Computer Science 15-410: Operating Systems
Mid-Term Exam (B), Spring 2012

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.	**20**		
4.	**20**		
5.	**15**		
	75		

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

I have not received advance information on the content of this 15-410 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. $\boxed{\text{10 points}}$ Short answer.

   (a) $\boxed{\text{5 points}}$ Give two reasons why memory containing user-program "text" (machine instructions) is typically configured to be read-only. We are expecting one or two sentences for each reason.

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

2. 10 points Arbitration.

In class we briefly discussed the question of how reasonable or unreasonable the behavior of a multi-processor memory controller is likely to be when multiple processors simultaneously attempt an atomic memory operation (such as XCHG) on the same memory location. Below is pseudo-code for a possible implementation of arbitration logic inside a memory controller. This "code" will be run once per memory cycle and will receive a bit-mask of the processors which are trying to run an atomic memory operation in that memory cycle. In this simplified model, processors which, during this memory cycle, are either not accessing memory at all or are issuing non-atomic memory accesses are handled by some other mechanism we are not concerned with. Also, this model ignores the specific memory addresses being operated on: if, in a given memory cycle, two processors are trying to make atomic accesses, the logic chooses a (single) winner regardless of whether they were operating on the same address or different addresses.

If a given processor makes an atomic request in one cycle and is not selected as the winner by the arbitration logic, it will generally try again in the next cycle (e.g., LOCK XCHG will generally retry until it succeeds in doing its swap), though this is not strictly guaranteed (some other atomic instruction, which includes "fail" or "give-up" semantics, might be used, or a processor trying to do an XCHG might "give up" for a while if an interrupt causes it to switch to a different instruction stream). Also, if a processor is chosen as the winner during one cycle, it could conceivably try again in the next cycle (imagine a sequence of back-to-back LOCK XCHG instructions).

```
/**
 * NCPU is the number of processors in the system.
 * This number is guaranteed to be fewer than the
 * number of bits in a word (32).
 */
#define NCPU 8

/**
 * @param reqmask    bit-mask: 1<<p indicates processor p is requesting
 * @return           bit-mask: 1<<p indicates the winner (exactly one)
 *                             if return is 0, no CPU wins this time
 */
unsigned int choose (unsigned int reqmask)
{
  int p;

  for (p = 0; p < NCPU; ++p) {
    if (reqmask & (1 << p)) {
      return (1 << p);
    }
  }
  return (0);
}
```

(a) $\boxed{\text{3 points}}$ Briefly describe what's wrong with the logic presented above (in terms of issues relevant to this class).

(b) $\boxed{\text{7 points}}$ Write a small piece of code which does a better job, in terms of addressing the issue you identified in the previous part. Because "variables" in your code will end up as machine registers, your solution should not use a genuinely excessive amount of space (using four 32-bit registers for each processor seems like probably a lot of space). Because transistors are cheap these days, and the number of processors isn't too large, you shouldn't worry "too much" about execution time: $O(n^2)$ is ok if you need it.

3. $\boxed{\text{20 points}}$ Trouble in the Barbershop

Your friend Luigi is thinking of running a barber shop downtown. He figures it will be mostly easy living: he can sit around the shop all day, sleeping most of the time, and cut some hair when people arrive in the waiting room. Depending on which vacant storefront he rents, he will end up with waiting rooms of various sizes. If his waiting room ever fills up, he will need to turn some customers away unsatisfied. In order to estimate how much space he should rent, he wants to run some simulations. The code for his model appears below. Note: For the purposes of this question, you may assume that condition variables are "as FIFO as possible."

```
#define MAX_WAITERS 10
#define CUSTOMERS_PER_DAY 30
static int num_waiters = 0;
static int barber_busy = 0;
static int cut_done = 0, chair_full = 0;
static mutex_t  waiting_room_lock,   chair_lock;
static cond_t   waiting_room_block,  chair_block,  barber_block;

static void cut_hair()             // Luigi does this (reluctantly)
{
    mutex_lock(&chair_lock);
    while(!chair_full) {
        cond_wait(&chair_block, &chair_lock);
    }
    mutex_unlock(&chair_lock);
    sleep(17); // Haircuts take some time
    mutex_lock(&chair_lock);
    cut_done = 1;
    cond_broadcast(&chair_block);  // No more than two
    mutex_unlock(&chair_lock);
}
static void get_hair_cut()         // Customers do this (they hope)
{
    mutex_lock(&chair_lock);
    while (chair_full) {
        cond_wait(&chair_block, &chair_lock);
    }
    chair_full = 1;
    cond_broadcast(&chair_block);  // No more than two
    while (!cut_done) {
        cond_wait(&chair_block, &chair_lock);
    }
    cut_done = 0;
    chair_full = 0;
    cond_broadcast(&chair_block);  // No more than two
    mutex_unlock(&chair_lock);
}
```

```
static void check_waiting_room()
{
    mutex_lock(&waiting_room_lock);
    if (num_waiters == 0) {
        cond_wait(&barber_block, &waiting_room_lock); // Bonus: naptime!!!
    } else  {
        cond_signal(&waiting_room_block); // Waiting customer: rush back to chair
    }
    mutex_unlock(&waiting_room_lock);
}

void *barber(void *ignored)
{
    while(1) {
        check_waiting_room();
        cut_hair();
        barber_busy = 0;
    }
    return NULL; // Not gonna happen!
}
```

```
void *customer(void *ignored)
{
    mutex_lock(&waiting_room_lock);
    if (barber_busy || (num_waiters > 0)) {
        if (num_waiters == MAX_WAITERS) {
            /* There's no room to wait, so I will have to go home without
             * a haircut today.  Luckily my hair won't grow all that much
             * overnight!
             */
            mutex_unlock(&waiting_room_lock);
            return ((void *) -1);
        }
        /* In the waiting room!   No need to check waiting room capacity again,
         * since I know I have a space.  Maintain accurate count of waiters.
         */
        while (barber_busy) {
            num_waiters++;
            cond_wait(&waiting_room_block, &waiting_room_lock);
            num_waiters--;
        }
        /* Ready to depart from waiting room */
    } else {
        /* Luigi is asleep, so we just wake him up and go in */
        cond_signal(&barber_block);
        /* Ready to depart from waiting room */
    }
    barber_busy = 1;
    mutex_unlock(&waiting_room_lock);
    get_hair_cut();
    return NULL;
}
```

```
int main()
{
    int ctids[CUSTOMERS_PER_DAY], btid;
    mutex_init(&waiting_room_lock);  cond_init(&waiting_room_block);
    mutex_init(&chair_lock);         cond_init(&chair_block);
    cond_init(&barber_block);

    thr_init(3*PAGE_SIZE);

    btid = thr_create(barber, NULL);
    assert(btid > -1);

    int c;
    for(c = 0; c < CUSTOMERS_PER_DAY; c++) {
        ctids[c] = thr_create(customer, NULL);
        assert(ctids[c] > -1);
        sleep(25);
    }
    for(c = 0; c < CUSTOMERS_PER_DAY; c++) {
        thr_join(ctids[c], NULL);
    }
    task_vanish(0); // don't wait around for Luigi: he's probably asleep
}
```

This code doesn't work to Luigi's satisfaction, but (since he's a barber, not a computer scientist) he can't clearly explain what's going wrong or figure out why. Describe how this code can get Luigi's virtual shop into a situation which is impossible in the real world or which would threaten Luigi's business with bankruptcy, *then back up your claim with an execution trace in the tabular format used in class.* If you can't describe and show a particular problem, a small amount of partial credit may be awarded for correct descriptions of how the code is incorrectly structured.

Suggestions for working on this problem:

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

Luigi	Cust 0
lock(wroom)	...
wait(bblock)	...
...	lock(wroom)

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!

(a) $\boxed{\text{15 points}}$ Show a trace of the problem if you can. Otherwise you can receive partial credit for saying what's wrong structurally. *(Don't forget about the second part of the problem, two pages ahead.)*

You may use this page as extra space for the first part of the barbershop question if you wish.

(b) $\boxed{\text{5 points}}$ Briefly describe how to fix or restructure the code to solve the problem.

Andrew ID: ____________

4. $\boxed{\text{20 points}}$ Super semaphores

The semaphore is a thread-synchronization primitive based on the metaphor of a pool of identical resources; threads request a resource from the pool by calling `sem_wait()`. In some situations, threads frequently need multiple resources at the same time. In this problem, your job will be to implement a version of semaphores where both `sem_wait()` and `sem_signal()` are extended to include a count indicating the number of resources being acquired or released, respectively. To simplify your job, users of these "super semaphores" are restricted as follows: once a thread acquires resources using `sem_wait()`, it is *guaranteed* not to call `sem_wait()` again until it has invoked `sem_signal()` one or more times in such a way that it has released all resources from this particular semaphore. Also, it is guaranteed that a single thread will never request more resources from a single semaphore than the peak number of resources which the semaphore has ever held. Thus, usage examples "A" and "B" below are invalid. If you wish, you may optionally *state as an assumption* that a "super semaphore" will never have more resources available than when it was initialized (*if you state that assumption*, then usage example "C" below is invalid). For exam purposes, you may assume that "guarantees" are *always* true and your code does not need to verify they are true or react if they aren't true.

Example A (invalid!) Thread 0
`sem_init(s,3)`
`sem_wait(s,2)`
`sem_signal(s,1)`
`sem_wait(s,1) // still holding 1!`

Example B (invalid!) Thread 0
`sem_init(s,3)`
`sem_wait(s,4) // stuck!`

Example C (??) Thread 0
`sem_init(s,3)`
`sem_signal(s,1) //` $4 > 3$

(Continued on next page)

Your mission

In this problem you will implement "super semaphores" using mutexes and condition variables. You may *not* use other atomic or thread-synchronization synchronization operations, such as, but not limited to: semaphores (obviously), reader/writer locks, `deschedule()/make_runnable()`, or any atomic instructions (`XCHG`, `LL/SC`). You may assume that there is a mutex implementation available for use, with the same interface as provided with P2; you may assume that this mutex implementation has bounded waiting (e.g., is FIFO) and does not block threads. You may likewise assume a P2-compliant condition-variable implementation, which you may assume to be strictly-FIFO if you wish. **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). **However, 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. You may use non-synchronization-related thread-library routines in the "`thr_xxx()` family," e.g., `thr_getid()`.

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!

There are multiple "legal" solutions (ones that operate defensibly or "reasonably well" for every valid execution sequence). Some solutions that are not only legal but also "nice" are too long and complicated for us to expect them as exam solutions. Observe that, compared to regular semaphores, these "super semaphores" have "interesting" states, in which multiple resources are available and multiple threads, with potentially different needs, are waiting. If you can spend a little more design time (say, ten minutes) to get a reasonable-length solution which is not only legal but also handles some of the "interesting" states in a "nice" fashion as opposed to a "crude" fashion, you will receive a slightly higher score (we expect valid answers to fall into three different classes, which may be scored differently). But don't spend *too* much time thinking about "nice"... getting something that works is more important, plus there are other questions on the exam!

The remainder of this page is intentionally blank.

Please declare a `struct sem` and implement:

- `int sem_init(sem_t *sem, int initial)`,
- `void sem_wait(sem_t *sem, int request)`,
- `void sem_signal(sem_t *sem, int release)`

You do not need to implement `sem_destroy()`.

If you wish, you may also declare an auxiliary structure, `struct aux`, *but this is strictly optional.*

```
typedef struct sem {









} sem_t;
```

```
typedef struct aux {






} aux_t;
```

...space for super-semaphore implementation...

You may use this page as extra space for your super-semaphore solution if you wish.

5. $\boxed{\text{15 points}}$ `Nuts & Bolts`.

Although the availability of computing power has been increasing at a remarkable pace in recent history, there is still a drive to squeeze as much computation as possible out of the hardware on hand. To that end, compiler writers have come up with all sorts of tricks to speed up code.

One technique which can give a small speedup to compiled code is *omitting the frame pointer*.

Recall from 15-213 and Project 0 that a compiled function generally has a "prologue" (start-up code) and an "epilogue" (clean-up code) looking like this:

```
push %ebp             # prologue
mov %esp, %ebp        # prologue
...
mov %ebp, %esp        # epilogue
pop %ebp              # epilogue
ret
```

Now, consider the following C function:

```
// sum first n values of f(): f(n-1)+f(n-2)...+f(0)
int partial_series(int n) {
    int sum = 0;
    while (n > 0) {
        sum += f(n);
        n--;
    }
    return sum;
}
```

(Continued on next page)

That C code might be compiled as follows:

```
.global partial_series
partial_series:
        # Set up stack frame
        push    %ebp            # <- XXX
        mov     %esp, %ebp      # <- XXX
        sub     $16, %esp

        # Save %ebx and %esi
        mov     %ebx, 8(%esp)
        mov     %esi, 4(%esp)

        # Load n off of stack, and clear sum
        mov     8(%ebp), %ebx
        mov     $0, %esi

        # If n < 0, leave early
        test    %ebx, %ebx
        jle     .done

.loop:
        # Call f(n)
        mov     %ebx, (%esp)
        call    f

        # Add result to sum; decrement n
        add     %eax, %esi
        sub     $1, %ebx

        # Continue if needed
        test    %ebx, %ebx
        jg      .loop

.done:
        # Return sum
        mov %esi, %eax

        # Restore saved registers
        mov     8(%esp), %ebx
        mov     4(%esp), %esi
        mov     %ebp, %esp      # <- XXX
        pop     %ebp            # <- XXX
        ret
```

(a) $\boxed{\text{5 points}}$ How might you make it possible for `partial_series()` to run correctly without saving a frame pointer on the stack? In other words, suppose the lines marked "XXX"' were deleted. Which other lines would need to be modified to fix the program? You may indicate changes by writing directly on the program listing above. *Please make sure your marks are clear and easy to understand!!!.*

Suppose omitting the frame pointer becomes popular, and you wish to adapt your Project 0 implementation to this new reality. You may recall that in P0, we provided you with a `struct functsym_t`, with the following definition:

```
typedef struct {
  void *addr;                 // Address where the function starts
  char name[FUNCTS_MAX_NAME];  // Name of the function
  argsym_t args[ARGS_MAX_NUM]; // List of arguments that this function takes
} functsym_t;
```

Now that there are no frame pointers, the part of `traceback()` that traces from frame to frame will need to work differently. Assume that the "compiler output" above fairly represents how all functions would be compiled in the omitted-frame-pointer environment.

(b) $\boxed{\text{10 points}}$ Describe what information will need to be added to the `functsym_t` structure for each function and also describe how this new information will be used by `traceback()`. If you wish, you can skip the "base case" (how the first frame is located) and describe the "induction case" (assuming a pointer to some part of a frame and a pointer to the relevant `functsym_t` structure, what happens next? (Memory validity checking and argument printing were significant parts of Project 0, but since the removal of frame pointers will not affect that code significantly, there is no need for you to describe any of that in your answer.)

Andrew ID: ______________

You may use this page as extra space for your `traceback()` solution if you wish.

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 ls(int size, char *buf);

/* "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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.