Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (D), Fall 2018

1. **Please read the entire exam before starting to write. This should help you avoid getting bogged down on one problem.**
2. Be sure to put your name and Andrew ID below *and also* put your Andrew ID at the top of *each* following page.
3. This is a closed-book in-class exam. You may not use any reference materials during the exam.
4. If you have a clarification question, please write it down on the card we have provided. Please don't ask us questions of the form "If I answered like this, would it be ok?" or "Are you looking for ...?"
5. The weight of each question is indicated on the exam. Weights of question *parts* are estimates which may be revised during the grading process and are for your guidance only.
6. Please be concise in your answers. You will receive partial credit for partially correct answers, but truly extraneous remarks may count against your grade.
7. **Write legibly even if you must slow down to do so!** If you spend some time to *think clearly* about a problem, you will probably have time to write your answer legibly.

Andrew Username	
Full Name	

Question	Max	Points	Grader
1.	**10**		
2.	**10**		
3.	**15**		
4.	**20**		
5.	**15**		
	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.

I have not received advance information on the content of this 15-410/605 mid-term exam by discussing it with anybody who took part in the main exam session or via any other avenue.

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 should use Approach A or Approach B?" According to the 15-410 design orthodoxy, you should follow a specific process to resolve your question. Please describe that process, providing enough details and/or examples to demonstrate to your grader that you understand the concept and can apply it when necessary. Try to use specific examples rather than general terms.

(b) [5 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: 0xffffefdc, ebx: 0x00000000, ecx: 0x00000000,
edx: 0xffffefdc, edi: 0xfffff014, esi: 0x00000001,
ebp: 0xffffefe0, esp: 0xffffefc4, eip: 0xffffefdc,
 ss:     0x002b,  cs:     0x0023,  ds:     0x002b,
 es:     0x002b,  fs:     0x002b,  gs:     0x002b,
eflags: 0x00000202
```

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

2. [10 points] "Uplock" starvation.

As part of P2, you implemented a readers/writers lock that supported "downgrading": a thread holding a write lock can "partially release" the lock by switching to reader mode. In this exam question, we will ask you to consider a different kind of readers/writers lock, called an "uplock," that has an "upgrade" operation instead of a "downgrade" operation. In particular, the only way to obtain a writer-mode lock on an uplock is to first acquire a reader-mode lock and then upgrade to writer mode. The unlock operation, which may be invoked by a thread that holds either a reader-mode lock or a writer-mode lock, releases all claim on the lock. That is, `uplock_rdlock()` may be followed immediately by `uplock_unlock()`; the other legal sequence is `uplock_rdlock()`, then `uplock_wrlock()`, then `uplock_unlock()`. The "uplock" object supports the typical `uplock_init()` and `uplock_destroy()` operations, with typical semantics (e.g., it is not ok to invoke `uplock_destroy()` while an uplock is held or threads are waiting on one).

A small example program using an uplock is displayed on the next page. In the program, a single integer variable, `value`, is covered by an uplock. One reader thread repeatedly obtains the uplock in reader mode in order to fetch and print the value of the variable; meanwhile, the original thread repeatedly obtains the uplock in reader mode and upgrades to writer mode in order to modify the variable. For exam purposes, you should assume that the example program is correct and that all library calls made by the program succeed.

On the page after the example program is code for a *broken* uplock implementation. Your job will be to diagnose a bug.

The remainder of this page is intentionally blank.

```
#define STEPS 100
static uplock_t lock;
static volatile int value = 0;

static void *reader(void *arg) {
    (void)arg;

    int step = 0;
    while (step < STEPS) {
        uplock_rdlock(&lock);
        printf("%d\n", value);
        uplock_unlock(&lock);
        ++step;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    thr_init(4096);      // exam: cannot fail
    uplock_init(&lock);  // exam: cannot fail

    int tid = thr_create(reader, NULL);  // exam: cannot fail

    int step = 0;
    while (step < STEPS) {
        uplock_rdlock(&lock);
        uplock_wrlock(&lock);
        value = step;
        uplock_unlock(&lock);
        ++step;
    }
    thr_join(tid, NULL);   // exam: cannot fail
    uplock_destroy(&lock);
    thr_exit(NULL);
    return NULL;
}
```

Below is the problematic uplock implementation. Note that because this is "exam-mode code" you should assume that all correct invocations of thread-library primitives always succeed, and that all invocations of the uplock functions will be legal.

```
typedef struct uplock {
    mutex_t mutex;
    cond_t  writer_cond;
    cond_t  reader_cond;

    int    writing;    // initially: 0
    size_t wr_waiting; // initially: 0
    size_t rd_running; // initially: 0
} uplock_t;

int uplock_init(uplock_t *up) { // code omitted }
void uplock_destroy(uplock_t *up) { // code omitted }

void uplock_rdlock(uplock_t *up) {

    mutex_lock(&up->mutex);

    while (up->wr_waiting > 0 || up->writing) {
        cond_wait(&up->reader_cond, &up->mutex);
    }
    up->rd_running++;

    mutex_unlock(&up->mutex);
}

/* Warning: The caller MUST already hold the lock for reading.  */
void uplock_wrlock(uplock_t *up) {

    mutex_lock(&up->mutex);

    assert(up->rd_running > 0);
    up->rd_running--;
    up->wr_waiting++;

    while (up->writing || up->rd_running > 0) {
        cond_wait(&up->writer_cond, &up->mutex);
    }
    assert(up->wr_waiting > 0);
    up->wr_waiting--;
    up->writing = 1;

    mutex_unlock(&up->mutex);
}
```

```
void uplock_unlock(uplock_t *up) {

    mutex_lock(&up->mutex);

    if (up->writing) {
        assert(up->rd_running == 0);

        up->writing  = 0;
        int can_read = (up->wr_waiting == 0);

        mutex_unlock(&up->mutex);

        if (can_read) {
            cond_broadcast(&up->reader_cond);
        } else {
            cond_signal(&up->writer_cond);
        }
    } else {
        assert(up->rd_running > 0);
        up->rd_running--;

        mutex_unlock(&up->mutex);
        cond_signal(&up->writer_cond);
    }
}
```

First, briefly describe in words how this uplock implementation can lead to *starvation*. That is, say how some thread or class of thread can, while trying to obtain, upgrade, and/or release an uplock, can try an unbounded number of times without succeeding, while other threads or classes of thread can repeatedly obtain, upgrade, and/or release that uplock. Then present a trace which supports your claim. The starvation scenario you describe, and the trace you present, may be based on the small test program shown above, or may result from any *legal* uplock operations invoked by threads in some other program.

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 to your advantage to use scrap paper or the back of some page to experiment with draft traces, so that the answer you write below is easy for us to read.*

You may use this page for the uplock question.

3. [15 points] Parallel-sorting deadlock.

For this problem, we will be considering a parallel sorting algorithm, though not a particularly good one. The program provided seeks to sort a randomly-generated array of size `SLOTS`. It spools up `NTHREADS` threads, each of which runs for a fixed number of iterations. In each iteration, a thread attempts to acquire two different slots with indices `x` and `y`. After acquiring them, it swaps them if necessary, then releases them. While acquiring the first slot, the thread will block if it has already been acquired. For anti-deadlock purposes, while acquiring the second slot, the thread may decide to release the first slot and start over. Unfortunately, this sorting program can deadlock!

You will find that `main()` does not do anything particularly interesting: it initializes the thread library, `rand_lock`, and array, then creates and joins the worker threads. You will also find that `rand_int()` is not particularly interesting; it simply generates a random number in a thread-safe manner (`genrand()` is not thread-safe).

```
int main() {
    thr_init(4096); // exam: no failure
    sgenrand(get_ticks());
    mutex_init(&rand_lock); // exam: no failure

    for (int i = 0; i < SLOTS; i++) {
        mutex_init(&array[i].mtx); // exam: no failure
        cond_init(&array[i].cvar); // exam: no failure
        array[i].owner = -1;
        array[i].waiters = 0;
        array[i].value = rand_int();
    }

    int tids[NTHREADS];
    for (int i = 0; i < NTHREADS; i++)
        tids[i] = thr_create(sorter, (void *)i); // exam: no failure
    for (int i = 0; i < NTHREADS; i++)
        thr_join(tids[i], NULL); // exam: no failure

    int inversions = 0;
    for (int i = 0; i < SLOTS; i++) {
        for (int j = i+1; j < SLOTS; j++)
            if (array[i].value > array[j].value)
                inversions++;
        mutex_destroy(&array[i].mtx);
        cond_destroy(&array[i].cvar);
    }
    printf("inversions: %d\n", inversions);

    mutex_destroy(&rand_lock);
    thr_exit(0);
}
```

```
#define SLOTS 25
#define NTHREADS 20
#define ITERS 100

#define MAX(x,y) (((x) < (y)) ? (y) : (x))
#define MIN(x,y) (((x) < (y)) ? (x) : (y))

typedef struct {
    int owner;
    unsigned int value;
    int waiters; // bit-vector
    mutex_t mtx;
    cond_t cvar;
} slot_t;

static slot_t array[SLOTS];

static mutex_t rand_lock;
unsigned int rand_int() {
    mutex_lock(&rand_lock);
    int res = genrand();
    mutex_unlock(&rand_lock);
    return res;
}

void swap_slots(unsigned int x, unsigned int y) {
    int less = MIN(array[x].value, array[y].value);
    int more = MAX(array[x].value, array[y].value);
    array[x].value = x < y ? less : more;
    array[y].value = x < y ? more : less;
}

void release(int idx) {
    slot_t *s = &array[idx];
    mutex_lock(&s->mtx);
    s->owner = -1;
    mutex_unlock(&s->mtx);
    cond_broadcast(&s->cvar);
}
```

```
bool acquire(int desired_idx, int owned_idx, int id) {
    slot_t *desired = &array[desired_idx];
    slot_t *owned = owned_idx == -1 ? NULL : &array[owned_idx];

    int acquired = true;
    mutex_lock(&desired->mtx);

    if (desired->owner != -1) {
        desired->waiters |= (1 << id);
        while (desired->owner != -1) {
            if (owned && (owned->waiters & (1 << desired->owner))) {
                acquired = false;
                break;
            }
            cond_wait(&desired->cvar, &desired->mtx);
        }
        desired->waiters &= ~(1 << id);
    }

    if (acquired)
        desired->owner = id;

    mutex_unlock(&desired->mtx);
    return acquired;
}

void *sorter(void *arg) {
    int id = (int)arg;
    for (int iter = 0; iter < ITERS; iter++) {
        unsigned int x = rand_int() % SLOTS;
        unsigned int y = rand_int() % SLOTS;
        if (x == y) continue;

        acquire(x, -1, id); // first grab can't fail
        if (acquire(y, x, id)) {
            swap_slots(x, y);
            release(y);
        }
        release(x);
    }
    return NULL;
}
```

(a) [4 points] Show *clear, convincing* evidence of deadlock. Begin by *describing the problem* in one or two sentences; then *clearly specify a scenario*. Explicitly indicate how each necessary deadlock ingredient is present in the scenario you describe.

(b) [8 points] Now provide an execution trace resulting in a deadlock. 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 as extra space for the deadlock question if you wish.

(c) [3 points] Explain in detail (though code is *not* required!) how the program could be modified to not deadlock. 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.

4. [20 points] Targeted 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 implement a synchronization object called a "targetable condition variable" (abbreviated TCV). It is like a regular condition variable, with two key differences. First, TCVs support a "cancel" operation, which allows one thread to indicate that a specific other thread, identified by its thread i.d., should stop waiting and be given a particular cancellation code. Second, each time a thread waits on a TCV, the `wait()` operation returns a value. The return value will be zero when the wait ended because the condition became true, i.e., because some other thread invoked `signal()`, and the return value will be non-zero when the thread's waiting was explicitly cancelled by some other thread.

As an example, consider the following trace which which demonstrates the relationship between `tcv_cancel()` and `tcv_signal()`.

Time	Thread 0	Thread 1
0	i = tcv_wait(v,m)	
1	...wait...	
2		tcv_signal(v)
3		tcv_cancel(v,0,-17) → -1
4	i → 0	
5	j = tcv_wait(v,m)	
6	...wait...	
7		tcv_cancel(v,0,-17) → 0
8		tcv_signal(v) // no effect
9	j → -17	

A small example program using a targeted condition variable is displayed on the next page.

The remainder of this page is intentionally blank.

```
#define NTHREADS 10
#define NROUNDS 100

tcv_t   tcv;
mutex_t mutex;
int     tids[NTHREADS];
int     counter = 0;
int     aborted = false;

void* work(void* index_arg);
void* control(void* ignored);

int main(void) {
    thr_init(4096);      // exam: no failure
    tcv_init(&tcv);      // exam: no failure
    mutex_init(&mutex);  // exam: no failure

    tids[0] = thr_create(control, NULL);  // exam: no failure

    for (int t = 1; t < NTHREADS; t++) {
        tids[t] = thr_create(work, (void*)t);  // exam: no failure
    }

    for (int t = 0; t < NTHREADS; t++) {
        thr_join(tids[t], NULL);  // exam: no failure
    }

    mutex_destroy(&mutex);
    tcv_destroy(&tcv);
    thr_exit(0);
}

int cancel(int index) {
    return tcv_cancel(&tcv, tids[index], index);
}
```

The remainder of this page is intentionally blank.

```
void* control(void* ignored) {
    char c;
    int  abort_index = 1;
    while ((c = getchar()) != 'q') {
        if (isdigit(c)) {
            for (int i = 0; i < (c - '0'); i++) {
                tcv_signal(&tcv);
            }
        } else if (c == 't') {
            if (cancel(abort_index) == 0) {
                abort_index++;
            }
        }
    }
    mutex_lock(&mutex);
    aborted = true;
    for (; abort_index < NTHREADS; abort_index++) {
        cancel(abort_index);
    }
    mutex_unlock(&mutex);
    return NULL;
}

void* work(void* ignored) {
    int result = 0;
    for (int r = 0; r < NROUNDS && result == 0; r++) {
        // Do work
        sleep(genrand() % 100);
        mutex_lock(&mutex);
        if (!aborted) {
            result = tcv_wait(&tcv, &mutex);
        }
        mutex_unlock(&mutex);
    }

    return NULL;
}
```

Your task is to implement a targetable condition variable with the following interface. Note that you will not need to implement a `broadcast()` operation.

- `int tcv_init(tcv_t *t)`
  The targetable condition variable shall be initialized. It is illegal for an application to use the targetable condition variable before it has been initialized or to initialize a targetable condition variable when it is already initialized and in use. `tcv_init()` shall return 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, not because you must write code that returns error indications).
- `void tcv_destroy(tcv_t *t)`
  The targetable condition variable shall be destroyed. It is illegal for a program to invoke `tcv_destroy()` if any threads are operating on it.
- `int tcv_wait(tcv_t *t, mutex_t *mp)`
  The targetable condition variable shall wait until signalled (`tcv_signal`) or cancelled (`tcv_cancel`). The mutex `mp` will be released when waiting and reacquired before returning. The mutex will be reacquired even if the wait was cancelled. `tcv_wait()` shall return 0 if successfully signalled (`tcv_signal()`) or a non-zero value if cancelled (`tcv_cancel()`).
- `void tcv_signal(tcv_t *t)`
  The targetable condition variable shall be signalled, waking up a single waiting thread if one exists.
- `int tcv_cancel(tcv_t *t, int tid, int result)`
  If the indicated thread is waiting on this targetable condition variable, it will be awakened and the return value from `tcv_wait()` will be `result`; the result of `tcv_cancel()` will be zero. Otherwise, the result of `tcv_cancel()` will be a negative error code. Threads other than the indicated one should not be awakened.

The remainder of this page is intentionally blank.

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 `tcv_t` here. If you need one (or more) auxiliary structures, you may declare it/them here as well.

```
typedef struct {

} tcv_t;
```

(b) [15 points] Now please implement `tcv_init()`, `tcv_wait()`, `tcv_signal()`, and `tcv_cancel()`.

. . . space for targetable condition variable implementation . . .

. . . space for targetable condition variable implementation . . .

. . . space for targetable condition variable implementation . . .

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

When working on your P2 thread library, your partner has an idea to make `thr_create()` "more efficient" by "preloading" the thread stack using a function called `stack_create()` which is called by `thr_create()`. You are skeptical of this idea and show your partner some stack diagrams to argue that copying stacks is a bad idea. The diagrams are based on invoking the `stack_create()` function without involving the remainder of `thr_create()`, which hasn't been fully written yet anyway. As you draw your diagrams, you will consider various implementations of a function called `copy_stack()`.

For your convenience, both the C program and the corresponding assembly are shown on subsequent pages.

You may assume that the "main" stack is [0xFFFFE000, 0xFFFFFFFF] inclusive and the "new stack" is [0x2000E000, 0x2000FFFF] inclusive.

This problem has five parts. **Please read all five parts before starting on the first one.**

The remainder of this page is intentionally blank.

```
#define STACK_SIZE 4096
#define NEW_STACK_LOW 0x2000E000
#define MAIN_STACK_LOW 0xFFFFE000

void* allocate_stack(void) {
    new_pages((void*)NEW_STACK_LOW, STACK_SIZE);
    return (void*)NEW_STACK_LOW;
}

/* Copies the contents of the current stack into the new stack such that the
 * new_stack is valid even if the current stack goes out of scope.
 */
void copy_stack(void* stack_low);

int stack_create(void* (*func)(void*), void* arg) {
    int  a  = 3;
    int* ap = &a;

    void* new_stack_low = allocate_stack();

    // Part A

    copy_stack(new_stack_low);

    // Part B,C,D

    a = 2;

    return *ap;
}

void* do_work(void* arg) {
    // Don't do too much work
    return NULL;
}

int main(void) {
    // See register dump in disassembly listing for the initial state at this point
    int result = stack_create(do_work, (void*)0xF00D);

    task_vanish(result);
}
```

```
01000000 <allocate_stack>:
    # Implementation Not Shown

01000021 <stack_create>:
 1000021: push   %ebp
 1000022: mov    %esp,%ebp
 1000024: sub    $0x10,%esp
 1000027: movl   $0x3,-0xc(%ebp)
 100002e: lea    -0xc(%ebp),%eax
 1000031: mov    %eax,-0x4(%ebp)
 1000034: call   1000000 <allocate_stack>
 1000039: mov    %eax,-0x8(%ebp)
    # PART A
 100003c: mov    -0x8(%ebp),%eax
 100003f: mov    %eax,(%esp)
 1000042: call   1000087 <copy_stack>
    # PART B, C, D
 1000047: movl   $0x2,-0xc(%ebp)
 100004e: mov    -0x4(%ebp),%eax
 1000051: mov    (%eax),%eax
 1000053: leave
 1000054: ret

 01000055 <do_work>:
    # Implementation Not Shown

 0100005f <main>:
 100005f: push   %ebp
 1000060: mov    %esp,%ebp
 1000062: sub    $0xc,%esp
# register state after instruction 1000062
    # eax = 0x00000000, ebx = 0x00000000, ecx = 0x00000000
    # edx = 0x00000000, edi = 0xfffff014, esi = 0x00000001
    # ebp = 0xffffefe4, esp = 0xffffefd8, eip = 0x01000065
 1000065: movl   $0xf00d,0x4(%esp)
 100006d: movl   $0x1000055,(%esp)
 1000074: call   1000021 <stack_create>
 1000079: mov    %eax,-0x4(%ebp)
 100007c: mov    -0x4(%ebp),%eax
 100007f: mov    %eax,(%esp)
 1000082: call   1001468 <task_vanish>
```

(a) [6 points] Your task is to finish filling in the stack diagram for the "main" stack where the code is labeled Part A.

Any memory location where the value cannot be determined should be marked with a '?'. The "description" column should be filled in with a succinct description of the value stored the memory location. A good description may be a variable name, a register name, function name, etc. If a description cannot be determined for a specific memory location the description should be marked with a '?'.

Address	Value	Description
0xFFFFEFDC		
0xFFFFEFD8		
0xFFFFEFD4		
0xFFFFEFD0		
0xFFFFEFCC		
0xFFFFEFC8		
0xFFFFEFC4		
0xFFFFEFC0		

(b) [2 points] Consider this implementation of `copy_stack()`.

```
void copy_stack(void* new_stack_low) {
    // Just do a memcpy, easy!
    memcpy((char*)new_stack_low, (char*)MAIN_STACK_LOW, STACK_SIZE);
}
```

Finish filling in the stack diagram for the "new stack" that has been copied (when the code reaches the point marked Part B). If any values are wrong, identify them and explain why the value is wrong.

Any memory location where the value cannot be determined should be marked with a '?'. The "description" column should be filled in with a succinct description of the value stored the memory location. A good description may be a variable name, a register name, function name, etc. If a description cannot be determined for a specific memory location the description should be marked with a '?'.

Address	Value	Description
0x2000EFDC		
0x2000EFD8		
0x2000EFD4		
0x2000EFD0		
0x2000EFCC		
0x2000EFC8		
0x2000EFC4		
0x2000EFC0		

(c) [2 points] Now consider this implementation of `copy_stack()`.

```
extern int* get_ebp(void); // returns %ebp of the calling function

void copy_stack(void* new_stack_low) {
    int* main_ebp = get_ebp();

    memcpy((char*)new_stack_low, (char*)MAIN_STACK_LOW, STACK_SIZE);

    // Chase %ebp up the stack
    // Exam: Assume this terminates without accessing invalid memory
    while (*main_ebp > MAIN_STACK_LOW) {
        int* patched_ebp =
            (int*)((char*)main_ebp - MAIN_STACK_LOW + NEW_STACK_LOW);
        *patched_ebp = *main_ebp - MAIN_STACK_LOW + NEW_STACK_LOW;
        main_ebp     = (int*)*main_ebp;
    }
```

Finish filling in the stack diagram for the "new stack" that has been copied (when the code reaches the point marked Point C). If any values are wrong, identify them and explain why they are wrong.

Any memory location where the value cannot be determined should be marked with a '?'. The "description" column should be filled in with a succinct description of the value stored the memory location. A good description may be a variable name, a register name, function name, etc. If a description cannot be determined for a specific memory location the description should be marked with a '?'.

Address	Value	Description
0x2000EFDC		
0x2000EFD8		
0x2000EFD4		
0x2000EFD0		
0x2000EFCC		
0x2000EFC8		
0x2000EFC4		
0x2000EFC0		

(d) [2 points] Finally, consider an "ideal" or "oracle" implementation of `copy_stack()` (for some reason, we will not show the code for the "oracle" implementation). Fill in the stack diagram below for the "new stack" after the perfect `copy_stack()` implementation has completed (when the code reaches the point marked Part D)

Any memory location where the value cannot be determined should be marked with a '?'. The "description" column should be filled in with a succinct description of the value stored the memory location. A good description may be a variable name, a register name, function name, etc. If a description cannot be determined for a specific memory location the description should be marked with a '?'.

Address	Value	Description
0x2000EFDC		
0x2000EFD8		
0x2000EFD4		
0x2000EFD0		
0x2000EFCC		
0x2000EFC8		
0x2000EFC4		
0x2000EFC0		

(e) [3 points] Explain why it is not a good idea in the general case to copy a stack.

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.