Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Fall 2022

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.
3. **PLEASE DO NOT WRITE FAINTLY WITH PENCIL. Please write in ink, or, if writing in pencil, please ensure that zero strokes in zero words are faint. Using a mechanical pencil with thin lead is probably unwise.**
4. This is a closed-book in-class exam. You may not use any reference materials during the exam.
5. **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 ...?"**
6. 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.
7. Please be concise in your answers. You will receive partial credit for partially correct answers, but truly extraneous remarks may count against your grade.

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) [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. You should say why/how the wrong value led to an exception, i.e., merely claiming a register has a "wrong" value will not receive full credit.
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).

Please be sure that your description of the fatality and the code, taken together, clearly support your diagnosis.

```
Registers:
eax: 0x00000000, ebx: 0x00000000, ecx: 0x0000000a,
edx: 0x0000000d, edi: 0x00000000, esi: 0x00000000,
ebp: 0x00401404, esp: 0xffffefc8, eip: 0x0100005c,
 ss:     0x002b,  cs:     0x0023,  ds:     0x002b,
 es:     0x002b,  fs:     0x002b,  gs:     0x002b,
eflags: 0x00000282
```

You may use this page for your register-dump answer.

(b) [6 points] In lecture we discussed two ways that a virtual memory system could implement least-recently-used page replacement (don't worry, this isn't a virtual memory question!). The problem statement is: every time a page is referenced, an operation `reference(page_number)` is invoked; every now and then the page daemon invokes `lru_which(void)`, which returns the number of the page which was *least* recently referenced (note that calling `lru_which()` twice in a row should generally return two different page numbers, not the same page number twice).

The two approaches we discussed were a doubly-linked list of blobbies (where each blobby consisted of previous and next pointers and a page number) and a big array of per-page timestamps.

In lecture we examined a design matrix in accordance with the 15-410 design orthodoxy, including two time metrics.

> Present a design matrix for these two approaches, including the two time metrics we discussed *and at least one additional metric*. Also, though in lecture we added a third implementation approach, *do not do that now*. Instead, present a justification for choosing either the blobby-queue approach or the timestamp approach.

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

2. [15 points] Faulty Condition Variables

As you may have noticed, outside the Cohon University Center many stones are imprisoned in cages. As leader of the "Free the UC Stones" movement sweeping Carnegie Mellon, you are running a covert operation in which a team of threads cooperates to insert and remove virtual stones into and out of a bucket. Each team is configured with a balanced set of parameters—maybe one insertion thread adds 20 stones, while two removal threads remove 10 stones each; or maybe two insertion threads each add 15 stones, while one removal thread removes 30 stones. Meanwhile, the size of the bucket (buffer) is an independent parameter: maybe it's a 5-stone bucket/buffer, maybe it's 10 stones—whatever. Of course you recognize this as an instance of the classic "producer/consumer" prpoblem, and write up some sample code using condition variables.

Meanwhile, your OS partner recently attended a tech talk given by a company called Purple Valley. Purple Valley has a real-time micro-kernel which makes kernel-implemented semaphores available to user-space programs via `sem_wait()` and `sem_signal()` primitives. Your partner bridges the gap between your cvar-based code and the semaphore-based kernel by coding up some semaphore-based cvars; the semaphores are used as a tool to block and unblock threads.

Of course, something goes wrong, and your partner is asleep, so you need to find the problem. You are to assume that the producer/consumer code shown below is correct, and also assume that the Purple Valley real-time micro-kernel semaphores are correct, i.e., you are to assume the problem is in your partner's cvar code.

Refer to the following code for the producer-consumer problem. Again, assume it works correctly.

The remainder of this page is intentionally blank.

```
#define BUCKET_SIZE 5       // or something else
#define LIFTING_SLEEP 7     // or something else
#define DISPOSAL_SLEEP 9    // or something else

static mutex_t count_lock; // correctly initialized elsewhere
static int nstones = 0;
static cond_t removed;     // correctly initialized elsewhere
static cond_t added;       // correctly initialized elsewhere

void *insertion_thread(void *vstones) {
    int stones = (int) vstones;
    for (int s = 0; s < stones; s++) {
        /* Pick up a stone.  Heavy!  Takes a while. */
        sleep(LIFTING_SLEEP);
        /* Now wait until there is an empty slot, insert stone, notify. */
        mutex_lock(&count_lock);
        while (nstones == BUCKET_SIZE)
            cond_wait(&removed, &count_lock);
        ++nstones;
        cond_signal(&added);
        mutex_unlock(&count_lock);
    }
    return (0);
}

void *removal_thread(void *vstones) {
    int stones = (int) vstones;
    for (int s = 0; s < stones; s++) {
        /* Wait for a stone, claim it, notify. */
        mutex_lock(&count_lock);
        while (nstones == 0)
            cond_wait(&added, &count_lock);
        --nstones;
        cond_signal(&removed);
        mutex_unlock(&count_lock);
        /* Dispose of stone.  Heavy!  Takes a while. */
        sleep(DISPOSAL_SLEEP);
    }
    return (0);
}
```

Now refer to the condition-variables code below—do *not* assume this code works correctly! Note that certain functions, such as `cond_broadcast()` and `cond_destroy()`, are not relevant to this problem, and so their implementation has been omitted.

```
/* cond.c */

typedef struct cond {
  mutex_t waiters_lock;
  int waiters;
  sem_t s;
} cond_t;

int cond_init(cond_t *cv) {
    mutex_init(&cv->waiters_lock);
    sem_init(&cv->s, 0); // 0 for blocking purposes
    cv->waiters = 0;
    return (0);
}

void cond_wait(cond_t *cv, mutex_t *mp) {
    mutex_unlock(mp);
    mutex_lock(&cv->waiters_lock);
    ++cv->waiters;
    mutex_unlock(&cv->waiters_lock);
    sem_wait(&cv->s); // block
    mutex_lock(mp);
}

void cond_signal(cond_t *cv) {
    mutex_lock(&cv->waiters_lock);
    if (cv->waiters) {
        --cv->waiters;
        sem_signal(&cv->s); // unblock one
    }
    mutex_unlock(&cv->waiters_lock);
}
```

(a) [12 points] Depending on the exactly how/when the insertion and removal threads invoke the `cond_signal()` and `cond_wait()` shown above, one or more condition variables end up working incorrectly and insertion and/or removal threads can get stuck. Please construct a trace of how a thread can get stuck. If possible, try to use one insertion thread (called "I") and one extractor thread (called "E"), though we will consider traces involving other sets of threads. Use the execution trace format presented in class, e.g.,

I	E
–waiters	
	++waiters

At the bottom of your trace, summarize in a sentence or two what "the problem" is.

You may use this page for your cvar trace if you wish.

(b) [3 points] The UC Stones can be freed only with your help! Write code that replaces the area of problematic implementation which you identified in the condition variable code. We are expecting under 15 lines of code (though we will consider any solution you provide).

3. [15 points] Deadlock.

Over lunch with a friend you find yourself reminiscing about your recently-completed summer internship at WhatSnappyChatter, a social-messaging company which was bought some years back by a large social-media company for $14 quintillion. The WhatSnappyChatter division had a problem which required the services of a hotshot CMU student to debug it.

The company's "app" has some strange architectural features. For example, the app has a small fixed number of threads, and `thr_getid()` has been hacked so it always returns a value between 0 and `NUM_THREADS-1`. You were working on an internal messaging system that enables these threads to send and receive small messages with each other (for the purposes of this exam, we will assume that the value contained in each message is an `int`—boring, but simple). For some reason which isn't clear to you (maybe a patent??), a core feature of the messaging system is that send calls block until the data has been received by the receiver thread, but it is *very important* for receive operations to return *very quickly* whether or not anything is waiting.

Here are some details about the implementation:

- You may assume that there are only `NUM_THREADS` threads using this system and their thread id's range between 0 and `NUM_THREADS-1`.
- `send_message(msg, to)` should be used to send data (in this case an int) to another thread, and should block until the recieving thread calls `recv_message()`.
- Each thread is given an "inbox" and an "outbox" in two global arrays; each thread's inbox and outbox are located by using the owning thread's id as an index.
- If thread i wishes to send a message to thread j, it will first put the data into its outbox, and then put its thread id in thread j's inbox.
- For thread j to receive this message, it will check its own inbox, see i's thread id, and then find the data in thread i's outbox. At this point the message has been passed successfully.
- In order to avoid deadlock, a thread is not allowed to send a message to itself.
- In order to avoid deadlock, `send_message()` will return an error if thread i tries to send a message to thread j while it already has a pending message from thread j (i.e. thread j is already waiting for thread i to pick up a message it sent).

When reading the code below, you should assume all library and system calls return normally, and the `mailboxes_init()` call is successful.

The remainder of this page is intentionally blank.

```
#define NUM_THREADS 10

typedef struct {
    sem_t sender_lock;
    mutex_t data_lock;
    int from;
} inbox;

typedef struct {
    sem_t send_complete;
    int msg;
} outbox;

inbox inboxes[NUM_THREADS];
outbox outboxes[NUM_THREADS];

int mailboxes_init()
{
    int i;
    for (i = 0; i < NUM_THREADS; i++)
    {
        // assume initialization functions cannot fail
        sem_init(&inboxes[i].sender_lock, 1);
        mutex_init(&inboxes[i].data_lock);
        inboxes[i].from = -1;

        sem_init(&outboxes[i].send_complete, 0);
        outboxes[i].msg = 0;
    }
    return 0;
}
```

The remainder of this page is intentionally blank.

```
int recv_message(int *msg)
{
    int my_tid = thr_getid();

    mutex_lock(&inboxes[my_tid].data_lock);
    int sender = inboxes[my_tid].from;
    mutex_unlock(&inboxes[my_tid].data_lock);

    if (sender != -1)
    {
        inboxes[my_tid].from = -1;
        *msg = outboxes[sender].msg;
        sem_signal(&outboxes[sender].send_complete);
        return sender;
    }
    return -1;
}

int send_message(int msg, int to)
{
    int my_tid = thr_getid();

    // DL: a thread is not allowed to send messages to itself
    if (to == my_tid)
    {
        return -1;
    }

    outboxes[my_tid].msg = msg;

    sem_wait(&inboxes[to].sender_lock);
    mutex_lock(&inboxes[to].data_lock);

    inboxes[to].from = my_tid;

    mutex_unlock(&inboxes[to].data_lock);

    // DL: fail to send message if recipient is already waiting on us to read
    mutex_lock(&inboxes[my_tid].data_lock);
    if (inboxes[my_tid].from == to)
    {
        inboxes[to].from = -1;
        mutex_unlock(&inboxes[my_tid].data_lock);
        sem_signal(&inboxes[to].sender_lock);
        return -1;
    }
    mutex_unlock(&inboxes[my_tid].data_lock);

    sem_wait(&outboxes[my_tid].send_complete);
    sem_signal(&inboxes[to].sender_lock);

    return 0;
}
```

You were called in because of a disagreement in the development team. In particular, the author of the messaging library claimed that, no matter how the library is used by its client threads, it won't "internally deadlock." In other words, either the send and receive operations will complete in a reasonable way, or at least one thread invoking `send_message()` will receive a return code of -1.

Unfortunately, the code shown above *can* deadlock. Show *clear, convincing* evidence of deadlock. Your evidence should include a "tabular execution trace," a well-annotated process/resource graph, or both. Missing, unclear, or unconvincing traces will result in only partial credit. Note that `send_message()` returning an error code does *not* count as deadlock.

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

4. [20 points] Starvation.

Naturally you aspire to serve as a teaching assistant for this class after completing it. You recall that while your group was implementing readers/writers locks it was challenging to ensure that writers weren't starved by readers. You suspect that maybe some other groups didn't manage to ensure this important property. So you figure you can demonstrate your TA capabilities by writing a test program, perhaps useful in an autograder context, that can be linked and run against a student rwlock implementation of unknown quality to probe whether or not readers can starve writers.

Of course, it is extremely difficult for a simple test program to verify that rwlocks *never* starve writers, but the goal of this test code is to catch some/many implementations that have some tendency to starve writers. Writing test code is difficult because target code can be broken in so many ways. For example, if `rwlock_init()` reliably trashes the caller's stack, your test code will naturally crash or hang before being able to form an opinion about writer starvation. Thus you should probably assume that the rwlock implementation you will be testing is either completely correct in every way, including never starving writers, or that it is completely correct in every way *except* that it starves writers.

Your program should, with very high probability, print the string "FAIL" if the rwlock implementation you are testing starves writers, and should, with very high probability, print the string "PASS" if it does not (if your program prints multiple "PASS" or "FAIL" strings, that is acceptable as long as all are "PASS" or all are "FAIL", i.e., no mixtures!). Your program should make its decision in less than ten seconds as long as reasonable assumptions hold true (for example, if the code you are testing is broken and that causes your test code to hang instead of producing an answer, that's ok). It is also permissible for your program to print "FAIL," or to invoke `assert()` or `panic()`, if it detects an impossible outcome, though you are *not* required to check for inconsistencies (see below). What your test program does after printing its verdict doesn't matter: your test program is allowed to hang, crash, etc., because the printed answer will be reviewed by a live human being or else by a test-harness driver.

Other assumptions and restrictions:

1. Your test code can use these regular Project 2 thread-library primitives: mutexes, condition variables, and/or semaphores (obviously it must use rwlocks). You may not use the `deschedule()` or `make_runnable()` system calls.
2. You may assume that mutexes, condition variables, and semaphores are correctly implemented. If you wish, you may state assumptions about specific fairness properties of mutexes, condition variables, and/or semaphores.
3. Note that, as an interface-compliant test program, *you cannot inspect or modify the internals of any thread-library data objects*–because your test code must run against every student thread library, you are strictly a *client* of the abstractions provided by each library. Because this is test code, you *may* use normally-distasteful code constructs such as yield loops or even spin-waiting if you must; however, solutions with smaller quantities of distasteful code are likely to receive higher scores.
4. **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).

5. 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.
6. You may wish to refer to the "cheat sheets" at the end of the exam.

Because you think some day your test might test multiple rwlocks in parallel, you decide to encapsulate the housekeeping data structures (e.g., rwlocks, condition variables, mutexes, ints, chars, doubles, ...) used by your test in a `struct test`; in the interests of keeping the test code simple, you decide you will try to limit the struct to containing six fields.

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 here. If we cannot understand the solution you provide on this page, your grade will suffer!

The remainder of this page is intentionally blank.

(a) [5 points] Please declare your `struct test` here. Also write a function `void test_init(test_p np)` to initialize a test.

```
typedef struct test {









} test_t, *test_p;

void test_init(test_t *tp)
{









}
```

(b) [15 points] Now please write your test code. The suggested structure is one to three small helper functions and a `main()` function.

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

5. [10 points] `Nuts and bolts`

   (a) [4 points] Write a sequence of up to five x86-32 instructions which, when run, will have the effect of placing the address of any one of the instructions (you get to pick which one) into `%eax`. It is OK for you to destroy the values in other registers.

(continued on next page)

(b) [6 points] Consider the following C code:

```
int g;                /* 1 */

void foo() {
    int i=1;          /* 2 */
    static int s;     /* 3 */
}

void bar() {
    static int s=1;   /* 4 */
}

int main(int argc, char **argv) /* 5, 6 */
{
    return 0;
}
```

For each variable, indicate with a check mark where in memory it is stored at runtime. Assume `-O0`, where all variables are initially stored in memory rather than registers.

	text	data	bss	heap	stack
1. `g`					
2. `i`					
3. `s`					
4. `s`					
5. `argv`					
6. `argv[0]`					

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 */
```

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.

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.