Computer Science 15-410: Operating Systems
Mid-Term Exam (B), Fall 2008

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.	10		
4.	20		
5.	15		
6.	5		
	75		

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

Andrew ID: ______________

1. $\boxed{\text{10 points}}$ Short answer.

   (a) $\boxed{\text{5 points}}$ Identify and explain one reason for constructing a program so that it uses multiple threads instead of just one thread. We are expecting two to four sentences of answer.

   (b) $\boxed{\text{5 points}}$ Identify and explain a second reason for programs to be multi-threaded.

2. $\boxed{\text{15 points}}$ Dining Philosophers

Joshua, Caroline, and Matt go to a Chinese restaurant at a busy time of the day. The waiter apologetically explains that the restaurant can provide only two pairs of chopsticks (for a total of four chopsticks) to be shared among the three people.

Joshua proposes that all four chopsticks be placed in an empty glass at the center of the table and that eating should obey the following protocol.

```
/* Run by each philosopher */

while (!time_to_grade_P2()) {
  discuss_grading_plan();
  acquire_one_chopstick(); /* may block */
  acquire_one_chopstick(); /* may block */
  eat();
  release_one_chopstick(); /* does not block */
  release_one_chopstick(); /* does not block */
}
```

(a) $\boxed{\text{7 points}}$ Explain briefly but convincingly why this dining plan cannot lead to deadlock. A one-sentence answer is probably too short to be convincing, but convincing answers shorter than six sentences exist. Your answer should probably refer to one or more theoretical characteristics of deadlock.

Suppose now that instead of three diners there will be an arbitrary number, D. Furthermore, each diner may require a different number of chopsticks to eat. For example, it is possible that one of the diners is an octopus, who for some reason refuses to begin eating before acquiring eight chopsticks. The second parameter of this scenario is C, the number of chopsticks which would simultaneously satisfy the needs of all diners at the table. For example, Joshua, Caroline, Matt, and one octopus would result in $C = 14$. Each diner's eating protocol will be as displayed below.

```
int s, nsticks = my_chopstick_requirement();

while (!time_to_grade_P2()) {
  discuss_grading_plan();
  for (s = 0; s < nsticks; ++s)
    acquire_one_chopstick(); /* may block */
  eat();
  for (s = 0; s < nsticks; ++s)
    release_one_chopstick(); /* does not block */
}
```

(b) $\boxed{\text{8 points}}$ What is the smallest number of chopsticks (in terms of D and C) needed to insure that deadlock cannot occur? Explain. Your answer must be organized and convincing.

3. $\boxed{\text{10 points}}$ Philosophers Dining

Knowing a theoretical chopstick bound is all well and good, but how will such a system actually work?

You will provide us with code for three functions. Because global variables would restrict us to one table per restaurant, your code will use an alternate formulation involving pointers to table objects.

- `typedef struct table { ... } *table_p;`
- `void table_init(table_p tp, int nguests, int nsticks)`
- `void table_acquire_one_chopstick(table_p tp)`
- `void table_release_one_chopstick(table_p tp)`

We will not worry about `table_destroy()`. Your solution can use mutexes and/or condition variables, but *may not* use semaphores, reader/writer locks, or machine-dependent synchronization instructions. 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. You may not use assembly code, inline or otherwise. For the purposes of the exam you should assume an error-free environment (memory allocation will always succeed; thread-library primitives will not detect internal inconsistencies or otherwise "fail," etc.). You may wish to refer to the "cheat sheets" at the end of the exam.

The remainder of this page is intentionally blank.

(a) $\boxed{\text{3 points}}$ Please declare your `struct table` here. Also write a function `void table_init(table_p tp, int nguests, int nsticks)` to initialize a table for some guests. You may assume that *nsticks* has been chosen appropriately in accordance with the needs of the *nguests* guests.

```
typedef struct table {









} *table_p;

void table_init(table_p tp, int nguests, int nsticks)
{









}
```

(b) $\boxed{\text{7 points}}$ Now please write `table_acquire_one_chopstick()` and `table_release_one_chopstick()`, which should, respectively, acquire and release one chopstick at the given table.

.

You may use this page as extra space for your table solution if you wish.

4. $\boxed{\text{20 points}}$ Critical-section protocol

Consider the following proposed critical-section protocol. The intuition is that a lock consists of an "outside," a "gate," a "waiting room," and at most one holder of the lock. When the gate is open, one or more threads will enter through the gate and then close it. At this point, multiple threads may be in the waiting room (past the gate), but this set will not increase in size. The waiting-room threads increment "next" as necessary to select a thread in the waiting room. That thread holds the lock and enters the critical section, and all other threads will wait.

To unlock, the unlocker picks the next thread to run before leaving. The next thread will be picked from the waiting room if possible, otherwise it will be one of the threads outside the waiting room: "mygate" will be used to override the global understanding that the gate is closed. If no thread at all is waiting, the waiting room will be opened again by setting gate to TRUE.

```
/* Global variables (we are implementing just one lock) */
#define N 10           // static max number of threads (for exam purposes)
int locked = FALSE;
int gate = TRUE;
int want[N] = FALSE;   // threads that want the lock
int mygate[N] = FALSE; // special gate access for at most one picked thread
int active[N] = FALSE; // threads that got past gate and want the lock
int next = 0;
/* Each thread has one thread-local variable, i */

void lock()
{
   // announce interest in lock
   want[i] = TRUE;

   // gate is TRUE when no one is using or getting the lock
   while (!gate && !mygate[i])
     continue;       // wait to enter
   gate = FALSE;     // no one else can enter

   active[i] = TRUE; // register that we're in the active set

   // no one else can enter, so we have an unchanging active set
   // anyone can figure out who goes next:
   // search, starting from last locker, for an active member
   while (locked || next != i) {
     if (!active[next])
       next = (next + 1) % N;
   }
   // now next == i, it's our turn
   locked = TRUE;
}

```

```
void unlock()
{
    // figure out who's next:
    // active[] set first, starting "to my right"
    for (k = 1; k < N; k++) {
        int j = (i + k) % N;
        if (active[j]) { // j goes next
            next = j;
            active[i] = FALSE;
            locked = FALSE;
            return;
        }
    }
    // otherwise admit someone from want[]
    for (k = 1; k < N; k++) {
        int j = (i + k) % N;
        if (want[j]) { // j goes next
            next = j;
            mygate[j] = TRUE;
            active[i] = FALSE;
            mygate[i] = FALSE;
            locked = FALSE;
            return;
        }
    }
    // no one is waiting
    locked = FALSE;
    gate = TRUE;
}
```

There is a problem with this protocol. That is, it does not ensure that all three requirements (mutual exclusion, progress, and bounded waiting) are always met. Identify a requirement which is not met and lay out a scenario which demonstrates your claim. Use the format presented in class and in the homework assignment, as exemplified below.

T0	T1
w[0] = T	
	w[1] = T

Be sure that the execution trace you provide us with is easy to read and conclusively demonstrates the claim you are making. *It is very important that you not depict impossible execution sequences!* If you wish to claim that some pattern repeats, you must clearly indicate that. Finally, it is strongly recommended that you draft your solution using the scrap paper at the end of the exam before writing anything here.

Use this page as space for the critical-section protocol question.

You may use this page as extra space for the critical-section protocol question if you wish.

5. 15 points Nuts & Bolts.

Consider the following (abbreviated and edited) code from 410kern/boot/head.S, some of the assembly-language underpinnings of the 15-410 kernel base code. Note that `_start` is the entry point to which the boot loader transfers execution; `df_handler` is the start of the double-fault handler; `lgdt()` and `lidt()` inform the hardware of the locations of the Global Descriptor Table and Interrupt Descriptor Table respectively; and `void mb_entry(mbinfo_t *info, void *istack)` is the first C code run in the 410 code base (it is passed a pointer to a structure filled in by the boot loader and a pointer to the stack which is in use).

Also, `void blat(uint8_t *bytes)` is a function, not shown, which fills the screen with many copies of an error message. The message is specified not in standard string format but in a format more convenient for `blat()`.

```
.global _start, init_gdt, init_idt, init_tss
.text
init_idt:
    .space 64
    .long 0x00100998
    .long 0x00108f00
    # ...omitted...
init_gdt:
    .long 0x00000000
    .long 0x00000000
    .long 0x2be00068
    .long 0x00c08b10
    # ...omitted...
init_tss:
    .space 8
    .word 0x0018
    .space 94 /* fills with 0's */
df_handler:
    cli
    /* We got here because we were in trouble, so be conservative */
    movl $SEGSEL_KERNEL_DS, %eax                            # AAA
    movw %ax, %ds                                           # AAA
    movw %ax, %es                                           # AAA
    movw %ax, %fs                                           # AAA
    movw %ax, %gs                                           # AAA
    movw %ax, %ss                                           # AAA
    leal istack, %esp                                       # BBB
    subl $2048, %esp      /* attempt at preservation */     # CCC
    /* Now we probably have a runtime environment */
    leal fbytes, %eax
    pushl %eax
    call blat
df_handler_loop:
    jmp df_handler_loop
```

```
_start:
    leal istack, %esp     /* Load initial stack pointer */
    movl $init_gdt, %eax
    subl $init_idt, %eax
    subl $1, %eax
    pushl %eax            /* idt limit == sizeof(idt)-1 */
    pushl $init_idt
    call lidt
    movl $init_tss, %eax
    subl $init_gdt, %eax
    subl $1, %eax
    pushl %eax            /* gdt limit == sizeof(gdt)-1 */
    pushl $init_gdt
    call lgdt
    addl $16, %esp
    pushl %esp            /* Pass in the top of the initial stack */
    pushl %ebx            /* Pass in the multiboot structure */
    call mb_entry         /* Call the C entry point */
    pushl $pbytes
    call blat
stuck:
    jmp stuck

.data

fbytes:
    .asciz "D\ro\ru\rb\rl\re\r \rf\ra\ru\rl\rt\r!\r \r"

pbytes:
    .asciz "K\re\rr\rn\re\rl\r \rc\ra\rn\rn\ro\rt\r \rr\re\rt\ru\rr\rn\r(\r)\r!\r \r"

/* Initial stack */
.space 4096
istack:
# End of file head.S
```

(a) $\boxed{\text{5 points}}$ What is the motivation behind the lines of code marked “AAA”? What might go wrong if those lines of code were missing?

(b) $\boxed{\text{5 points}}$ What is the motivation behind the line of code marked “BBB”? What might go wrong if that line of code was missing?

(c) $\boxed{\text{5 points}}$ What is the motivation behind the line of code marked “CCC”? Who or what is (potentially) enabled to do what by the presence of that line of code?

6. $\boxed{\text{5 points}}$ Design.

A key component of Project 0, the stack crawler, was safely reading values (or not reading values) via pointers which might be invalid. This task can be accomplished by employing several different techniques. Briefly describe two arguments for using the `SIGSEGV` approach. Then pick one other approach and briefly describe two arguments for using it instead of `SIGSEGV`.

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 cas2i_runflag(int tid, int *oldp, int ev1, int nv1, int ev2, int nv2);
int get_ticks();
int sleep(int ticks); /* 100 ticks/sec */

/* 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);
```

Thread-Library Cheat-Sheet

```
int mutex_init( mutex_t *mp );
int mutex_destroy( mutex_t *mp );
int mutex_lock( mutex_t *mp );
int mutex_unlock( mutex_t *mp );

int cond_init( cond_t *cv );
int cond_destroy( cond_t *cv );
int cond_wait( cond_t *cv, mutex_t *mp );
int cond_signal( cond_t *cv );
int 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 );
int sem_wait( sem_t *sem );
int sem_signal( sem_t *sem );
int sem_destroy( sem_t *sem );

int rwlock_init( rwlock_t *rwlock );
int rwlock_lock( rwlock_t *rwlock, int type );
int rwlock_unlock( rwlock_t *rwlock );
int rwlock_destroy( rwlock_t *rwlock );
```

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.