Computer Science 15-410: Operating Systems
Mid-Term Exam (B), Spring 2006

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.	**15**		
2.	**10**		
3.	**10**		
4.	**20**		
5.	**20**		
	75		

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 _____

1. [15 points] Short answer.

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) [5 points] M:N threads

(b) [5 points] Starvation

(c) [5 points] Write an x86 instruction the execution of which accesses the text region, the rodata region, and the bss region, in that order. If you wish, you may provide some "setup code" written in C, to establish context. If you need to use more than one instruction, try to use as few as possible.

2. [10 points] The mysterious `argv[]`

Imagine a program is invoked by typing

```
./a.out a b all the day
```

Draw a picture of the resulting `argv`, *including the strings*. Label each memory location you are filling with its address (you may "draw" memory in bytes or words as you see fit). You may choose any "plausible" addresses you wish, as long as they make sense. Assume a 32-bit machine with any plausible byte order.

3. [10 points] Recall that, in the Project 1 environment (where an interrupt does not cause a privilege change and hence there is not a stack switch), an interrupt causes the processor to push on the stack, in order, `EFLAGS`, `CS`, and `EIP`.

   Why is there an `IRET` instruction? Is it merely an "abbreviation" for a short sequence of other instructions? If so, show them. If not, argue why not.

4. [20 points] `setjmp()/longjmp()`

The standard C library includes a family of functions with names like `setjmp()` and `longjmp()`. Here we will consider the "in the old days" versions of these functions, which ignored all the intricate complex details of signals. These functions provided a primitive exception-throwing facility, so that a deeply nested function could abandon not only its own execution but also the execution of its caller and caller's caller, up to a higher level of the program which had previously indicated it could handle a specific kind of problem. The program below will either list its command line parameters or print "Fatal error 99" if none were provided.

```
jmp_buf bail_out;

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

  if ((error = setjmp(&bail_out)) != 0) {
    printf("Fatal error %d\n", error);
    exit(error);
  }
  return (process(argc, argv));
}

int process(int argc, char *argv[])
{
  int a;

  checkcount(argc);
  for (a = 1; a < argc; ++a)
    printf("Arg %d is %s\n", a, argc[a]);
  return (0);
}

void checkcount(int argcount)
{
  if (argcount < 2)
    longjmp(&bail_out, 99);
}
```

Answer the following questions with respect to the 15-410 environment (x86-32, gcc stack discipline, etc.). Just as a reminder, caller-save registers are `%eax`, `%ecx`, `%edx`, and callee-save registers are `%ebx`, `%esi`, `%edi`. The instruction sequence to call `foo(1,2)` is

```
pushl $2
pushl $1
call _foo
```

(a) [5 points] Write a C struct declaration which clearly indicates what a `jmp_buf` contains.

(b) [5 points] Provide assembly code for `setjmp()`. We do not intend to be picky about exact syntax, but you should be careful to make it clear to your grader that you understand how to get the job done.

(c) [10 points] Provide assembly code for `longjmp()`. We do not intend to be picky about exact syntax, but you should be careful to make it clear to your grader that you understand how to get the job done.

5. [20 points] Assume we have two mutexes and two threads:

```
mutex_t mutex_a, mutex_b;

void process_ab(int *workload)
{
  for(int i=0; i< 100; i++) {
    mutex_lock(&mutex_a);      /*AB1*/
    mutex_lock(&mutex_b);      /*AB2*/
    (*workload)++;             /*AB3*/
    mutex_unlock(&mutex_b);    /*AB4*/
    mutex_unlock(&mutex_a);    /*AB5*/
  }
}

void process_ba(int *workload)
{
   for(int i=0; i< 100; i++) {
    mutex_lock(&mutex_b);       /*BA1*/
    mutex_lock(&mutex_a);       /*BA2*/
    (*workload)++;              /*BA3*/
    mutex_unlock(&mutex_a);     /*BA4*/
    mutex_unlock(&mutex_b);     /*BA5*/
   }
 }

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

   thr_init(64*1024);
   mutex_init(&mutex_a);
   mutex_init(&mutex_b);
   thr_create(process_ab, &workload);
   thr_create(process_ba, &workload);
   thr_join(0, NULL, NULL);
   thr_join(0, NULL, NULL);
   printf("workload %d\n", workload);
   thr_exit(0);
}
```

(a) [5 points] Explain why this process can deadlock, citing the four ingredients needed for deadlock and giving a list of the lines that would need to execute to create the deadlock.

You may use this page as extra space for your deadlock explanation if you wish.

(b) [5 points] Assume that someone modifies `process_ba()` as follows, where `mutex_is_unlocked()` returns non-zero if and only if the mutex is not locked.

```
void process_ba(int *workload)
{

  for(int i=0; i< 100; i++) {
    mutex_lock(&mutex_b);                /*2BA1*/
    if (mutex_is_unlocked(&mutex_a)) {   /*2BA2*/
      mutex_lock(&mutex_a);              /*2BA3*/
      (*workload)++;                     /*2BA4*/
      mutex_unlock(&mutex_a);            /*2BA5*/
    } else {
     /* Whoops! Try again */             /*2BA6*/
    }
  mutex_unlock(&mutex_b);                /*2BA7*/
  }
}
```

Can this modified process deadlock? Explain in terms of the four ingredients. If the process can deadlock, give a list of line executions that would lead to the deadlock.

(c) [5 points] Is it safe to assume that line 2BA4 executes atomically even though it may be composed of multiple instructions? Explain why or why not.

(d) [5 points] If the main program prints out a value of `workload`, is that value deterministic? Either give the value and explain it, or explain why the value is not deterministic. Indicate which version of `process_ba()` you are basing your answer on.