Computer Science 15-410: Operating Systems
Mid-Term Exam (A), Fall 2004

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.	20		
4.	15		
5.	15		
	75		

I have not discussed the contents of the 15-410 midterm with anybody who took part in a conflict exam session.

Signature: ____ Date ___________________________

1. $\boxed{\text{10 points}}$ Give a *brief* definition of each of the following terms as they apply to this course. You may add a sentence providing an example or a clarification if you wish, but there is no need for long answers.

   (a) $\boxed{\text{2 points}}$ BSS

   (b) $\boxed{\text{2 points}}$ Caller-save

   (c) $\boxed{\text{2 points}}$ `inb()`

(d) $\boxed{\text{2 points}}$ trap frame

(e) $\boxed{\text{2 points}}$ `POPA`

2. $\boxed{\text{15 points}}$ Interrupt handling.

Imagine that, after you finish OS (it can be done) and leave CMU, you are hired by a computer hardware start-up. After you've settled in and learned how the source-control system works, a hardware engineer comes into your office to fill you in on the latest processor-architecture news. The compiler team has sold management on the idea of building a new processor with 1024 general-purpose user registers. Their claim is that far too much time is spent shuffling data to and from memory and that if they are only given enough registers they'll be able to do all sorts of fancy inter-procedure optimizations so that any important data value will essentially never need to be pushed out of the lightning-fast world of registers into the cruel, cold world of memory.

While you are initially excited and inspired by the prospect, you begin to worry. You ask if it will be too expensive to save 1024 registers to memory as part of entering each interrupt handler (and then, of course, to restore all 1024 registers as part of returning from the handler).

The hardware engineer dismisses your concern, pointing out that people type slowly, generating only a handful of interrupts per second. Also, the OS's timer interrupt goes off only 100 times per second.

However, that analysis only makes you more concerned. Your first assignment will be writing the device driver for the new machine's 10-gigabit Ethernet interface, which is capable of delivering literally millions of packets per second to the processor. You know that if you save and restore that many registers so often that you will totally flood the memory system.

The hardware engineer suggests that management has already bought into the massive-register-file approach and they're not likely to give up on their most-radical feature because of one device driver to be written by one lowly programmer. You are advised that if you think there's a problem you'd better solve it yourself, and confine your solution to your own code.

(a) $\boxed{\text{10 points}}$ How can you write your interrupt handler to avoid swamping the memory system? Do you need to do something particularly painful to make this approach work?

(b) $\boxed{\text{5 points}}$ Is there a way you can get revenge on the compiler group by requesting a compiler feature (e.g., a command-line flag) which would make it easier for you to write device driver interrupt handlers at the expense of a bit more work for the compiler group?

3. $\boxed{\text{15 points}}$ Process model

When working on your thread library for Project 2, we asked you to support "implicit thread exit." In other words, in the code below, when `foo()` finishes execution, the result of `return (3);` should be the same as the result of `thr_exit((void *)3);`.

```
int
main()
{
    void *status;

    thr_init(16*1024);
    thr_join(thr_create(foo, (void *) 0), NULL, &status);
    thr_exit(status);
}

void *
foo(void *ignore)
{
    return ((void *)3);
}
```

We did not require you to make "implicit exit" work for the initial (root) thread, the one which calls `main()`. Please outline an approach for making that work, i.e., so that the program above would behave identically if `thr_exit(status)` were replaced with `return((int) status)`. There is more than one way to do it!

4. 20 points Deadlock

Your cousin Terry has recently begun work at a small independent music production company (once these were called "record labels," but these days there are no more records). Terry's department is responsible for duplicating CDs in response to orders from wholesalers. Since the company represents many artists with very small followings, the company uses a "just-in-time" production approach: CDs are duplicated using CD-R burners in response to each individual order. Terry is responsible for maintaining the duplication software, which was written by a programmer who has left the company. The core of the software can be abstracted as follows:

```
int
copy_cd(char *album_name, int ncopies)
{
    int int max_burners, src_drive, *dst_drives, d;
    char linebuf[128], sectbuf[2048];

    max_burners = get_system_config(TOTAL_DRIVES, BURNER);
    while (ncopies > max_burners) {
        copy_cd(album_name, max_burners);
        ncopies -= max_burners;
    }

    dst_drives = calloc(ncopies, sizeof (dst_drives[0]));

    src_drive = alloc_drive(READER);
    printf("Please insert album \"%s\" into drive %d and hit RETURN\n",
        album_name, src_drive);
    (void) fgets(linebuf, sizeof (linebuf), stdin);

    for (d = 0; d < ncopies; d++) {
        dst_drives[d] = alloc_drive(BURNER);
        printf("Please insert blank CD-R into drive %d and hit RETURN\n", dst_drives[d]);
        (void) fgets(linebuf, sizeof (linebuf), stdin);
    }

    while (read_sector(src_drive, sectbuf, sizeof (sectbuf)) >= 0)
        for (d = 0; d < ncopies; d++)
            write_sector(dst_drives[d], sectbuf, sizeof (sectbuf);

    close_and_deallocate(src_drive);

    for (d = 0; d < ncopies; d++)
        close_and_deallocate(dst_drives[d]);
    free(dst_drives);

    return (0);
}
```

One day your cousin comes to work and notices that all the CD-copying employees are sitting around, reading newspapers, doing their nails, etc. When asked why they aren't working, they reply that the system is "stuck," neither copying CDs nor asking them to do anything. Terry asks them why they haven't tried rebooting the system, and their response is that they did that once but that it "got stuck" again soon after.

(a) $\boxed{\text{5 points}}$ Draw a picture to illustrate how the system is "stuck."

(b) $\boxed{\text{5 points}}$ Briefly discuss the two aspects of the application requirements which you believe are the most difficult to address when solving the problem.

(c) $\boxed{\text{10 points}}$ How would you suggest Terry solve the problem? What should be added to the system? Your main constraint is that `copy_cd()` must "work the same way" in terms of making `ncopies` copies of a CD before returning. Your answer need not include any code, pseudo or otherwise, but it should be clear to us that you are addressing the problem in a thoughtful way (i.e., `fatal("cannot proceed!!")` isn't any better than Terry's co-workers have now).

While this is not actually true, you may assume that all CD's take the same amount of time to read (and write).

5. $\boxed{\text{15 points}}$ Concurrency

Consider the folowing mutex implementation for uniprocessors (which has been simplified for exam purposes by removing assertions and error-handling):

```
#define LOCKED 1
#define UNLOCKED 0
typedef struct mutex {
   int status;
   int owner;
} mutex_t;

extern int xchg(int *lockptr, int newval); /* atomic; returns old val */

void mutex_init(mutex_t *m) {
   m->status = UNLOCKED;
   m->owner = -1;
}

void mutex_lock(mutex_t *m) {
   while (xchg(&m->status, LOCKED) != UNLOCKED)
     yield(m->owner);
   m->owner = gettid();
}

void mutex_unlock_one(mutex_t *m) {
   m->owner = -1;
   m->status = UNLOCKED; /* x86 does not require XCHG */
}

void mutex_unlock_two(mutex_t *m) {
   m->status = UNLOCKED; /* x86 does not require XCHG */
   m->owner = -1;
}

void mutex_unlock_three(mutex_t *m) {
   m->owner = -1;
   m->status = UNLOCKED; /* x86 does not require XCHG */
   yield(-1);
}

void mutex_unlock_four(mutex_t *m) {
   m->status = UNLOCKED; /* x86 does not require XCHG */
   m->owner = -1;
   yield(-1);
}
```

Which implementation of `mutex_unlock()` is best? Why? Your answer should probably be two or three medium-length paragraphs.