CSC-501: Operating Systems Principles (Fall 2015)

I'm in Monterey; You're Taking an Exam

Please Read All Questions Carefully!
There are fourteen (14) total numbered pages.

Please put your Name, Unity ID, and student ID on THIS page, and JUST YOUR student ID (but NOT YOUR NAME or UNITY ID) on every other page.
Why are you doing this? So I can grade the exam anonymously. So, particularly important if you think I have something against you! But of course, I don't. Probably.

The exam is closed everything (books, notes, discussion, cell phone, and computer), but you can have one double-sided letter-size cheat-sheet. You have 75 minutes.

Your work must be individual. Cheating will be punished instantly. Please focus on your own exam.

Name: _____Solution__________
Unity ID: ________________________
Student ID: ________________________

Name: ___________________ Unity ID: ___________________ Student ID: ___________________

Grading Page

	Points	Total Possible
Q1		10
Q2		10
Q3		15
Q4		15
Q5		15
Q6		15
Q7		10 (bonus 10)
Total		100 (bonus 10)

I hope I can still answer questions you may have during the exam, but I am in Monterey.

Check the previous grading page to see how many points each program worth.

Anyway, please **read each question carefully** and **arrange your time wisely!**

Good luck!

Name: __________________ Unity ID: __________________ Student ID: __________________

1. **Basics of Scheduling:** Scheduling policies can be easily depicted with some graphs. For example, let's say we run job A for time units, and then run B for 10 time units. Our graph of this policy might look like this:

```
       |
CPU    |AAAAAAAAAABBBBBBBBBB
       |____________________
  0          10        20
```

In the following, let's draw a few of these pictures.

(a) Draw a picture of Shortest Job First (SJF) scheduling with three jobs, A, B, and C, with run times of 5, 10, and 15 time units, respectively.

Make sure to LABEL the x-axis appropriately.

```
       |
CPU    | AAAAA BBBBB BBBBB CCCCC CCCCC CCCCC
       |____________________________________________________________
```

(b) What is the average TURNAROUND TIME for SJF for jobs A, B, and C?

50/3

(c) Draw a picture of ROUND-ROBIN SCHEDULING for jobs A, B, and C, which each run for 6 time units, assuming a 2-time-unit time slice; also assume that the scheduler (S) takes 1 time unit to make a scheduling decision.

Make sure to LABEL the x-axis appropriately.

```
       |
CPU    | SAA SBB SCC SAA SBB SCC SAA SBB SCC
       |____________________________________________________________
```

(d) What is the average RESPONSE TIME for round robin for jobs A, B, and C?

4

(e) What is the average TURNAROUND TIME for round robin for jobs A, B, and C?

24

2. **Subtle Differences:** In this question, we'll examine some subtle differences within an OS; your job is to determine what the effects of these small changes are on the behavior of the operating system

(a) With the round robin (RR) scheduling policy, a question arises when a new job arrives in the system: should we put the job at the front of the RR queue, or the back? Does this subtle difference make a difference, or does RR behave pretty much the same way either way? (Explain)

Discuss possible effects on response time
Mentioning possible starvation effect on front-of-queue a bonus

(b) The multi-level feedback queue policy periodically moves all jobs back to the top-most queue. On a particular system, this is usually done every 10 seconds; the subtle difference we examine is that this value has been shortened to 1 second. How does this subtle difference affect the MLFQ scheduler? In general, what is the effect of shortening this value?

Lesson starvation
More rapidly re-evaluate behavior of job
Done too often, becomes more and more like Round Robin

(c) The timer interrupt is a key mechanism used by the OS. Usually, it waits some amount of time (say 10 milliseconds) and then interrupts the CPU. In this subtle difference, the interrupt is not based on time but rather based on the number of TLB misses the CPU encounters; once a certain number of TLB misses take place, the CPU is interrupted and the OS runs. How does this subtle difference affect the timer interrupt and its usefulness?

Timer interrupt is there to allow OS to retain control of the CPU. If based on TLB misses, no longer possible, as code could run in tight loop and take over system

3. **Paging and Page Tables.** Assume the following: a 32-bit virtual address space, with a 1KB page size.

   (a) How many bits are in the *offset* portion of the virtual address?
   10 bits (for a 1KB page, you need 10 bits to address each byte).

   (b) How many bits are in the *VPN* portion of the virtual address?
   That leaves 22 bits for the VPN (32 bits total - 10 bits of offset).

   Now, let's focus on the page table. Assume each page table entry is 4 bytes in size. Assuming a linear page table:

   (c) How many entries are in the table?
   2^{22} entries (one for each virtual page).

   (d) What is the total size of the table?
   $2^{22}*4$

   (e) In a live system, how much memory would be occupied by page tables? (What factors affect this?)
   16MB times the number of processes

   Linear page tables are too big. Hence, people came upon the idea of a *multi-level page table*, which uses a *page directory* to point to page-sized chunks of the page table. Assume we wish to build such a table, with two levels (as discussed in class). To access such a table, the VPN is split into two components: the $VPN_{PageDir}$ which indexes into the page directory, and the $VPN_{ChunkIndex}$ which indexes into the page of the page table where the PTEs are located.

   (f) How many PTEs fit onto a single page in this system?
   We know a PTE is 4 bytes in size. Hence, 256 entries fit into one 1KB page.

   (g) How many bits are thus needed in the $VPN_{ChunkIndex}$?
   We need 8 bits to tell us which entry we are referring to (2^8= 256).

   (h) How many bits are needed in the $VPN_{PageDir}$?
   Given that the VPN is 22 bits, and subtracting 8 for the ChunkIndex, that leaves us 14 bits for the PageDir.

   (i) How much memory is needed for the page directory?
   Depends on how big each page directory entry (PDE) is. Assuming 4 bytes, we get 2^{14} entries times 4 bytes/entry or 64KB.

   Finally, given the following memory allocations, write down both (a) how much memory our multi-level page table consumes and (b) how much memory a linear page table consumes:.

   (j) Code is located at address 0 and there are 100 4-byte instructions. The heap starts at page 1 and uses 3 total pages. The stack starts at the other end of the address space, grows backward, and uses 3 total pages.
   - Multi-level page table size?
     2 pages (1KB each) plus the page directory (64KB), or 66KB.
   - Linear page table size?
     16 MB.

   (k) Code is located at address 0 and there are 100 4-byte instructions. The heap starts at page 1 and uses 1000 total pages. The stack starts at the other end, grows backwards, and uses 1000 total pages.
   - Multi-level page table size?
     8 pages (1KB each) plus the page directory (64KB) gets us to 72KB.
   - Linear page table size?
     16 MB.

   (l) The entire address space (every page) is used by the process.
   - Multi-level page table size?
     If all pages are used, you get the full linear page table (16MB) plus the page directory (64KB).
   - Linear page table size?
     16 MB.

Name: ___________________ Unity ID: ___________________ Student ID: ___________________

4. **Tracing Virtual Memory.** This question asks you to consider everything that happens in a system on a memory reference. Assume the following: 32-bit virtual addresses, 4KB page size, a 32-entry TLB, linear page tables (if it matters), and LRU replacement policies whenever such a policy might be needed by either hardware or software. Assume further that there are only 1024 pages of physical memory available. In this question, you will be running some test code and saying what happens when that code is run.

Before the test code is run, though, the following initialization code is run once (before testing begins). This code simply allocates (`NUM PAGES*PAGE SIZE`) number of bytes and then sets the first integer on each page to 0, where `PAGE SIZE` is 4KB (as above) and `NUM PAGES` is a constant (defined below). Assume `malloc()` returns page-aligned data in this example.

```
// allocate NUM_PAGES*PAGE_SIZE bytes
void *orig = malloc(NUM_PAGES * PAGE_SIZE);

void *ptr = (int *) orig;
for (i = 0; i < NUM_PAGES; i++) {
  *ptr = 0; // init first value on each page
  ptr += PAGE_SIZE;
}
```

The code we are now interested in running, which we will call *the test code*, is the following:

```
ptr =(int *) orig;
for (i = 0; i < NUM_PAGES; i++) {
  int x = *ptr; // load value pointed to by ptr
  ptr += PAGE_SIZE;
}
```

For these questions, assume we are only interested in memory references to the malloc'd region through `ptr` (that is, ignore stores to `x` and instruction fetches). *How many TLB hits, TLB misses, and page faults occur during the test code when ...*

	TLB hits	TLB misses	Page Faults
(a) `NUM PAGES` is 16?	**16**	**0**	**0**
(b) `NUM PAGES` is 32?	**32**	**0**	**0**
(c) `NUM PAGES` is 2048?	**0**	**2048**	**2048**

Assume a memory reference takes roughly time M and that a disk access takes time D. *How long does the test code take to run (approximately), in terms of* M *and* D*, when...*

	TLB hits	TLB misses	Page Faults
(d) `NUM PAGES` is 16?	**16M**		
(e) `NUM PAGES` is 32?	**32M**		
(f) `NUM PAGES` is 2048?		**2048*2M**	**2048D**

Now assume we change the various replacement policies in the system to **MRU**. Given this change, *how long does the test code take to run (approximately), in terms of* M *and* D*, when...*

	TLB hits	TLB misses	Page Faults
(g) `NUM PAGES` is 16?	**16M**		
(h) `NUM PAGES` is 32?	**32M**		
(i) `NUM PAGES` is 2048?	**32M**	**2016*2M**	**1024D**

Finally, assume you are to run this code on a new machine that you know very little about. In fact, you wish to use the test code to *learn* how big the TLB is and how much memory is on the given system.

(j) How could you use the test code above to learn these facts about the physical hardware?
You could basically time how long an iteration of the loop lasts. If it lasts something like NUM PAGES times memory access time, those are TLB hits; if it lasts twice that, TLB misses, and thus you can know how big the TLB is. One more big jump occurs when NUM PAGES is finally bigger than memory and causes lots of (very slow) disk accesses.

Name: __________________ Unity ID: __________________ Student ID: __________________

5. **Code, Segments. Code Segments?**

We have the following segment table:

Segment	Base	Bounds	Protection
0	0x1000	0x100	Read
1	0x2000	0x200	Read/Write
2	0x5000	0x500	Read/Write

The segment number is taken from the top two bits of each 32-bit virtual address, whereas the rest of each address is the offset into the segment. We then observe this series of accesses and resulting translations, each of which seems to be wrong in some way. Your job: in what way is each translation wrong? In other words, what should each access have done in comparison to what it did do? (Note: each may be wrong in a different way)

(f) Load 0x00000010 → Segmentation violation

should have worked (in segment 0, translates to 0x00001010).

(g) Load 0x40000300 → Loaded word from physical address 0x00002300

should be out of bounds (segment 1 is only 0x200 in size, 0x300 is out of bounds)

(h) Load 0x80000300 → Loaded word from physical address 0x00002300

is in segment 2 and thus base address should be 0x5000

(i) Store 0x00000050 → Stored word to physical address 0x00001050

Protection violation: read only segment

(j) Load 0xC0000010 → Loaded word from physical address 0x00000010

Segmentation violation: tries to access segment 3 which is not defined

Now, assume we have a system with the following setup. There are two segments supported by the hardware. Address spaces are small (1KB), and the amount of physical memory on the system is 16KB. Assume that the segment-0 base register has the value 1KB, and its bounds (size) is set to 300 bytes; this segment grows upward. Assume the segment 1-base register has the value 5KB in it, and its bound is also 300; this segment grows downward (the negative direction).

Assume we have the following program:

```
void *ptr = 20;
while (ptr <= 1024) {
  int x = *((int *)ptr); // LINE 1: read what is at address 'ptr'
  ptr = ptr + 20;        // LINE 2: increment 'ptr' to a new address
}
```

(k) What virtual addresses are generated by this program at `LINE 1` by dereferencing `ptr`, assuming the program runs to completion? (Please just list the addresses as generated by the loads from memory via the dereferencing of `ptr`; don't worry about instruction fetches or the store into `x`, for example)

Starts at virtual address 20; increments by 20 until 1020 (assuming no crash, etc.)

(l) What physical addresses will be generated by dereferencing `ptr` before the program crashes?

Access to virtual addresses 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240, 260, 280 all are fine (14 iterations). Access to 300 exceeds the bounds of segment 0 and thus will cause an exception. The above virtual addresses must be added to the seg 0 base register (value 1024) to get physical addresses. Thus, 1044, ..., 1324 will be generated before crashing.

(m) For those virtual addresses in (k), what virtual addresses are legal but have not been dereferenced as the program crashes? List these virtual addresses and their corresponding physical addresses.

If the program skipped the loop iterations that would cause a crash, it would skip the middle part of the address space which is not mapped and has no legal addresses. The last 300 bytes of the AS are also legal (below 1024 starting at 724), and hence those could have been accessed legally. For example, if 1020 was accessed, it would be legal and translate to 5KB minus 4 or 5116 (remember the stack grows the other way). The first legal address in this range would be 740 which translates to 4836. Thus, every virtual address between 4836 and 5116 (inclusive) could have been legally accessed, if the loop skipped over 300 to 720 (which would all fault).

6. **Concurrency:** You are given multi-threaded code and the initial state (the "inputs") of the program. Your task is to figure out the possible outputs of the code.
Here is the initial state of some variables in the first program we examine:

```
int g = 10;
int *f = NULL;
```

Here are the code snippets for each of two threads:

```
Thread 1                                Thread 2
-----------                             ------------
lock(m);                                lock(m);
f = &g;                                 f = NULL;
... // do some other stuff              unlock(m);
printf("%d\n", *f);
unlock(m);
```

(a) What are all the possible outputs of this program, given an arbitrary interleaving of Threads 1 and 2?

10

The code gets rewritten as follows, to make the lock more "fine grained" by moving the "other stuff" out of the critical section:

```
Thread 1                                Thread 2
-----------                             ------------
lock(m);                                lock(m);
f = &g;                                 f = NULL;
unlock(m);                              unlock(m);
... // do some other stuff
Lock(m);
printf("%d\n", *f);
unlock(m);
```

(b) What are all the possible outputs of the program now?

10 or seg-fault

(c) You've been taught that locks around critical sections prevent "bad things" from happening. Is that true in part (b)? If so, why? If not, why not?

No. The two critical sections should really be one

Let's examine another program, again with two threads:

```
Thread 1                              Thread 2
-----------                           ------------
pending = 1;                          pending = 0;
while (pending) {
  printf("hello\n");
}
```

(d) What are the possible outputs of this program, given an arbitrary interleaving of Threads 1 and 2?

0 to arbitrary number lines of hello

(e) How could we re-write the code such that Thread 2 would only run after "hello" has been printed at least one time?

int iOnce = 0;

```
Thread 1                    Thread 2
-----------                 -----------
pending = 1;                while (iOnce == 0)
while (pending) {             ;
  printf("hello\n");        pending = 0;
  iOnce = 1;
}
```

7. **Lock Implementation.** You are given a new atomic primitive, called FetchAndSubtract(). It executes as a single atomic instruction, and is defined as follows:

```
int FetchAndSubtract(int *location) {
  int value = *location; // read the value pointed to by location
  *location = value - 1; // decrement it, and store result back
  return value; // return old value
}
```

You are given the task: write the lock_init(), lock(), and unlock() routines (and define a lock_t structure) that use FetchAndSubtract() to implement a working lock. (You will get 10 points as bonus if you write two significantly different implementations)

```
typeset struct _lock_t {
  int ticket;
  int turn;
} lock_t;
lock_init(lock_t *lock) {
  lock->ticket = lock->turn = 0;
}
lock((lock_t *lock) {
  int myturn = FetchAndSubstract(&lock->ticket);
  while (myturn != lock->turn)
    ;
}
unlock((lock_t *lock) {
  FetchAndSubstract(&lock->turn);
}
```

The above is a ticket lock. A simpler solution:
init: set lock to something
lock: while (FetchAndSubstract(&lock) < something) ;
unlock: set lock to something