Computer Science 15-410: Operating Systems
Mid-Term Exam (C), Fall 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.	**15**		
4.	**15**		
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] Medium answer.

Give a three-to-five sentence definition of each of the following terms as it applies to this course. Your goal is to make it clear to your grader that you understand and can apply the concept when necessary.

(a) [5 points] Mode switch

(b) [5 points] Context switch

(c) [5 points] Memory barrier

2. [10 points] Out of Business.

Consider the following code:

```
typedef struct biz {
    mutex_t m;
    cond_t new_work;
    int going_out_of_business;
    queue q;
} biz_t;

/** @brief  Return next work item, or NULL if going out of business
  * @param  bp  Pointer to a biz struct
  */
workitem *
find_work(biz_t *bp)
{
  workitem *w = 0;

  mutex_lock(&bp->m);

  while(!w && !bp->going_out_of_business) {
    if (!(w = (workitem *) dequeue(&bp->q)))
      cond_wait(&bp->new_work, &bp->m);
  }
  mutex_unlock(&bp->m);
  return(w);
}
```

For this question you may assume Project 2 thread library semantics and the existence of an appropriate `void enqueue(queue *q, void *item)`.

(a) [4 points] Fill in the contents of `add_work()`:

```
/** @brief  Add an item to the work queue
 *
 * @param  bp  Pointer to a biz struct
 * @param   w  Pointer to a work item
 * @return  0 for success, <0 for error
 */
int
add_work(biz_t *bp, workitem *w)
{




}
```

(b) [6 points] Fill in the contents of `go_out_of_business()`:

```
/** @brief  Shut down this particular business, no matter what
 *
 * @param  bp  Pointer to a biz struct
 */
void
go_out_of_business(biz_t *bp)
{









}
```

3. [15 points] Traveling Traceback.

After completing your thread library, you find yourself enamored of the "Pebbles" kernel environment and are eager to port software to it. You realize that one key obstacle stands between you and running your Project 0 `traceback()` implementation on top of the reference kernel: Pebbles doesn't have `SIGSEGV`, `msync()`, `write()`, or `/proc`.

(a) [7 points] Briefly explain what you need to accomplish and a way of accomplishing it. Make your explanation clear and convincing. As was the case for Project 0, you are not required to `traceback()` a program with multiple running threads. Cleaner or more-comprehensive solutions will generally receive more points.

(b) 8 points Embody your approach in code for this sub-case: you have an `%ebp` value stored in a C variable, `void *ebp`, and you wish to obtain the value of `ebp` for the next-oldest frame. Write `void *fetch_ebp(void *ebp)`. We will not deduct points for missing semi-colons or other compilation errors. We will focus on whether your approach is fully thought out and will work.

4. [15 points] Getting grilled.

Your student organization takes advantage of one of the last days with good weather to have a grill party. For the purposes of this problem, assume that your organization agrees unanimously on whether or not the burgers to be grilled will be vegetarian. However, some members demand cheese on their burgers, while others reject cheese.

Your task is to codify the procedures to be followed by the event organizer, the chef, and attendees who wish to eat burgers. Your roommate, an ECE major, points out that partygoers are not focused solely on food, taking some time out to socialize, and claims this calls for an an asynchronous, "clock-free" (or "data-flow") design. This makes sense to you and you churn out the code below. On the "everybody at the party will be a responsible adult" principle, you extend your Project 2 thread library to include `thread_detach(tid)` which ("instantly") reconfigures a thread so that nobody needs to `thr_join()` it—and, indeed, make it illegal to do so.

```
extern void
  talk_to_roommate(int sec),
  take_a_drink(int sec),
  admire_scenery(int sec);
extern void *chef(void *ignored);
extern void *feeder(void *who);
extern void light_grill(void);
extern int somebody_looks_hungry(void),
  complex_function(int who),
  grill_temperature(void);

#define COOKING_TEMP 180 /* degrees */
#define COOKING_TIME 45 /* seconds */
#define NSLOTS 5
typedef enum { EMPTY, FIRST_SIDE, SECOND_SIDE, READY} slotstate;
struct slot {
  slotstate state;
  int has_cheese;
  int since;
} grill[NSLOTS];

mutex_t grill_m;
cond_t flip;
cond_t ready;
```

```
/* main() runs "grill party organizer" thread */
int
main(void)
{
  int person, tid, s;

  thr_init(128*1024);
  mutex_init(&grill_m);
  cond_init(&flip);
  cond_init(&ready);

  for (s = 0; s < NSLOTS; s++) {
    grill[s].state = EMPTY;
    grill[s].has_cheese = 0;
  }

  light_grill();
  while (grill_temperature() <= COOKING_TEMP)
    sleep(1);
  thr_create(chef, 0xfeedfeed);

  while (grill_temperature() > COOKING_TEMP) {
    talk_to_roommate(11); /* seconds */
    while (person = somebody_looks_hungry()) {
      while ((tid = thr_create(feeder, person)) < 0)
        talk_to_roommate(3); /* seconds */
      thr_detach(tid); /* now no thr_join() is necessary/possible */
    }
  }
  return (0);
}
```

```
void * chef(void *ignored)
{
  int background_task = 0, s;

  while (grill_temperature() > COOKING_TEMP) {
    switch(background_task++) {
      case 0:
        talk_to_roommate(17); /* seconds */
        break;
      case 1:
        take_a_drink(3); /* seconds */
        break;
      case 2:
        admire_scenery(12); /* seconds */
        background_task = 0;
        break;
    }
    /* Oh, right!  The grill! */
    mutex_lock(&grill_m);
    for (s = 0; s < NSLOTS; s++) {
      switch(grill[s].state) {
        case EMPTY:     /* start a new burger cooking */
        grill[s].state = FIRST_SIDE;
        grill[s].has_cheese = 0;
        grill[s].since = time(NULL);
        break;
      case FIRST_SIDE:
        if (time(NULL) - grill[s].since < COOKING_TIME)
          continue;
        if (grill[s].has_cheese)
          panic("Must flip unflippable burger, slot %d\n", s);
        grill[s].state = SECOND_SIDE;
        grill[s].since = time(NULL);
        cond_signal(&flip);
        break;
      case SECOND_SIDE:
        if (time(NULL) - grill[s].since < COOKING_TIME)
          continue;
        grill[s].state = READY;
        grill[s].since = time(NULL);
        cond_signal(&ready);
        break;
      case READY:
        cond_signal(&ready); break;
      default:
        panic("Grill on fire, slot %d state %d\n", s, grill[s].state);
        break;
      }
    }
    mutex_unlock(&grill_m);
  }
  return (0);
}
```

```
/* invariant: each cheese-seeker cheeses one
 * burger and consumes one cheesed burger
 */
void *
feeder(void *who)
{
  int want_cheese = complex_function((int) who);
  int did_cheese = 0;
  int done = 0;
  int s;

  mutex_lock(&grill_m);

  while (!done) {

    while (want_cheese && !did_cheese) {
      /* Cheese a burger myself, don't steal a pre-cheesed one. */
      /* Make sure there's enough time for cheese to melt. */

      cond_wait(&flip, &grill_m);

      for (s = 0; s < NSLOTS; s++) {
        if ((grill[s].state == SECOND_SIDE) && !grill[s].has_cheese &&
          (time(NULL) - grill[s].since < COOKING_TIME/2)) {
          grill[s].has_cheese = 1;
          did_cheese = 1;
          break;
        }
      }
    }

    while (!done) {
      cond_wait(&ready, &grill_m);
      for (s = 0; s < NSLOTS; s++) {
        if ((grill[s].state == READY) &&
          (grill[s].has_cheese == want_cheese)) {
            grill[s].state = EMPTY; /* yum! */
            done = 1;
          break;
        }
      }
    }
  }
  mutex_unlock(&grill_m);
  return (0);
}
```

Unfortunately, at the end of the party, the executive board of your organization meets in secret and votes to strip you of your position and the associated title ("Supreme Grill Coder").

(a) [5 points] Identify and explain why your organization was unhappy with this approach. If you can identify multiple problems, list up to three and indicate their order of seriousness (i.e., "most", "intermediate", "least"). Make sure your explanation of each problem is sufficient to provide a genuine diagnosis; no points will be assigned for unsupported claims.

(b) [10 points] Explain how to fix the most-serious problem you identified. If you wish to make changes to the data structures, your explanation can probably consist of declarations, a paragraph or two of text motivating the data structure changes, and a *small* quantity of pseudo-code–just enough to make it clear how the flow of your solution is different.

If you do not wish to make any data structure changes, your explanation can probably consist of a description of how the code must be adjusted and a *small* quantity of pseudo-code.

It is *not necessary* for you to rewrite all of the code from scratch.

You may use this page as extra space for the grill question if you wish.

5. [20 points] Consider the following critical-section protocol:

```
volatile boolean waiting[2] = { false, false };
volatile int turn = 0;

do {
    waiting[i] = true;
    if (turn == j) {
      while (waiting[j]) {
        /* wait with back-off */
        waiting[i] = false;
        waiting[i] = true;
      }
    } else {
      while (waiting[j]) {
        /* wait eagerly */
        continue;
      }
    }
   ...critical section...
   turn = j;
   waiting[i] = false;
   ...remainder section...
} while (1);
```

(This protocol is presented in "standard form," i.e., when process 0 is running this code, $i == 0$ and $j == 1$; when process 1 is running this code, $i == 1$ and $j == 0$, so i means "me" and j means "the other process")

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, i.e.,

P0	P1
waiting[0] = false;	
	turn = 0;

Be sure that the execution trace you provide us with is easy to read and conclusively demonstrates the claim you are making.

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