CS 537: Introduction to Operating Systems
Fall 2015: Midterm Exam #2

This exam is closed book, closed notes. All cell phones must be turned off. No calculators may be used.

You have two hours to complete this exam.

There are two parts to this exam: the first is true/false, the second is multiple choice. Some of the T/F questions are very simple, and some will take you awhile to determine. We expect you'll end up spending approximately equal amounts of time on each part.

Write all of your answers on the accu-scan form with a #2 pencil.

These exam questions must be returned at the end of the exam, but we will not grade anything in this booklet.

Unless stated (or implied) otherwise, you should make the following assumptions:

- The OS manages a single uniprocessor
- All memory is byte addressable
- The terminology lg means $\log_2$
- 2^{10} bytes = 1KB
- 2^{20} bytes = 1MB
- Page table entries require 4 bytes
- Data is allocated with optimal alignment, starting at the beginning of a page
- Assume leading zeros can be removed from numbers (e.g., 0x06 == 0x6)
- The system contains multiple processors.

Good luck!

Part 1: Straight-forward True/False from Virtualization [1 point each]
Designate if the statement is True (a) or False (b).

1) Two processes reading from the **same virtual address** will access the same contents.
2) Two threads reading from the **same virtual address** will access the same contents.
3) On a uniprocessor system, there may only be one **ready** process at any point in time.
4) The **convoy effect** occurs when shorter jobs must wait for longer jobs.
5) **Stacks** are used for procedure call frames, which include local variables and parameters.
6) A **virtual address** is identical to a **logical address**.
7) With **dynamic relocation**, hardware dynamically translates an address on every memory access.
8) The OS may not manipulate the contents of an MMU.
9) With pure segmentation (and no other support), fetching and executing an instruction that performs a store from a register to memory will involve exactly **two memory references**.
10) Paging approaches suffer from **internal fragmentation**, which grows as the size of a page grows.
11) The **number** of virtual pages is always identical to the number of physical pages.
12) A TLB **caches** translations from full virtual addresses to full physical addresses.
13) **TLB reach** is defined as the number of TLB entries multiplied by the size of each TLB entry.
14) On a **context switch**, the TLB must be flushed to ensure that one process cannot access the memory of another process.
15) A longer scheduling time slice is likely to decrease the overall TLB miss rate in the system.
16) If the **valid bit** is clear (equals 0) in a PTE needed for a memory access, the running process is likely to be killed by the OS.
17) There is a separate page table for every active process in the system.
18) An **inverted page table** is efficiently implemented in hardware.
19) TLBs are more beneficial with multi-level page tables than with linear (single-level) page tables.
20) When a page fault occurs, it is less expensive to replace a clean page than a dirty page.

Part 2: Straight-forward True/False from Concurrency [2 points each]
Designate if the statement is True (a) or False (b).

21) Threads that are part of the same process share the same code segment.
22) Threads that are part of the same process share the same heap.
23) Threads that are part of the same process share the same Program Counter.
24) Context-switching between threads of the same process requires flushing the TLB or tracking an ASID in the TLB.
25) With user-level threads, if one thread of a process blocks, all the threads of that process will also be blocked.
26) A mutex is identical to a lock.

27) As long as a context-switch does not occur within a critical section, the code within a critical section will execute atomically and no race condition will occur.
28) If a lock guarantees progress, then it is deadlock-free.
29) A lock that performs spin-waiting can provide fairness across threads (i.e., threads receive the lock in the order they requested the lock).
30) Lock implementations for modern hardware rely on the fact that single word loads and stores are atomic.
31) A lock implementation should block instead of spin if it only be used on a uniprocessor.
32) A lock implementation should block instead of spin if it is known that the lock will not be acquired for a relatively long time.
33) Periodically yielding the processor while spin waiting reduces the amount of wasted time to be proportional to the duration of a time-slice.
34) A condition variable can be used to provide mutual exclusion.
35) For a thread to call wait() on a condition variable, that thread must first hold a lock related to that condition variable.
36) With producer/consumer relationships and a finite-sized circular shared buffer, producing threads must wait until there is an empty element of the buffer.
37) Broadcasting to a condition variable is likely to have greater performance implications when there are more waiting threads.
38) Semaphores and condition variables are equivalent.
39) To implement a thread_join() operation with a semaphore, the semaphore should be initialized to 1.
40) To implement a thread_join operation with a semaphore, the thread_join() code will call sem_post().
41) A goal of a reader/writer lock is to ensure that either just one reader or just one writer can hold the lock.
42) Atomicity problems can be most easily fixed by using condition variables.
43) Deadlock requires four conditions: ordering of requests, hold-and-wait, no preemption, and circular waiting.
44) Livelock is identical to deadlock.
45) One way to ensure that deadlock does not occur is to require that locks are acquired in a fixed, linear order.

Part 3. Fork and Thread_Create() [15 points?]

For the next two questions, consider the following code:

```
int a = 0;
for (int i = 0; i < 4; i++) {
    if (fork() == 0) {
        printf("Hello!\n");
        a++;
        exit(1);
    } else {
        printf("Hello!\n");
        a++;
    }
}
printf("a is %d\n", a);
```

46) How many times will the message "Hello!\n" be displayed?
   a) 4
   b) 8
   c) 16
   d) 32
   e) None of the above

47) What will be the final value of "a" as displayed in the final line of the program?
   a) Due to race conditions, "a" may have different values on different runs of the program.
   b) 0
   c) 4
   d) 8
   e) None of the above

Imagine similar questions, but for example code that starts up multiple threads (instead of processes). Also understand what values will be over time for variables that have been allocated on the heap or the stack.

Part 4. Project 2b – Expected behavior of MLFQ [10 points?]

Imagine you are shown a graph; along the x-axis is time; along the y-axis is the priority of the process that was scheduled at that point in time. Given the MLFQ scheduler you implemented in Project 2b, what probably happened in the workload at interesting points in time to cause the resulting graph?

Part 4. Understanding impact of scheduling interleaving given assembly code (e.g., homework simulations) [20 points?]

For the next questions, assume that two threads are running the following code (this is the same `looping-race-nolock.s` code you may have seen in homework simulations). This code is incrementing a variable (e.g., a shared balance) many times in a loop; there is no locking.

```
# assumes %bx has loop count in it
.main
.top
# critical section
mov 2000, %ax  # get the value at the address
add $1, %ax    # increment it
mov %ax, 2000  # store it back

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

Assume that the %bx register begins by holding the value 3, so that each thread will perform the critical section 3 times. Assume that the address 2000 originally contains the value 0.
Assume that the scheduler runs the two threads producing the following order of instructions:

```
Thread 0                 Thread 1
1000 mov 2000, %ax                                48)
1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
                         1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
1002 mov %ax, 2000
------ Interrupt ------  ------ Interrupt ------
                         1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
                         1002 mov %ax, 2000       49)
                         1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
1004 test $0, %bx
1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
                         1004 test $0, %bx
                         1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
1000 mov 2000, %ax
1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
                         1000 mov 2000, %ax       50)
                         1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
```

```
1002 mov %ax, 2000
1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
                         1002 mov %ax, 2000          51)
------ Interrupt ------  ------ Interrupt ------
1004 test $0, %bx
1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
                         1003 sub  $1, %bx
                         1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
1000 mov 2000, %ax
1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
                         1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
1002 mov %ax, 2000
1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
                         1000 mov 2000, %ax
                         1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                         1002 mov %ax, 2000          52)
                         1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
                         1004 test $0, %bx
                         1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
1006 halt
----- Halt;Switch -----  ----- Halt;Switch -----
------ Interrupt ------  ------ Interrupt ------
                         1006 halt
```

For each of the designated lines, determine the contents of the memory address 2000 AFTER that assembly instruction executes.

a) 0
b) 1
c) 2
d) 3
e) None of the above

(If you want to see the answer, this instruction stream was generated by running: `./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000 -i 2 -r -c`)

Part 6. Ticket Locks [10 points?]

Imagine you have the following implementation of ticket locks:

```
int FAA(int *ptr) { // assume this is atomic
    int old = *ptr;
    *ptr = old + 1;
    return old;
}

typedef struct __lock_t {
    int ticket;
    int turn;
}
void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while (lock->turn != myturn); // spin
}
void release (lock_t *lock) {
    lock->turn++;
}
```

Imagine you have the following stream of requests across three threads A, B, and C.

A: acquire() 53) myturn?
B: acquire()
C: acquire()
A: release() 54) turn?
B: release() 55) which process will acquire lock?
B: acquire()
A: acquire() 56) myturn?
C: release() 57) turn? 58) which process will acquire lock?

For questions 53, 54, 56, 57, given the specified point in time, what will be the value of either myturn for the thread in question or the global value of turn when the function completes? Your choices are:

a) 0
b) 1
c) 2
d) 3
e) None of the above

For questions 55 and 58, given the specified point in time, which process will acquire the lock at this point? Your choices are:

a) Thread A
b) Thread B
c) Thread C
d) Race condition such that no particular thread is guaranteed to acquire lock

Part 7. Impact of scheduling multi-threaded C code (given hardware atomic instructions, locks, condition variables, or semaphores) [50 points?]

Assume the done variable has been properly initialized. Imagine you have the following implementation of thread_join() and thread_exit():

```
void thread_join() {
        Mutex_lock(&m);          // p1
        while (done == 0)        // p2
            Cond_wait(&c, &m);   // p3
        Mutex_unlock(&m);        // p4
}

void thread_exit() {
        Mutex_lock(&m);          // c1
        done = 1;                // c2
        Cond_signal(&c);         // c3
        Mutex_unlock(&m);        // c4
}
```

Imagine there is a parent thread, P, and a child thread, C. Assume you have a scheduler that runs P and C such that each statement in the C-language language code is atomic. We will tell you which thread was scheduled to run by showing you either a "P" (for the parent) or a "C" (for the child) to designate that one **line** of C-code was scheduled.

For example, a stream PPC means that the parent thread executed "mutex lock()" and the test for the line "while (done == 0)", and then the child thread executed one statement in "mutex_lock()".

Function calls that may have to wait for something to happen (e.g., mutex_lock() and cond_wait()) are treated specially.

For mutex_lock()), assume that function call requires one scheduling interval if the lock is currently available. If the lock is not available, assume the call spin-waits until the lock is available (e.g., if the parent thread holds the mutex lock, and the child tries to grab the lock, you may see a long instruction stream CCCCCCC that causes no progress for the child thread); once the lock is available, the next scheduling of the acquiring thread will cause that thread to obtain the lock (e.g., after the parent releases the lock, the next "C" will cause mutex_lock() to complete).

The rules for cond_wait are similar. When the parent thread calls cond_wait(), no matter how long the scheduler runs the parent (e.g., PPPPPPPPPP), the parent thread will be waiting in cond_wait(). After the child runs and does the work necessary for the cond_wait() to complete, then the next scheduling of the parent (i.e., the next "P") will cause the cond_wait() line to complete.

If an instruction stream shows a thread being scheduled past the end of the code (e.g., CCCCCC), assume the thread executes instructions that are not relevant to the shown code.

Assume the instruction stream PPCCCCC.

(some answers are in bold to get you started…)

59) Which line will the parent run when it is scheduled again?

a) p1

b) p2

c) p3
d) p4
e) Code beyond p4

60) Which line will the child run when it is scheduled again?

a) c1
b) c2
c) c3
d) c4
e) Code beyond c4

Assume the scheduler continues on with PPPPC.

61) Which line will the parent run when it is scheduled again?

a) p1
b) p2
c) p3
d) p4
e) Code beyond p4

62) Which line will the child run when it is scheduled again?

a) c1
b) c2
c) c3
d) c4
e) Code beyond c4

Assume the scheduler continues on with CCPPP.

63) Which line will the parent run when it is scheduled again?

a) p1
b) p2
c) p3
d) p4
e) Code beyond p4

64) Which line will the child run when it is scheduled again?

a) c1
b) c2
c) c3
f) c4
g) Code beyond c4

Assume the scheduler continues on with CCPPP.

65) Which line will the parent run when it is scheduled again?

a) p1
b) p2
c) p3
d) p4

e) Code beyond p4

66) Which line will the child run when it is scheduled again?

a) c1
b) c2
c) c3
d) c4
e) Code beyond c4

You can assume there will be multiple code examples of this form for you to step through. Some will have race conditions and some will not.