CS-537: Midterm (Spring 2012)

Subtle Differences

Please Read All Questions Carefully!

There are nine (9) total numbered pages.

Please put your FULL NAME (mandatory) on THIS page only.

Name: Sol ution

Grading Page

	Points	Total Possible
Q1		10
Q2		10
Q3		10
Q4		10
Q5		10
Q6		10
Q7		10
Q8		10
Q9		10
Q10		15
Q11		15
Q12		10
Q13		10
Q14		10
Q15		10
Total		160

"It pays to be obvious, especially if you have a reputation for subtlety." -Isaac Asimov

In life, sometimes a subtle difference makes no difference: you put on some blue pants instead of some green ones.

Other times, a subtle difference makes all the difference in the world: you forget to put on any pants at all.

In this exam, 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.

Remember to read all of the questions carefully. Also note that two questions are worth a little more than the rest. Finally, good luck!

1. 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)

Most answers accepted, assuming a reasonable discussion of possible effects on response time. Mentioning possible starvation effect on front-of-queue ⇒ bonus.

2. You write a UNIX shell, but instead of calling fork() then exec() to launch a new job, you instead insert a subtle difference: the code first calls exec() and then calls fork(). What is the impact of this change to the shell, if any? (Explain)

New shell:

```
exec(cmd, args);
fork();
```

Doesn't work! Shell replaced w/ new command, never forks!

3. 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?

General effect:
moves all jobs to top, thus lessening starvation + also more rapidly re-evaluating behavior of job.
Problem: if done too often, scheduler becomes more + more like RR

4. The lottery scheduler relies on random numbers in order to pick the winner of a lottery. This subtly-different lottery scheduler uses a simplified random number generator, which rotates through the following five pseudo-random numbers: 133, 12, 800, 442, 917. How does this change affect the behavior of the lottery scheduler?

A small fixed # RNG is bad for lottery as some jobs will never be chosen, repeatedly. Starvation, not behaving as expected, etc.

5. 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 fundamentally 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.

6. A TLB often has a valid bit in each entry; the valid bit tells us whether the the particular TLB entry should be examined when looking for translations. In this subtle difference, the TLB has no such valid bit. What are the implications of such a difference?

valid bit in TLB is useful to enable h/w to know which translations should be checked for hit.

w/o one, have to use TLB much more carefully, ensuring that only valid translations are present at all times (starting @ boot!)

7. In a subtly-different system with a software-managed TLB, the OS does not install a translation into the TLB upon the first TLB miss on a particular virtual address. Rather, it increments a counter in the page table entry (PTE). When that counter reaches 3, this subtly-different approach then updates the TLB with the desired translation. How does this lazy TLB update affect the behavior of the system?

This type of policy can be bad (it may cause more misses) or good (only really valuable translations are present), depends strongly on workload.

8. With base-and-bounds based virtual memory, two registers (base and bounds) are used to implement a primitive form of virtualization. The subtle change we explore here is to the bounds register. Specifically, in this subtly-different base-and-bounds, the bounds register is checked only on writes to memory, and not on reads from memory. What is the impact of this change?

Good: can't overwrite memory that isn't yours

Bad: might leak sensitive info from process -> process or OS -> process

9. In a page table, a per-page reference bit is sometimes used to help track which pages are being actively used. A subtle change to the per-page reference bit makes it into a per-page 32-bit counter; when a page is referenced, the corresponding counter is incremented. What is the impact of this change on page replacement within the OS? Can (or should) the OS policy be changed to make use of it?

Bad: more space needed to store info need to change OS alg to use extra bits

Good: can possibly get a much better sense of how pages are being used (a lot vs. a little) and thus make better replacement decisions

10. In this subtle difference, we have changed the theme of the exam entirely to allow you to do this one question from the homeworks without any subtle changes at all. It is, of course, the multi-level page table question. Assume (as always) a 15-bit virtual address, with a 32-byte page size, and a two-level page table. The format of both the page directory entry (PDE) and the page table entry (PTE) is the same: a valid bit followed by a 7-bit page frame number. The page directory base register (PDBR) is set to decimal 73. Here is a dump of memory (OK, there is one subtle difference; instead of all of the pages, we only give you the subset of physical memory you need to see):

```
pg   6: 0a 1c 01 14 0b 1a 19 0a 0a 1a 0c 14 02 0c 1c 0c 15 04 0e 13 17 11 08 05 08 07 04 13 0f 1d 0f 1e
pg  73: a2 d2 97 96 d9 7f 87 b4 b7 f2 f4 82 bf 7f be 93 e8 9d 99 9e f1 7f 7f b0 d8 da eb b1 81 c3 c2 f6
pg 114: 7f 7f 7f 7f 82 7f 7f 7f 7f 7f 7f 7f 99 7f 7f 7f 7f 7f 7f 86 7f 7f 7f 7f 7f 7f 8f 7f 7f 7f 7f 7f
```

The first virtual address is to translate is 0x1787. What happens when you try to load this virtual address? (if valid, what value do you get back?)

0x1787 ⇒ 00101 | 11100 | 00111
⇓
5th entry on page 73 ⇒ x7f ⇒ 0 111 1111 → not valid, fault

The second virtual address in question is 0x2665. What happens when you try to load it? (if valid, what value do you get back?)

0x2665 ⇒ 01001 | 10011 | 00101 → 5th entry on p. 6 ⇒ x1a
⇓
9th entry on page 73 ⇒ 0xf2 ⇒ 1 1110010 → valid, page 114
19th entry on page 114 ⇒ 0x86 ⇒ 1 0000110 → valid, page 6

11. A simple page table entry (PTE) usually contains at least some kind of valid bit followed by the page-frame number. In this subtly-different approach, the order of the two is switched. Thus, the entry contains a PFN followed by a valid bit. Assume the following linear page table for a virtual address space of size 128 bytes with 32-byte pages:

```
0xFE
0x0F
0xFE
0x09
```

0xFE ⇒ 1111 111 0 — not valid
0x0F → 0000 111 1 — valid = page 7
0xFE ⇒ — not valid
0x09 ⇒ 0000 100 1 — valid ⇒ page 4

0 → 31 fault
32 → 63 valid on PFN 7
64 → 95 fault
96 → 127 valid on PFN 4

For the following virtual addresses, say whether each is a valid virtual address or a fault; for those that are valid, what physical address results from the translation?

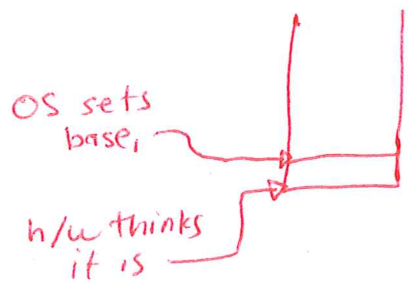
```
VA 0x065 (decimal: 101) --> ?
VA 0x00c (decimal:  12) --> ?
VA 0x026 (decimal:  38) --> ?
VA 0x058 (decimal:  88) --> ?
```

0x85
fault
0xE6
fault

0x65 ⇒ 11 | 0 0101 → 100 0 | 0 101 → 0x85

0x26 ⇒ 01 | 0 0110 → xlate → 111 0 | 0110 → 0xE6

12. Assume we have a system that uses segmentation to provide a virtual memory to processes. Assume the segmentation chops the address space into two parts (segment 0 and 1); segment 0 grows in the increasing direction, while segment 1 grows backwards. Unfortunately, there is confusion over the interpretation of the base register of segment 1; while the hardware thinks it should point to the physical address one beyond the bottom of the backwards-growing segment, the OS has been subtly changed to assume it points to the last byte of the backwards-growing segment. Describe what would happen when running processes on this subtly-changed virtual memory system.



result: chaos!
OS thinks it is OK to access last byte on stack → h/w doesn't
⇒ likely fault

13. This code snippet provides a "solution" to the producer/consumer problem, in which a bounded buffer is shared between producer threads and consumer threads. The solution uses a single lock (m) and two condition variables (fill and empty).

```
Consumer:
  while (1) {
    mutex_lock(m);
    if (numfull == 0)
      wait(fill, m);
    tmp = get();
    signal(empty);
    mutex_unlock(m);
  }
```

```
Producer
  for (i = 0; i < 10; i++) {
    mutex_lock(m);
    if (numfull == MAX)
      wait(empty, m);
    put();
    signal(fill);
    mutex_unlock(m);
  }
```

Some subtle changes are made to the condition variable library: wait() is changed so that it never wakes up unless signaled; more importantly, signal() is changed so that it immediately transfers control to a waiting thread (if there is one), thus implicitly passing the lock to the waiting thread as well. Does this subtle change affect the correctness of the producer/consumer solution above? (Describe)

Code above usually doesn't work because of failure to re-check condition

Change of semantics of signal makes it work, as now waiting thread knows condition is true when it awakes (no need to recheck)

See Mesa vs. Hoare semantics in book for details

14. A spin lock acquire() can be implemented with an atomic exchange instruction as follows:

```
while (xchg(&lock, 1) == 1)
  ; // spin
```

Recall that xchg() returns the old value at the address while atomically setting it to a new value. This new subtly-different lock acquire() is implemented as follows:

```
while (1) {
  while (lock > 0)
    ; // spin
  if (xchg(&lock, 1) == 0)
    return;
}
```

T1 T2

What kind of difference does this new lock make? Does it work? How does it change the behavior of the lock?

Lock works! Just spins until it thinks the lock is free (T1 above) then uses atomic exchange to acquire; if fails, try again! called a test+test-and-set lock (used because xchg is expensive)

15. Observe the following multi-thread safe list insertion code:

```
typedef struct __node_t {
    int              key;
    struct __node_t *next;
} node_t;

mutex_t m     = PTHREAD_MUTEX_INITIALIZER;
node_t *head = NULL;

int List_Insert(int key) {
    mutex_lock(&m);
    node_t *n = malloc(sizeof(node_t));
    if (n == NULL) { return -1; } // failed to insert
    n->key  = key;
    n->next = head;
    head    = n;                  // insert at head
    mutex_unlock(&m);
    return 0;                     // success!
}
```

← bad!

The code has some problems alas. In this final question, *you* need to insert a subtle change to fix the code and make it work correctly (feel free to augment the code above with your answer).

Code returns w/ lock held if malloc fails

easy fix! {
mutex_unlock(&m);
return -1;
}