CS 537: Introduction to Operating Systems
Fall 2019: Midterm Exam #2
November 6, 2019

This exam is closed book, closed notes.

All cell phones must be turned off and put away.

No calculators may be used.

You have two hours to complete this exam.

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:

- 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).

There are 90 questions on this exam.

Good luck!

Virtualization Review [1 point each]
Designate if each statement is True (a) or False (b).

1) On a timer interrupt, the OS is responsible for transitioning from user to kernel mode.
2) Increasing the time-slice of an RR scheduler is likely to decrease the overhead imposed by scheduling.
3) Increasing the time-slice of an RR scheduler makes the scheduler behave more like FCFS.
4) An MLFQ scheduler assumes the presence of an oracle that knows the length of a job's CPU burst in advance.
5) The fork() system call creates a child process whose execution begins at the entry point main().
6) A process can close() the stdout file descriptor.
7) With dynamic relocation, the OS determines where the address space of a process is allocated in virtual memory.
8) With dynamic relocation, the OS can move an address space after it has been placed in physical memory.
9) A Gantt chart illustrates how virtual pages are mapped to physical pages
10) With pure segmentation (with no paging or TLBs), each segment of each process must be allocated contiguously in physical memory.
11) With pure segmentation (with no paging or TLBs), fetching an instruction will involve exactly two memory references.
12) Threads running within the same process are likely to share the same value for the base register, but not the bounds register, in the MMU.
13) Given a constant number of bits in a virtual address, the size of a linear page table increases with more (virtual) pages.
14) Given a fixed number of virtual pages, the size of a linear page table decreases with a smaller address space.
15) Given a two-level page table, there may be two TLB misses when performing a virtual to physical address translation.
16) If the TLB does not support ASIDs, all page table entries (PTEs) are marked invalid on a context switch.
17) With a linear page table, all virtual pages within an address space must be allocated.
18) Given a 2-level page table (and no TLB), exactly 3 memory accesses are needed to fetch each instruction.
19) LRU with N+1 physical pages will contain (cache) the same virtual pages as LRU with N pages, plus the contents of one more virtual page.
20) Demand paging loads the address space of a process into physical memory when that process is started
21) The clock algorithm must be able to determine if a page has been accessed in some particular time interval

Concurrency [2 points each]
Designate if the statement is **True (a)** or **False (b).**

22) The clock frequency of CPUs has been increasing exponentially each year since the VAX-11.
23) Threads that are part of the same process share the page tables.
24) Threads that are part of the same process share the same registers.
25) Context-switching between threads of the same process requires flushing the TLB
26) A problem with user-level thread implementations is that if one thread in the application process blocks, then all the threads in that application process are blocked as well.
27) Locks prevent the OS scheduler from performing a context switch during a critical section.
28) The **volatile** key word before a variable directs the compiler to protect that variable with locks.
29) Peterson's algorithm is able to provide mutual exclusion using simple load and store instructions for two threads running on a modern multiprocessor.
30) A context switch can occur in the middle of the atomic hardware instruction XCHG.
31) Atomic hardware instructions can be used as building blocks for providing mutual exclusion to user applications on multiprocessor systems.
32) It is possible for a lock implementation to provide fairness across threads (i.e., threads receive the lock in the order they requested the lock) with spin-waiting.
33) To minimize wasted CPU cycles on a multiprocessor system, a lock implementation should block instead of spin if the lock will held by another process for longer than the time required for a context-switch.
34) A ticket lock must atomically increment the ticket number that a thread is given when it tries to acquire the lock.
35) Two-phase waiting assumes that an oracle can predict the amount of time that the lock will be held by another process.
36) When a thread calls cond_wait() on a condition variable, cond_wait() returns immediately if a cooperating thread previously called cond_signal() on that condition variable.
37) The call cond_wait() must release a corresponding mutex.
38) With producer/consumer relationships and a circular shared buffer containing N elements, producing threads must wait when there is exactly 1 empty element in the buffer.
39) The performance of a multi-threaded application that uses broadcasting to a condition variable is likely to decrease as the number of waiting threads increases.
40) Condition variables can provide both ordering and mutual exclusion.
41) Semaphores can provide both ordering and mutual exclusion.
42) Semaphores can be built on top of condition variables and locks.

43) As the amount of code a mutex protects increases, the amount of concurrency in the application increases.
44) To implement a thread_join() operation with a semaphore, the semaphore is initialized to the value of 0 and the thread_join() code calls sem_wait().
45) The safety property for dining philosophers states that no two adjacent philosophers are eating simultaneously.
46) With a reader/writer lock, multiple readers can hold the lock simultaneously with a single writer.
47) With a wait-free algorithm, a thread that cannot acquire a lock must release any other locks that the thread holds.
48) Deadlock can be avoided if locks are always acquired in a well-defined, strict linear order.

Processes and Fork [7 total points]

Assume the following code is compiled and run on a modern Linux machine. Assume the code compiles correctly and fork() never fails.

```
volatile int a = 0;
main() {
   int b = 0;              // 1
   a++;                    // 2
   int rc = fork();        // 3
   b++;                    // 4
   fork();                 // 5
   if (rc == 0) {          // 6
         fork();           // 7
         a++;              // 8
   }
   b++;                    // 9
}
```

49) How many processes will execute **line 2** of this program?
   a) 1
   b) 2
   c) 4
   d) 8
   e) None of the above

50) How many processes will execute **line 4** of this program?
   a) 1
   b) 2
   c) 4
   d) 8
   e) None of the above

51) How many processes will execute **line 6** of this program?
   a) 1
   b) 2
   c) 4
   d) 8
   e) None of the above

52) How many processes will execute **line 8** of this program?
   a) 2
   b) 4
   c) 6
   d) 8
   e) None of the above

53) What will be the value of variable **b** after **line 4** has executed?
   a) Known, but will be different for different processes
   b) Unknown; may have unexpected values due to race conditions
   c) 1
   d) 2
   e) None of the above

54) What will be the value of variable **b** after **line 9** has executed?
   a) Known, but will be different for different processes
   b) Unknown; may have unexpected values due to race conditions
   c) 2
   d) 4
   e) None of the above

55) What will be the value of variable **a** after **line 9** has executed?
   a) Known, but will be different for different processes
   b) Unknown; may have unexpected values due to race conditions
   c) 2
   d) 4
   e) None of the above

Threads [9 total points]

Assume the following code is compiled and run on a modern Linux machine. Assume the code compiles correctly and none of the system calls fail. Look at the code carefully; you probably haven't seen this exact version before…

```
volatile int balance = 0;
pthread_mutex_t add_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t sub_mutex = PTHREAD_MUTEX_INITIALIZER;

void *add(void *arg) {
   for (int i = 0; i < 200; i++) {
         pthread_mutex_lock(&add_mutex);
         balance++;
         pthread_mutex_unlock(&add_mutex);
   }
   printf("Balance is %d\n", balance);
   return NULL;
}
void *sub(void *arg) {
   for (int i = 0; i < 200; i++) {
         pthread_mutex_lock(&sub_mutex);
         balance--;
         pthread_mutex_unlock(&sub_mutex);
   }
   printf("Balance is %d\n", balance);
   return NULL;
}
int main(int argc, char *argv[]) {
   pthread_t p1, p2;

   pthread_create(&p1, NULL, add, "A");
   pthread_create(&p2, NULL, sub, "B");
   pthread_join(p1, NULL);
   pthread_join(p2, NULL);

   printf("Final Balance is %d\n", balance);
}
```

56) When thread p1 prints "Balance is %d\n", what will p1 say is the value of `balance`?
   a) Due to race conditions or other timing differences, "balance" may have different values at this point on different runs of the program.
   b) -200
   c) 0
   d) 200
   e) None of the above

57) When thread p2 prints "Balance is %d\n", what will p2 say is the value of `balance`?
   a) Due to race conditions or other timing differences, "balance" may have different values at this point on different runs of the program.
   b) -200
   c) 0
   d) 200
   e) None of the above

58) When the main parent thread prints "Final Balance is %d\n", what will the parent thread say is the value of `balance`?
   a) Due to race conditions or other timing differences, "balance" may have different values at this point on different runs of the program.
   b) -200
   c) 0
   d) 200
   e) None of the above

Thread Join with Condition Variables [15 total points]
Your project partner wrote code to implement thread_join() and thread_exit() with locks and condition variables, but it is a little bit different than what you saw in lecture. Specifically, lines c2 and c3 have been swapped. Assume the code compiles correctly. It is your job to complete the initialization and to then determine whether or not this implementation is correct.

```
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
        Cond_signal(&c);         // c2
        done = 1;                // c3
        Mutex_unlock(&m);        // c4
}
```

Assume `thread_create()` which the parent thread invokes to create and start the child thread has the following signature:

```
int thread_create(pthread_t *thread,
                  void *(*start_routine)(void *), void *arg)
```

59) Ignoring any new problems your partner may have introduced, to what value should the **done** variable be initialized in order to join the parent and child?
a) -1
b) 0
c) 1
d) 2
e) None of the above

60) Ignoring any new problems your partner may have introduced, to what value should the **condition variable, c**, be initialized?
a) -1
b) 0
c) 1
d) 2
e) None of the above

Again, ignoring any problems your partner may have introduced, which of the following thread library routines could perform this initialization of **done** and the condition variable **c** (without causing any race conditions)? Assume that these variables are not initialized while a mutex lock is held. Note: **at least one** of these options is a correct possibility.

Answer **(a) yes, a correct possibility**, or **(b) no, could cause a race condition**

61) `thread_create()` in the parent thread before the `start_routine` of the child thread is called
62) `thread_create()` in the parent thread after the `start_routine` of the child thread is called
63) In the child thread before beginning its designated `start_routine`
64) `thread_join()`: As the first line of thread_join
65) `thread_exit()`: As the first line of thread_exit

66) Now, assuming everything has been initialized correctly, does your project partner's version of `thread_join()` and `thread_exit()` work correctly?
   a) **Only** if the parent calls `thread_join()` and `cond_wait()` before the child calls thread_exit()
   b) **Only** if the child completes `thread_exit()` before the parent calls `thread_join()`
   c) **Only** if there is **not** a context switch between when the child calls `cond_signal()` and sets `done = 1`
   d) Works correctly in all cases
   e) None of the above

Reader/Writer Locks with Semaphores [20 points total]
In this question, you will explore your understanding of the reader/writer lock code with semaphores. The code below is identical to that shown in class. No changes were made and no bugs were introduced (ignore any incorrect/inconsistent capitalization). Assume the code compiles correctly.

```
Acquire_readlock() {
   Sem_wait(&mutex);           // AR1 (line 1 of Acquire_readlock)
   If (ActiveWriters +         // AR2
        WaitingWriters==0) {   // AR3
         sem_post(OKToRead);   // AR4
         ActiveReaders++;      // AR5
   } else WaitingReaders++;    // AR6
   Sem_post(&mutex);           // AR7
   Sem_wait(OKToRead);         // AR8
}
Release_readlock() {
   Sem_wait(&mutex);           // RR1 (line 1 of Release_readlock
   ActiveReaders--;            // RR2
   If (ActiveReaders==0 &&     // RR3
     WaitingWriters > 0) {     // RR4
         ActiveWriters++;      // RR5
         WaitingWriters--;     // RR6
         Sem_post(OKToWrite);  // RR7
   }
   Sem_post(&mutex);           // RR8
}
Acquire_writelock() {
   Sem_wait(&mutex);           // AW1 (line 1 of Acquire_writelock)
   If (ActiveWriters + ActiveReaders + WaitingWriters==0) { // AW2
       ActiveWriters++;        // AW3
       sem_post(OKToWrite);    // AW4
   } else WaitingWriters++;    // AW5
   Sem_post(&mutex);           // AW6
   Sem_wait(OKToWrite);        // AW7
}
Release_writelock() {
   Sem_wait(&mutex);           // RW1
   ActiveWriters--;            // RW2
   If (WaitingWriters > 0) {   // RW3
      ActiveWriters++;         // RW4
      WaitingWriters--;        // RW5
      Sem_post(OKToWrite);     // RW6
   } else while(WaitingReaders>0) { // RW7
      ActiveReaders++;         // RW8
      WaitingReaders--;        // RW9
      sem_post(OKToRead);      // RW10
   }
   Sem_post(&mutex);           // RW11
```

67) Does this implementation of reader/writer locks give priority to readers or to writers? That is, if there is both a reader and a writer waiting to acquire this lock, which one will be given the lock next?

a) The reader
b) The writer
c) The thread that called acquire_readlock() or acquire_writelock() first
d) Strictly alternates between readers and writers
e) Cannot determine because cannot make assumptions about behavior of scheduler

68) How should the semaphore mutex() be initialized?

a) 0
b) 1
c) To the number of reader threads
d) To the number of writer threads
e) None of the above

69) How should the semaphore OKToRead be initialized?

a) 0
b) 1
c) To the number of reader threads
d) To the number of writer threads
e) None of the above

70) How should the semaphore OKToWrite be initialized?

a) 0
b) 1
c) To the number of reader threads
d) To the number of writer threads
e) None of the above

Assume the following calls are made by threads in the system and all three threads execute as far along as they can until they execute a statement that causes them to block (or wait). Assume that after a thread returns from one of these four functions, the thread executes other user code that does not involve any synchronization or blocking (i.e., some code beyond AR8, RR8, AW7, and RW11).

```
Reader Thread R0:     Acquire_readlock();
Writer Thread W0:     Acquire_writelock();
Reader Thread R1:     Acquire_readlock();
```

71) Where will reader thread R0 be in the code?

a) AR1
b) AR4
c) AR8
d) Beyond AR8
e) None of the above (including non-deterministic locations)

72) Where will thread W0 be in the code?

a) AW1
b) AW4
c) AW7
d) Beyond AW7
e) None of the above (including non-deterministic locations)

73) Where will thread R1 be in the code?

a) AR1
b) AR4
c) AR8
d) Beyond AR8
e) None of the above (including non-deterministic locations)

Continuing the same execution stream as above, now assume Thread R0 calls release_readlock() and the three threads again execute as far along as they can until they run into a statement that causes them to block (or wait).

74) Where will thread R0 be in the code?

a) RR1
b) RR7
c) RR8
d) Beyond RR8
e) None of the above (including non-deterministic locations)

75) Where will thread W0 be in the code?

a) AW1
b) AW4
c) AW7
d) Beyond AW7
e) None of the above (including non-deterministic locations)

76) Where will thread R1 be in the code?

a) AR1
b) AR4
c) AR8
d) Beyond AR8
e) None of the above (including non-deterministic locations)

77) Continuing the same execution stream, assume another read thread R2 begins and calls acquire_readlock() and executes as far along as it can before it must block. Where will thread R2 be in the code?

a) AR1
b) AR7
c) AR8
d) Beyond AR8
e) None of the above (including non-deterministic locations)

78) Continuing the same execution stream, when writer thread W0 eventually calls release_writelock(), what will happen next before threads must block?

a) Only Thread R1 will acquire the readlock, while R2 continues to wait for the readlock to be released
b) Thread R1 **or** R2 will acquire the readlock, but it is unknown which (the other waits for the readlock to be released)
c) First, Thread R1 will acquire the readlock, and then Thread R2 will acquire the readlock, while R1 still also has the readlock
d) Both Thread R1 and Thread R2 will acquire and concurrently hold the readlock, but the order in which each thread will return from acquire_readlock() is unknown
e) None of the above

Wait-Free Algorithms [8 total points]

Your project partner has written the following correct thread-safe implementation of `add()` using traditional locks for mutual exclusion:

```
void add (int *val, int amt) {
   mutex_lock(&m);
   *val += amt;
   mutex_unlock(&m);
}
```

You decide you would like to replace the lock with calls to the atomic hardware instruction `CmpAndSwap(int *addr, int expect, int new)`, which returns 0 on failure and 1 on success. The exact behavior of atomic `CmpAndSwap()` is defined as in class.

You know that `add()` modified to use `CmpAndSwap()` looks something like the following:

```
void add (int *val, int amt) {
   do {
        int old = *val;
   } while (CmpAndSwap(<Q1>, <Q2>, <Q3>) == <Q4>);
}
```

What are the correct replacements for the symbols < Q1, Q2, Q3, Q4> in the code above?

79) <Q1> should be replaced with
   a) val
   b) *val
   c) old + amt
   d) old
   e) None of the above

80) <Q2> should be replaced with
   a) val
   b) *val
   c) old +amt
   d) old
   e) None of the above

81) <Q3> should be replaced with
   a) val
   b) *val
   c) old+amt
   d) old
   e) None of the above

82) <Q4> should be replaced with
   a) 0
   b) 1
   c) val
   d) *val
   e) None of the above

Scheduling without locks given assembly code[16 total points]
For the next questions, assume that two threads are running the following code on a uniprocessor (this is the same `looping-race-nolock.s` code from homework simulations).

```
# assumes %bx has loop count in it
.main
.top
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
```

This code is incrementing a variable (e.g., a shared balance) many times in a loop. Assume that the %bx register begins with the value 3, so that each thread performs the loop 3 times. Assume the code is loaded at address 1000 and that **memory address 2000 originally contains the value 0**. Assume that the scheduler runs the two threads producing the following order of instructions (the first column shows the address of the executed instruction).

For each of the questions designated below, determine the contents of the %ax register or the memory address 2000 AFTER that assembly instruction executes.

a) 1
b) 2
c) 3
d) 4
e) None of the above

```
Thread 0                 Thread 1
1000 mov 2000, %ax
1001 add $1, %ax
1002 mov %ax, 2000
------ Interrupt ------  ------ Interrupt ------
                          1000 mov 2000, %ax        83) Contents of register %ax?
                          1001 add $1, %ax
                          1002 mov %ax, 2000        84) Contents of addr 2000?
                          1003 sub  $1, %bx
                          1004 test $0, %bx
                          1005 jgt .top
                          1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
1003 sub  $1, %bx
1004 test $0, %bx
1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
                          1001 add $1, %ax          85) Contents of register %ax?
------ Interrupt ------  ------ Interrupt ------
1000 mov 2000, %ax                                  86) Contents of register %ax?
1001 add $1, %ax
```

```
1002 mov %ax, 2000                                 87) Contents of addr 2000?
1003 sub  $1, %bx
1004 test $0, %bx
1005 jgt .top
1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
                        1002 mov %ax, 2000         88) Contents of addr 2000?
                        1003 sub  $1, %bx
                        1004 test $0, %bx
                        1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
1001 add $1, %ax
1002 mov %ax, 2000                                 89) Contents of addr 2000?
1003 sub  $1, %bx
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                        1000 mov 2000, %ax
                        1001 add $1, %ax
                        1002 mov %ax, 2000         90) Contents of addr 2000?
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
1006 halt
----- Halt;Switch -----  ----- Halt;Switch -----
                        1003 sub  $1, %bx
                        1004 test $0, %bx
                        1005 jgt .top
                        1006 halt
```

Congratulations on finishing!

See you in lecture tomorrow to talk more about file systems!