Computer Science 15-410: Operating Systems
Mid-Term Exam (B), Fall 2005

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.	**25**		
3.	**20**		
4.	**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. [10 points] Your high school friend Kelly comes to visit you near the beginning of 15-410 and attends a few lectures. Not having found the lecturer's style compelling, Kelly decides to get some revenge when the Project 1 tarball is issued. Here is Kelly's program.

```
/**
 * @brief Try *this* "shortest legal fairy tale"!!!!!!!
 */
int
kernel_main(void)
{
  return(kernel_main());
}
```

What happens when Kelly's program is run? If you feel that multiple scenarios are possible, please pick one and focus on its details. You may wish to break your answer into several parts, such as "Assuming the ... is ...", "At first, ...", "After some time..." and/or "Eventually...".

Your answer doesn't have to *exactly* match the particular quirks of x86 hardware as long as you justify your response in terms of plausible hardware. That is, we're looking for a plausible understanding of the parts of the problem and how they might well fit together.

You may use this page as extra space for the Kelly's-program question if you wish.

2. [25 points] Threads

Mike is working on a complicated multi-threaded simulation game, where each object in the game is simulated using a thread. To make the game easily extensible, Mike wrote an abstract interface for creating object threads, but it never worked. In the process of debugging, Mike stripped the code down to something that looks like the code below (error checking removed).

```
 1:  typedef struct {
 2:    int object_id;     /* unique id of this particular object */
 3:    char *owner_name;  /* name of the owner of this object */
 4:  } obj_desc_t;

 5:  void *object_thread(void *arg) {
 6:    obj_desc_t *desc = (obj_desc_t *)arg;
 7:    printf("I'm thread %d. I simulate object %d, which belongs to %s.\n",
 8:           thr_getid(), desc->object_id, desc->owner_name);
 9:    return NULL;
10:  }

11:  void make_object_thread(int id, char *name) {
12:    obj_desc_t desc = {id, name};
13:    thr_create(object_thread, (void*)&desc);
14:  }

15:  int main() {
16:    char *owners[] = {"Mike", "Rahul", "Dave", "Harry Q. Bovik"};
17:    const int num_owners = sizeof(owners) / sizeof(owners[0]);

18:    thr_init(STACKSIZE);

19:    for (i = 0; i < NUM_OBJECTS; ++i)
20:      make_object_thread(i, owners[i % num_owners]);

21:    for (i = 0; i < NUM_OBJECTS; ++i)
22:      thr_join(0, NULL, NULL);

23:    thr_exit(0);
24:  }
```

When Mike runs his code with `NUM_OBJECTS` = 8, he frequently gets the output he expects, but once in a while gets something "crazy" like the following:

```
I'm thread 11. I simulate object 3, which belongs to Dave.
I'm thread 12. I simulate object 3, which belongs to Harry Q. Bovik.
I'm thread 13. I simulate object 3, which belongs to Harry Q. Bovik.
I'm thread 14. I simulate object 3, which belongs to Harry Q. Bovik.
I'm thread 15. I simulate object 6, which belongs to Dave.
I'm thread 16. I simulate object 7, which belongs to Harry Q. Bovik.
I'm thread 17. I simulate object -2134567256, which belongs to
Thread 17 terminated due to segmentation fault.
I'm thread 18. I simulate object -2134567256, which belongs to
Thread 18 terminated due to segmentation fault.
[and then the program appears to hang--the shell doesn't print a prompt]
```

Mike begins by suspecting Rahul's thread library, but Rahul angrily protests that his library is *fine* except for maybe the readers-writers locks and if Mike doesn't trust him he can try his program on a whole *class* full of thread libraries–30 of them! Rahul turns out to be right: different thread libraries produce more or fewer "crazy" runs, which differ somewhat, but overall the evidence strongly suggests something is wrong with Mike's code.

(a) [10 points] Briefly identify the problem.

(b) [10 points] Explain how that problem resulted in the specified output.

(c) [5 points] Suggest a good way (using brief code outlines or text) to solve the problem. Better solutions will receive more points.

3. [20 points] Synchronization

Calvin, Hobbes, Suzie and Mr. Bun are all fond of cookies. Whenever they can, they run to the cookie jar and try to grab a cookie from the jar. They then go away and eat the cookie, following which they return to try and get another cookie. Calvin's mom, as any other mom, is aware of this and is concerned about the dental welfare of the kids/animals. Hence she ensures that at any time there are no more than two cookies in the jar. In other words, when the jar empties, she, not necessarily immediately, adds two cookies to it. She does this only when the jar is empty–if there are any cookies in the jar, she does not fill it further.

Now, the 2-cookie/4-takers situation has caused quite a bit of contention amongst the four friends. Calvin, being the most brilliant scientific mind of our times, has devised a plan to sort out this mess. He has designed an automated system based on semaphores.

The friends make requests to the "Calvin-o-tron" system whenever they want. The system tries to acquire the semaphore. If it does, it grants the requestor a cookie, else it enqueues the requestor. When Calvin's mom refills the jar, the system calls `sem_signal()`, twice (once per cookie).

Here is the code Calvin wrote.

```
typedef struct sem_queue {
   unsigned int tid;
   struct sem_queue *next;
} sem_queue_t;

typedef struct semaphore {
  mutex_t sem_mutex;
  unsigned int counter;
  sem_queue_t *queue;
 } sem_t;

int sem_wait(sem_t *sem)
{
   sem_queue_t *entry;

   mutex_lock(&sem->sem_mutex);
   while (sem->counter == 0) {
     entry = (sem_queue_t *) malloc(sizeof(sem_queue_t));
     if (!entry) {
       mutex_unlock(&sem->sem_mutex);
       return -1;
     }
     entry->tid = gettid();
     entry->next = sem->queue->next;
     sem->queue->next = entry;
     mutex_atomic_drop_with_thread_sleep(&sem->sem_mutex); /* cool! */
     mutex_lock(&sem->sem_mutex);
   }

   --sem->counter;
   mutex_unlock(&sem->sem_mutex);
   return 0;
}
```

```
int sem_signal(sem_t *sem)
{
  sem_queue_t *entry;

  mutex_lock(&sem->sem_mutex);
  ++sem->counter;
  if (sem->queue->next) {
     entry = sem->queue->next;
     sem->queue->next = entry->next;
     thread_wakeup(entry->tid);
     free(entry);
  }
  mutex_unlock(&sem->sem_mutex);
}
```

After some time, Hobbes complains that the "Calvin-o-tron" discriminates against tigers. Calvin, munching a cookie, explains that the source code is "obviously species-independent" so the only possible problem is that tigers have excessive appetites. Who is right? Explain.

4. [20 points] Write now?

Consider the implementation of the `write()` system call. As you know, the invoking user program specifies not only a file descriptor but also a buffer base and a byte count.

Many operating systems copy data from user memory into a file-system buffer cache in kernel memory before storing the file-system buffers onto disk. This works out well when one program writes a file and another program reads it soon thereafter–for example, a C compiler may generate an assembly-language file when is then read from the cache by the assembler.

Of course, the user-memory buffer denoted by the (base, count) pair may be very large. In general users of a computer want most of the RAM to be devoted to their programs rather than the kernel, so it is not difficult for a single `write()` system call to specify a larger memory range than can fit inside the kernel. One approach to this problem would be to copy and store one buffer at a time.

```
/**@brief store a user's I/O buffer to disk via one or more file system buffers.
 *
 * User program may ask for I/O not aligned on a file-system buffer boundary,
 * which we handle by writing part of a buffer the first time.  We may also
 * end up writing only part of the last buffer.
 *
 * @bug Error checking omitted for exam purposes.
 */
int sys_write(int fdnum, char *base, int len)
{
  struct openfile *of = acquireopenfile(fdnum);
  int olen = len;

  while (len > 0) {
    struct fsbuf *bp;
    int copylen, flags = 0;
    int alignfix = of->cursor % BUFFER_PAYLOAD_BYTES; /* misalignment degree */

    if (len + alignfix > BUFFER_PAYLOAD_BYTES)
      copylen = BUFFER_PAYLOAD_BYTES - alignfix;
    else
      copylen = len;

    if (copylen == BUFFER_PAYLOAD_BYTES)
      flags |= BUFFER_WRITE_ENTIRE; /* big speed hack, no need to fetch */

    bp = bufferacquire(of->device, of->file, of->cursor - alignfix, flags);

    memcpy(bp->contents + alignfix, base, copylen);
    base += copylen;
    len -= copylen;
    of->cursor += copylen; /* advance file's read/write cursor */

    bufferrelease(bp, BUFFER_DIRTY|BUFFER_STORE_NOW);
  }
  releaseopenfile(of);
  return (olen - len); /* == original value of len, unless error */
}
```

Note that `bufferacquire()` may block for a variable amount of time. The call may result in a cache hit, meaning the specified range of the specified file is already cached, so the buffer is locked and returned. Alternatively, if the system is not busy, a "blind write" of an entire aligned file system buffer can be quickly satisfied by pulling a blank buffer off of a free list, locking it, and assigning it to the specified chunk of your file. However, if the system is very busy, `bufferacquire()` will probably need to lock a dirty buffer, flush its contents to disk, potentially fill it with part of our file from disk, and return it to us–this will take *many* milliseconds. Other cases abound.

There are several issues with this approach to implementing `write()`. One is that disk devices are noticeably more efficient when they fetch or store large sequential extents in a single hardware transaction. Another is that storing information to disk takes a very long time–it is possible that by the time our one-at-a-time `sys_write()` finishes storing out one file system buffer that the rest of the user program's I/O buffer region has been evicted from memory onto the paging disk. In that case the bits will need to be read from the paging disk into memory in order to be written out to the file system disk, and your manager will not be happy about the poor performance.

Your officemate Leslie observes that the problem is that file system buffers are too small–they are sized with reference to efficiency of disk space allocation, not disk I/O throughput. Leslie suggests that you rewrite your code to use an intermediate abstraction layer called "superbuffers." Each superbuffer is an integral number of file system buffers (for example, 8), fixed at compile time. If you are willing to rewrite your code by replacing calls to `bufferacquire()` and `bufferrelease()` with `superbufferacquire()` and `superbufferrelease()`, and also replacing your abstraction-violating call to `memcpy()` (tsk, tsk) with a call to `superbufferfillbytes()`, then you'll be set.

Because it is so straightforward, Leslie even provides you with an implementation. For example, the `superbufferacquire()` function looks like this.

```
struct superbuffer *
superbufferacquire(dev_t dev, file_t *file, int cursor, int flags)
{
    struct superbuffer *sbp;
    int b;

    sbp = superbufferacquirestruct(); /* always succeeds, but may delay */
    for (b = 0; b < SUPER_FACTOR; ++b) {
      sbp->bp[b] = bufferacquire(dev, file, cursor, flags);
      cursor += BUFFER_PAYLOAD_BYTES;
    }
    return (sbp);
}
```

The good news is that on an idle system superbuffers dramatically increase `write()` performance for important usage patterns such as file copies. You and Leslie are pretty proud of your accomplishment.

But the *bad* news is that sometimes a busy system using superbuffers will slow down dramatically and then all file system activity will come to a stop–apparently forever.

(a) [5 points] Briefly but convincingly explain the performance problem superbuffers sometimes run into.

(b) [10 points] Having tasted the fruits of superbuffers, your programming team is unwilling to give them up. They ask you to design an approach to solving this problem. A clear explanation needn't provide code or pseudo-code (Leslie is all too willing to do that), though you may do so if you wish. Your explanation of your design must contain sufficient detail about how you will redefine and re-implement which parts of the buffer cache manager that your grader can determine the quality of your approach.

(c) [*5 points*] *Briefly* summarize ways that your solution is a good match for this problem.