Computer Science 15-410: Operating Systems
Mid-Term Exam (B), Fall 2009

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

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

Andrew ID: _____

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 conflict 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.

We are expecting each part of this question to be answered by two to five sentences. Your goal is to make it clear to your grader that you understand the concept *as it applies to this course* and can apply it when necessary.

(a) [5 points] Explain how a "trap" and a "fault" differ. Provide an example of each one and discuss how the hardware treats them differently.

(b) [5 points] What is "bounded waiting"?

2. [15 points] `main()` wrapper.

The C language specification states that if the `main()` function executes a `return` statement, the result is as if the value given to `return` had been passed as a parameter to `exit()` instead.

As we discussed in class, in many C run-time environments, this is implemented via a helpful "wrapper function" which "surrounds" the execution of `main()`. We also mentioned that, because surprising things would happen if a programmer accidentally wrote a function with the same name as the name of the `main()` wrapper, the `main()` wrapper sometimes has a name which is not a legal C identifier, such as `$main()`.

For that reason, and for other reasons which apply in certain environments, it may be necessary to write the `main()` wrapper in assembly language. Luckily, it's usually not too hard–which is good, since in this problem you will be asked to do that for the 15-410 P2/P3 run-time environment. You may assume that `main()` is declared as `int main(int argc, char *argv[])`—in other words, you don't need to worry about non-410 frills such as the environment vector or the "aux vector." You don't need to worry about multi-threaded programs—`exit()` was defined before threads anyway.

(a) [5 points] Briefly outline the "todo list" for the `main()` wrapper.

(b) [10 points] Now write the assembly code for the `main()` wrapper, which will be named `$main` (you may assume the assembler is willing to export a symbol named `$main`). Please be sure that your code is clear and comprehensible.

3. [15 points] Bridge problem

As you may know, Pittsburgh is known as "the city of bridges," by which we don't mean the "north bridge" and the "south bridge," but instead the Smithfield Street Bridge, the Fort Duquesne Bridge, etc. In this problem we will be studying a lesser-known bridge, the "Dinky Bridge," so called because it has only one lane (just one—*not* one in each direction). In addition to being narrow, the Dinky Bridge has a weight limit which allows only one car to be driving on it at any given time.

As each car approaches the Dinky Bridge, it invokes either `request(NORTH)` or `request(SOUTH)`; these operations may block the car for some time until the bridge is allocated to the car. When the `request()` operation returns, the car is allowed to drive across the bridge, at which point it invokes `complete()` (which does not take a parameter). So the standard control flow is

```
request(NORTH); drive(); complete();
```

or

```
request(SOUTH); drive(); complete();
```

In this problem we will ask you to evaluate two proposed implementations of `request()` and `complete()`. While reading each one, you may assume that all synchronization objects are properly initialized and that mutexes begin in the unlocked state. It is probably in your best interest to study both implementations before beginning to write your solution to any part of this problem.

```
/* First implementation -- minimal! */

mutex_t bm;
cond_t done;
int available = 1;

void request(int direction) { // "direction" is not used
  mutex_lock(&bm);
  while (!available)
    cond_wait(&done, &bm);
  available = 0;
  mutex_unlock(&bm);
}

void complete(void) {
  mutex_lock(&bm);
  available = 1;
  cond_signal(&done);
  mutex_unlock(&bm);
}
```

```
/* Second implementation - high quality! */

mutex_t bm;
cond_t done;
int next_ticket = 1;
int ready_for = 1;
extern void log_debug(char *s, ...);   // thread-safe

void request(int direction) { // "direction" neither used nor needed!

  int me, my_ticket;
  me = thr_getid();

  mutex_lock(&bm);
  my_ticket = next_ticket++;

  if (ready_for == my_ticket) {
    mutex_unlock(&bm);
    log_debug("%d happily crossing\n", me);
    return;
  } else {
    // I/O is not a "short instruction sequence"!
    mutex_unlock(&bm);
    log_debug("%d eagerly waiting\n", me);
    mutex_lock(&bm);

    while (ready_for != my_ticket)
      cond_wait(&done, &bm);
    mutex_unlock(&bm);

    log_debug("%d happily crossing\n", me);
  }
}

void complete(void) {
  mutex_lock(&bm);
  ++ready_for;
  cond_signal(&done);
  mutex_unlock(&bm);
}
```

(a) [5 points] *Briefly and clearly* state the most problematic thread-synchronization problem you believe the [*first*] implementation suffers from. Show a *clear and compelling* execution trace which supports your claim.

(b) [5 points] *Briefly and clearly* state the most problematic thread-synchronization problem you believe the [*second*] implementation suffers from. Show a *clear and compelling* execution trace which supports your claim.

(c) [5 points] Please *clearly* describe a *simple* code change which solves the problem with the second implementation which you identified in part (b). *It should be possible for you to unambiguously describe the change in one sentence*—or maybe two. You may show code if you wish, but you probably shouldn't: we are looking for a *fix* to one of these algorithms, not a new algorithm. Once you have described the change, also briefly describe why it works.

4. [15 points] Distributed computing.

The 15-410 course staff, having been stressed out for some time by inking student source code, has decided to take a weekend retreat to relax. Of course, the diversity of the staff introduces complications: Milo, who works remotely from the Bahamas, must fly in; Joshua, an adept single-engine pilot, wishes to fly himself to the retreat; Jacob insists on hitchhiking; nobody is willing to ride in a car driven by Michael; etc. So the decision is made that everybody will travel independently.

Because he has many independent-study classes this semester, Jacob volunteers to set up a multi-threaded TA-location server, based on a 410 Pebbles kernel extended with Michael's "PebNet" networking stack. As they travel, members of the course staff will be able to monitor each other's progress by sending queries to the location server.

The communication protocol is fairly simple. The server keeps track of how far each TA has traveled and how many times each TA has been queried by others. Each TA is represented by a thread in the server, which will receive queries directed to its personal "service identifier." Each client message consists of two integers, namely an operation code and an optional parameter. Each reply from the server consists of the operation code of the query, plus a single-integer response code. The "PebNet" messaging middleware layer transparently handles any lost, duplicated, or re-ordered network packets, so each client enjoys error-free communication between itself and the server. PebNet associates each query with a handle which identifies the remote entity which issueshd the query; this handle is used to direct responses to the right place. Finally, of course PebNet is completely thread-safe.

The application protocol includes three messages.

Update - The UPDATE message takes one parameter, the distance traveled since the previous update. The server updates the tracked total distance and replies with that value.

Query - The QUERY message takes one parameter, the identity of a TA whose progress is being queried. The server looks up the TA, increments the count of queries of that TA's location, retrieves the distance traveled so far, and replies with that value. If the TA does not exist, or has already arrived at the retreat and is no longer "in the system," the reply value is -1.

Fin - The FIN message includes a "dummy" parameter, which the server ignores. This message indicates that the TA who sent it has completed the journey and that the server should free all resources associated with that TA. The server replies with the number of queries for that TA's progress which were performed.

```
#include <thread.h>
#include <mutex.h>
#include <pebnet.h>   // pebnet.h declares the next 5 lines, inlined here:
  typedef struct { ... } pn_conn_t;
  typedef { ...} pn_remote_t;
  pn_conn_t *pn_create(int service); // create message endpoint
  void pn_recv(pn_conn_t *cp, pn_remote_t *who, int *cmdp, int *valp); // receive msg
  void pn_send(pn_conn_t *cp, pn_remote_t *who, int cmd, int val);     // send msg
  void pn_destroy(pn_conn_t *cp); // destroy message endpoint
```

```
typedef enum { MSG_UPDATE, MSG_QUERY, MSG_FIN } msg_t;
typedef enum { DAVE, GARTH, ELLY, JOSHUA, MILO, MICHAEL, JACOB } ta_id_t;

typedef struct {
    mutex_t lock;
    int id;
    int distance;
    int num_queries;
    pn_conn_t *conn;
} ta_t;

/**** Splay-tree map, plus locking wrappers ****/
ta_t *map_lookup(map_t *map, int id); /* returns NULL if the TA doesn't exist */
void map_insert(map_t *map, int id, ta_t *ta);
void map_remove(map_t *map, int id);
map_t ta_map;
mutex_t map_lock;

void ta_insert(ta_t *ta)
{
    mutex_lock(&map_lock);
    map_insert(&ta_map, ta->id, ta);
    mutex_unlock(&map_lock);
}

void ta_remove(int id)
{
    mutex_lock(&map_lock);
    map_remove(&ta_map, id);
    mutex_unlock(&map_lock);
}

ta_t *ta_lookup(int id)
{
    mutex_lock(&map_lock);
    ta_t *ta = map_lookup(&ta_map, id);
    mutex_unlock(&map_lock);
    return ta;
}

int main(int argc, char *argv[])
{
  thr_init(128*1024);
  ta_t *nta;

  nta = (ta_t *) malloc(sizeof (*nta));
```

```
  ta_init(nta, DAVE);  thr_create(ta_serve, nta);
  nta = (ta_t *) malloc(sizeof (*nta));
  ta_init(nta, GARTH); thr_create(ta_serve, nta);
  // ... same deal for other staff members ...

  thr_exit(0); // does NOT kill off other threads!
}

/**** TA lifecycle functions ****/
void ta_init(ta_t *ta, int id)
{
    mutex_init(&ta->lock);
    ta->id = id;
    ta->distance = ta->num_queries = 0;
    ta->conn = pn_create(id); // use my "ta #" as my PebNet service locator
    ta_insert(ta);
}

void ta_destroy(ta_t *ta)
{
    mutex_destroy(&ta->lock);
    pn_destroy(ta->conn);
}

/**** Client request processing functions ****/
int ta_update(ta_t *ta, int progress)
{
    mutex_lock(&ta->lock);
    ta->distance += progress;
    int distance = ta->distance;
    mutex_unlock(&ta->lock);

    return distance;
}

int ta_query(int id)
{
    ta_t *ta = ta_lookup(id);
    if (!ta) return -1;

    mutex_lock(&ta->lock);
    ta->num_queries++;
    int distance = ta->distance;
    mutex_unlock(&ta->lock);

    return distance;
```

```
}

int ta_querycnt(ta_t *ta)
{
    mutex_lock(&ta->lock);
    int queries = ta->num_queries;
    mutex_unlock(&ta->lock);

    return queries;
}

/**** The main processing loop ****/
void *ta_serve(void *p)
{
    ta_t *me = p;
    int exiting = 0;

    while (!exiting) {
        pn_remote_t whoever;
        int cmd = 0, arg = 0, reply = 0;

        pn_recv(me->conn, &whoever, &cmd, &arg); // block until request
        switch (cmd) {
        case MSG_UPDATE:
          reply = ta_update(me, arg);
          break;
        case MSG_QUERY:
          reply = ta_query(arg);
          break;
        case MSG_FIN:
          reply = ta_querycnt(me);
          exiting = 1;
          break;
        default:
          panic("protocol botch");
          break;
        }
        pn_send(me->conn, &whoever, cmd, reply); // reply (reliably)
    }

    ta_remove(me->id);
    ta_destroy(me);
    free(me);

    return NULL;
}
```

(a) [7 points] When this code is run, sometimes something bad happens due to improper synchronization of the application threads. State in a sentence or two what is wrong, and then show an execution trace (ideally, in the format used in the lecture slides). Please be sure that your trace clearly and convincingly shows the problem. Obvious abbreviations are ok (e.g., `query()` for `ta_query()`), though they must not be ambiguous or hard to understand. *Note that the problem you are looking for is located in code you can see*—you don't need to speculate that some library function works in a bad way (and such speculations are unlikely to receive high scores). If you believe you see multiple thread-synchronization problems, describe the one you think is the most harmful.

(b) [8 points] Present a solution for this problem. Both good solutions and distasteful solutions are possible; a good description of a distasteful solution is better than nothing, as it will receive some credit, but full credit will be reserved for solutions which are robust and scalable (i.e., could handle hundreds of TA's or longer-running TA-object manipulations). Your solution should include one to three sentences of description and a "code plan": a very brief description of a code change, and possibly a list of functions or places in the code where the change needs to be applied. Your goal is to *fix* the application, not to re-write it, so please do not show large amounts of code!

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

5. [20 points] Semaphore madness

Imagine that a member of the course staff likes semaphores *a whole lot*—maybe even *very much*. That person likes semaphores so much that she has implemented them as the core of a thread library and wants to do away with grimy mutexes (mutices?) and condition variables. However, for compatibility reasons it is necessary to support "legacy code." To that end, your assignment will be to implement mutexes and condition variables using only semaphores—no other synchronization primitives, atomic operations, or assembly language is allowed. For exam purposes, you will not be required to implement every mutex and condition-variable operation. You may assume every call to your code is legal and also assume that every library routine you invoke always succeeds.

(a) [5 points] Show how to implement mutexes in terms of semaphores. You will declare a `struct mutex` and show how to implement `mutex_init()`, `mutex_lock()`, `mutex_unlock()`, and `mutex_destroy()` (remember that you do not need to check for or handle errors). We have provided the skeleton of the structure declaration for you.

```
typedef struct mutex {

} mutex_t;
```

Now please implement `mutex_init()`, `mutex_lock()`, `mutex_unlock()`, and `mutex_destroy()`. Your code should be *clear* and *concise* for full credit.

...space for mutex implementation...

(b) [15 points] Now please declare a `struct cond` and implement `cond_init()`, `cond_wait()`, and `cond_signal()` (you do not need to implement either `cond_broadcast()` or `cond_destroy()`). If you wish, you may also declare an auxiliary structure, `struct aux`, *but this is strictly optional.*

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 here. If we cannot understand the solution you provide on this page, your grade will suffer!

```
typedef struct cond {

} cond_t;

typedef struct aux {

} aux_t;
```

...space for cvar implementation...

You may use this page as extra space for your cvar 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 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.

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.