

CS-537: Midterm Exam (Fall 2013)
Professor McFlub

Please Read All Questions Carefully!

There are fourteen (14) total numbered pages.

Please put your NAME (mandatory) on THIS page, and this page only.

Name: _____

Professor McFlub always makes mistakes, especially when teaching the undergraduate operating systems course. In this exam, you get to correct all of McFlub's mistakes, so that the world won't be misinformed about how operating systems function. Unfortunately, sometimes Professor McFlub is actually correct; in those cases, you should just point that out. Also unfortunately, sometimes McFlub is wrong in more than one way; you have to identify all of them! Finally, sometimes it is hard to tell; if that is the case, say so! You can't always know if McFlub is wrong.

Please save the world from McFlub - if you can!

Tips for good answers: just make a short, concise bulleted list of what is wrong, and please make sure to **include the correction**. For example:

- Uses term Foo incorrectly; should be calling it Bar instead.
- Computes average Foo incorrectly; should be 10 not 5.
- The missing code is $x = x + 1$ just after line 10 (with arrow to point to location where code goes).

If McFlub is completely correct, just say so.

If, in the question, McFlub asks you for specific help, help out! Be the good student that you know you are deep down.

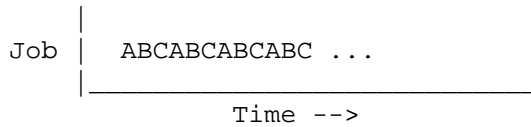
Grading Page

	Points	Total Possible
Q1		10
Q2		10
Q3		10
Q4		10
Q5		10
Q6		10
Q7		10
Q8		10
Q9		10
Q10		10
Q11		10
Q12		10
Q13		10
Q14		10
Q15		10
Q16		10
Total		160

1. Scheduling

“The picture here shows the scheduling of two processes: A, B, and C over time. Let’s look at it.”

(McFlub now draws this on the board)



“As you can clearly see, this is the behavior of a lottery scheduler, rotating between A, B, and C randomly. And you can trust the results: I traced this behavior on a lottery scheduler I built late last night.”

2. Shortest Time To Completion

“Shortest Time to Completion First is a great policy, if you care about response time. But today we’re going to worry about turnaround time. Let’s compute the turnaround times for STCF for these three jobs, which arrive at about the same time: Job A (which needs to run for 10 time units), Job B (needs 20), and Job C (needs 30). Let’s assume this happens when the jobs are run:

ABBCCC

The Turnaround for A is thus 0, 10 for B, and 20 for C. Thus, average turnaround for STCF is 10, which is great. See, I told you.”

3. MLFQ

“The Multi-level Feedback Queue has an intricate set of rules, used to build up a scheduler that achieves many different goals. It starts with a set of queues, ordered from highest to lowest priority. And then it adds rules for moving jobs among the queues. Let’s look at the rules, and see if they make sense:

- If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn’t).
- If $\text{Priority}(A) == \text{Priority}(B)$, A and B run in round-robin style.
- When a job enters the system, it is assigned a random priority and put in the queue corresponding to that priority.
- Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

That’s it! Amazingly, these rules are all you need. Well, that and a drink of coffee, if you’re tired like I am most of the time.”

4. Traps and Interrupts

“Traps, like system calls, and interrupts, such as from a programmable timer or disk, are handled very similarly in OSes. For example, when each occurs, the hardware saves some state (such as the program counter) and jumps into the OS. At that point, the OS handles the trap or interrupt. When finished, the OS returns, through a normal return instruction, and then retries the trapping or interrupted instruction.”

5. Base and Bounds

“Base/bounds-based virtual memory is really easy. Imagine you have a base register and a bounds register in each CPU. The base points to the physical memory location where an address space is relocated; the bounds tells us how big such an address space can be. Let’s do an example to understand this better.

Assume we have the following base/bounds pair:

```
Base    : 0x1000
Bounds  : 0x10
```

Now assume we have the following virtual memory references by a process:

```
0x0
0x4
0x8
0xc
```

The corresponding physical addresses that will be referenced are:

```
0x1000
0x1004
0x1008
FAULT (because this one is out of bounds)
```

Make sense?”

6. OS Involvement With Paging

“In a hardware-managed paging system, the OS has to get involved sometimes. For example, when a process starts running, the OS has to set the page-table base register (PTBR) to point to the page table of the process. Actually, that’s about it for OS involvement with paging, so now you know what you have to know. You’re welcome!”

7. Simple Paging

“With simple linear (array-based) page tables, assume we have the following page table, starting with the first entry and going to the last:

```
0x8000002a
0x00000000
0x00000000
0x00000000
0x8000003e
0x00000000
0x80000017
0x00000000
0x00000000
0x8000000e
0x00000000
0x00000000
0x8000001f
0x00000000
0x8000001a
0x80000001
```

Assuming this is the entire page table, and that each page is 1K, we can see that we have a virtual address space of size 32KB.

Now let’s look at how to do a translation. The format of each page table entry (PTE) above is as follows: A single **valid bit** followed by the **PFN (page frame number)**; you might recall this from the homework (run it with SEED=1000 and PAGESIZE=1k).

Let’s translate the following virtual address: 0x33ef. As you can quickly see, this is an INVALID ACCESS. Stupid program!”

(Is McFlub right? If not, please show him how the address really translates)

8. Segmentation

“With segmentation as our virtual memory system (not paging), assume we have a system with a 1KB virtual address space. Assume the segmented address space has two segments: a code/heap combined segment that grows in the positive direction, and a backwards-growing stack. The top bit of the virtual address is used to differentiate between these segments.

Here is some segment register information:

```
Segment 0 base (grows positive) : 0x00000000 (decimal 0)
Segment 0 limit                    : 1

Segment 1 base (grows negative) : 0x00000064 (decimal 100)
Segment 1 limit                    : 1
```

Now let's do some address translations!

Because the base of segment 0 is 0, translations are super easy. Here are three easy virtual address translations in segment 0:

```
VA 1: 0x00000065 (decimal: 101) --> 0x65 (101)
VA 2: 0x00000169 (decimal: 361) --> 0x169 (361)
VA 3: 0x000002ad (decimal: 685) --> 0x2ad (685)
```

See, easy!”

9. The Advantages of Paging

“Paging has a number of advantages over segmentation. They are... well... paging sounds better, looks better, and in general feels better. That's about it. Oh yeah, paging also makes address translation much faster than segmentation. And, it removes fragmentation of some kind. Yup, that's it.”

10. TLBs

“TLBs make paging systems run faster. Let’s imagine the following code snippet without TLBs:

```
int a[4096];
int b[4096];

int c = 0;

for (i = 0; i < 4096; i++) {
    c = a[i] + b[i];
}
```

Without TLBs, on a system with 4-byte integers and a 4KB page size, and assuming a linear page table, this code would access memory an additional 8192 times (due to page table accesses) – this is a worst-case analysis.

With TLBs, the page tables are only accessed 8 times in the worst-case. What a reduction! Oh, hold on: I think I might have forgotten to think about instruction references...”

11. Multi-level Page Tables.

“I want to make multi-level page tables clearer for you. Imagine you have a linear page table for a process with a 64-byte address space (yes, tiny) and 4-byte pages. Assume each PTE is 1 byte in size. A multi-level page table lets us chop the page table into page-sized chunks, and only allocate those chunks if there is a valid mapping in the page of PTEs. The page directory points to those valid chunks, as follows.

Page directory:

```
PageDirEntry00 (valid; points to A below)
PageDirEntry01 (not valid)
PageDirEntry02 (not valid)
PageDirEntry03 (not valid)
PageDirEntry04 (valid; points to B below)
PageDirEntry05 (not valid)
PageDirEntry06 (not valid)
PageDirEntry07 (valid; points to C below)
```

Page table pages:

```
PTE00 (valid)      <---- A
PTE01 (not)

PTE02 (not valid)
PTE03 (not valid)

PTE04 (not valid)
PTE05 (not valid)

PTE06 (not valid)
PTE07 (not valid)

PTE08 (not valid)
PTE09 (not valid)

PTE10 (valid)     <---- B
PTE11 (not valid)

PTE12 (not valid)
PTE13 (not valid)

PTE14 (valid)     <---- C
PTE15 (valid)
```

Make sense?”

12. Spin Locks

“Assume we have a new instruction called the **Load-and-Store-Zero**, and it does the following atomically (here is C pseudo-code):

```
int LoadAndStoreZero(int *addr) {
    int old = *addr;
    *addr   = 0;
    return old;
}
```

Let’s use it to build a spin lock. Oh, I forgot my notes. Can someone write down how to build a spin lock out of this instruction? Yeah, write down the code for both lock and unlock, as well as some kind of initialization routine. No, I don’t remember how it should work; this is why I asked you to do it!”

(please help McFlub by using the LoadAndStoreZero instruction to build a working spin lock)

13. Atomic Operations

“Fetch-and-add is a primitive that is useful in building many concurrent structures. Some architectures support it as a base instruction. On others, you have to build it. Assume here you only have the atomic exchange instruction, as we talked about before. This is how you build Fetch-and-add, which atomically increments the value at the given address by a specified amount:

```
int FetchAndAdd(int *addr, int amt) {
    int should;
    do {
        should = *addr + amt;
    } while (atomic_exchange(addr, should) != should);
}
```

Thus we have a fetch and add. Tada!”

14. Producers and Consumers

“Here is some code for the producer/consumer problem we are trying to solve. We’ll use a new primitive: `Pthread_cond_broadcast()`. Unlike traditional signaling, this wakes up *all* threads waiting on a condition. Here is some code using such a broadcast:

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == MAX)
            Pthread_cond_wait(&cv, &mutex);
        put(i);
        Pthread_cond_broadcast(&cv); // here!
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&cv, &mutex);
        int tmp = get();
        Pthread_cond_broadcast(&cv); // here!
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Now I think that this code doesn’t work. Or does it? Oh, I don’t know! Anyone have any ideas?”

15. Semaphores

“Semaphores are useful for lots of things. I found this code that uses semaphores. Pretty sure this is useful for something, though I don’t remember what.

```
typedef struct _foo_t {
    sem_t lock1;
    sem_t lock2;
    int   count;
} foo_t;

void foo_init(foo_t *f) {
    f->count = 0;
    sem_init(&f->lock1, 1); // init value of lock to 1
    sem_init(&f->lock2, 1); // same thing here for lock2
}

void do1(foo_t *f) {
    sem_wait(&f->lock1);
    f->count += 1;
    if (f->count == 1)
        sem_wait(&f->lock2);
    sem_post(&f->lock1);
}

void undo1(foo_t *f) {
    sem_wait(&f->lock1);
    f->count -= 1;
    if (f->count == 0)
        sem_post(&f->lock2);
    sem_post(&f->lock1);
}

void do2(foo_t *f) {
    sem_wait(&f->lock2);
}

void undo2(foo_t *f) {
    sem_post(&f->lock2);
}
```

Any ideas what’s going on here?”

16. **Deadlock?**

“I wrote some code and it keeps deadlocking. How frustrating! Here is how the code works. Thread 1 grabs Locks A and B (in some order); Thread 2 grabs Locks B and C (in some order); Thread 3 grabs Locks C and D (in some order); Thread 4 grabs Locks D and A (in some order). Why in the world does this code deadlock? Somebody help me; I just find concurrency hard.”