success/failure outcome is printed.

To investigate how successful this mockchain protocol could be, the 410 staff has written some simulation code to model the interactions between clients and miners.

Some important notes about the simulation code:

- Since this is an exam setting, assume that functions such as `thr_init()`, `thr_create()`, `malloc()`, `mutex_init()`, `enq()`, etc., never fail.
- Assume that any code which is not shown is entirely correct. *In particular, this means that your explanation of any bugs you may find should not rely on unspecified or assumed behavior of functions for which we did not explicitly provide the implementation.*
- The function clients use to generate their transactions is called `some_random_transaction()`. You don't need to know what most of it does. *However, you must assume that it initializes the semaphore of the returned transaction with a count of 0.*
- You'll find that the `main()` function is not very interesting. It simply initializes some miners, then launches threads to serve as miners and clients.

Sometimes, the simulation seems to "get stuck". Help the 15-410 staff debug the Mockchain simulator by examining the code provided on the subsequent pages.

```
#define NUM_CLIENTS 10
#define NUM_MINERS  4
#define REQUIRED_VOTES ((NUM_MINERS / 2) + 1) // majority consensus

typedef struct {
    void *data;
    bool accepted;
    sem_t finished; // initialized to 0 by some_random_transaction()
} transaction_t;

typedef struct {
    queue_t incoming_transactions;
    queue_t pending_validation;
    cond_t cv;
    mutex_t mutex;
} miner_t;

typedef struct  {
    void *data;
    unsigned int approval_count;
    unsigned int reply_count;
    mutex_t mutex;
    miner_t *owner;
} block_t;

/* array of all miners on the network */
miner_t miners[NUM_MINERS];

/* ------- Assume the below functions exist elsewhere ------- */

void qinit(queue_t *q);
void enq(queue_t *q, void *block);
int isempty(queue_t *q);
void *deq(queue_t *q); // don't call if isempty()

/* called by miner to check if a block should be accepted */
extern bool seems_legit(block_t *b);

/* called by client to generate a new transaction;
 * assume transaction semaphore is initialized with a count of 0
 */
extern transaction_t *some_random_transaction(void);
```

```
/* create a new block */
block_t *block_new(miner_t *owner, void *data) {
    block_t *block = calloc(1, sizeof(block_t));
    block->data = data;
    block->owner = owner;
    mutex_init(&block->mutex);
    return block;
}

/* respond to a request to validate a proposed block */
void validate_block(block_t *block) {
    mutex_lock(&block->mutex);
    block->reply_count++;
    if (seems_legit(block))
        block->approval_count++;
    mutex_unlock(&block->mutex);
    cond_signal(&(block->owner->cv));
}

/*
 * propose a new block to be appended to Mockchain;
 * blocks until other miners accept/reject proposal
 */
bool propose_block(miner_t *me, block_t *block) {
    /* broadcast proposal to all the miners */
    for (int i = 0; i < NUM_MINERS; i++) {
        /* don't send a request to ourself */
        if (&miners[i] == me) continue;
        mutex_lock(&miners[i].mutex);
        enq(&miners[i].pending_validation, block);
        cond_signal(&miners[i].cv);
        mutex_unlock(&miners[i].mutex);
    }
    /* approve our own block and wait for others to respond */
    mutex_lock(&block->mutex);
    block->approval_count++;
    block->reply_count++;
    while (block->reply_count < NUM_MINERS &&
           block->approval_count < REQUIRED_VOTES) {
        cond_wait(&me->cv, &block->mutex);
    }
    bool accepted = block->approval_count >= REQUIRED_VOTES;
    mutex_unlock(&block->mutex);
    return accepted;
}
```

```
void *run_miner(void *arg) {
    miner_t *me = (miner_t *) arg;
    mutex_lock(&me->mutex);

    while (1) {
        /* wait for some work to do */
        while(isempty(&me->pending_validation) &&
              isempty(&me->incoming_transactions)) {
            cond_wait(&me->cv, &me->mutex);
        }
        /* validate any blocks others may have proposed */
        while (!isempty(&me->pending_validation)) {
            block_t *new_block = deq(&me->pending_validation);
            validate_block(new_block);
        }
        /* accept new transactions from clients */
        while (!isempty(&me->incoming_transactions)) {
            transaction_t *t = deq(&me->incoming_transactions);
            mutex_unlock(&me->mutex);
            block_t *block = block_new(me, t->data);
            t->accepted = propose_block(me, block);
            sem_signal(&t->finished);
            mutex_lock(&me->mutex);
        }
    }
    return NULL;
}

void *run_client(void *arg) {
    while (1) {
        sleep(genrand() % 1000);
        transaction_t *t = some_random_transaction();
        miner_t *m = &miners[genrand() % NUM_MINERS];
        mutex_lock(&m->mutex);
        enq(&m->incoming_transactions, t);
        mutex_unlock(&m->mutex);
        cond_signal(&m->cv);
        sem_wait(&t->finished);
        if (t->accepted)
            printf("Transaction accepted :)\n");
        else
            printf("Transaction failed :(\n");
    }
    return NULL;
}
```

```
void miner_init(miner_t *m)
{
    qinit(&m->incoming_transactions);
    qinit(&m->pending_validation);
    cond_init(&m->cv);
    mutex_init(&m->mutex);
}

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

    /* exam: assume nothing fails */
    thr_init(PAGE_SIZE);
    sgenrand(get_ticks());

    for (int i = 0; i < NUM_MINERS; i++)
        miner_init(&miners[i]);
    for (int i = 1; i < NUM_MINERS; i++)
        thr_create(run_miner, (void*) &miners[i]);
    for (int i = 0; i < NUM_CLIENTS; i++)
        thr_create(run_client, NULL);

    run_miner((void*) &miners[0]);
    return 0;
}
```

(a) [10 points] Unfortunately, the code shown above can deadlock. Show *clear, convincing* evidence of a deadlock. Begin by *describing* the problem in one or two sentences; then *specify a scenario*, and finally show a *tabular execution trace*. Missing, unclear, or unconvincing traces will result in only partial credit. Your trace need not cover the operation of `main()`. You can use obvious notation, e.g., "`m0`" can be "miner thread 0," and it is ok to simply mention a function running to completion in the obvious way without going through every internal step, e.g., "`b_new(&m[0],...)`" can be a legitimate trace step for a thread.

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

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

(b) [5 points] Explain in detail how the 15-410 course staff could modify Mockchain to avoid deadlocks. Be sure to explain (in a theoretical / conceptual sense) why your solution works. *Solutions judged as higher-quality by your grader will receive more points.* This means that it is probably better to genuinely fix some problem than to replace a sensible assumption/parameter with an unrealistic assumption/parameter, though we will consider any solution you clearly describe. *We are not expecting you to show code, though you may if you wish.*

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

4. [20 points] "Double-condition" Variables.

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 design a new sync primitive called a double-condition variable ("dcond"). This new sync primitive is conceptually similar to having two separate condition variables and enabling users to wait on both of them. Specifically, when a thread calls `dcond_wait()` on a particular `dcond_t`, it should not be awakened until **two** corresponding `dcond_signal()` calls have been made: one to each of the logical underlying condition variables. The interface you are responsible for implementing is as follows:

- `int dcond_init(dcond_t *dc)` - initializes a double-condition variable. It is illegal for an application to use the double-condition variable before it has been initialized or to initialize a double-condition variable when it is already initialized and in use. `dcond_init` shall returns 0 on success or a negative error code on failure. 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 dcond_wait(dcond_t *dc, mutex_t *mp)` - The double-condition variable shall wait until **both** underlying logical condition variables have been signaled (`dcond_signal`). The mutex `mp` should be released when waiting and reacquired upon returning. *Note that the signals may arrive in either order*!
- `void dcond_signal(dcond_t *dc, int which)` - signal the single underlying condition variable specified by `s`. Note the `#define` values for `which` shown below.
- `void dcond_broadcast(dcond_t *dc, int which)` - broadcast to the single underlying condition variable specified by `s`. *You are not responsible for implementing this.*
- `void dcond_destroy(dcond_t *dc)` - destroy a double-condition variable. *You are not responsible for implementing this.*

Note the following definition for the `which` parameter:

```
#define CVAR_0 0
#define CVAR_1 1
```

The following trace should illustrate the expected behavior of a double-condition variable.

The remainder of this page is intentionally blank.

Time	Thread 1	Thread 2	Comments
1	`dcond_wait(dc, mp)`		
2		`dcond_signal(dc, CVAR_0)`	Signal the first logical cvar
3		`dcond_signal(dc, CVAR_0)`	Does **not** wake Thread 1!
4		`dcond_signal(dc, CVAR_1)`	
5			Thread 1 is awakened
6		`dcond_wait(dc, mp)`	
7	`dcond_signal(dc, CVAR_1)`		
8	`dcond_signal(dc, CVAR_0)`		
9			Thread 2 is awakened

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) [5 points] Please declare your `dcond_t` here. If you need one (or more) auxilary structures, you may declare it/them here as well.

```
typedef struct {

} dcond_t;
```

(b) [15 points] Now please implement `int dcond_init()`, `void dcond_wait()` and `void dcond_signal()`.

. . . space for your double-condition variable implementation . . .

. . . space for your double-condition variable implementation . . .

. . . space for your double-condition variable implementation . . .

5. [10 points] Scheduler states

If we examine the scheduler's data structures on a machine and observe two threads, one that is runnable and one that is blocked, typically the runnable thread entered the kernel via an interrupt and the blocked thread entered the kernel via a trap—but that might not necessarily be true!

(a) [3 points] Describe a plausible scenario in which a thread that is currently *runnable* entered the kernel via *an interrupt*—or argue that this is not a plausible situation.

(b) [2 points] Describe a plausible scenario in which a thread that is currently *runnable* entered the kernel via *a trap*—or argue that this is not a plausible situation.

(c) [3 points] Describe a plausible scenario in which a thread that is currently *blocked* entered the kernel via *a trap*—or argue that this is not a plausible situation.

(d) [2 points] Describe a plausible scenario in which a thread that is currently *blocked* entered the kernel via *an interrupt*—or argue that this is not a plausible situation.

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 */
```

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.