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.

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.

There are 83 questions on this exam.

Good luck!

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

1) **Cooperative multi-tasking** requires hardware support for a timer interrupt.
2) Two processes reading from the **same virtual address** will access the same contents.
3) The **convoy effect** occurs when longer jobs must wait for shorter jobs.
4) **Stacks** are used for procedure call frames, which include local variables and parameters.
5) A **RR scheduler** may preempt a previously running job.
6) The shorter the time slice, the more a RR scheduler gives similar results to a FIFO scheduler.
7) The OS provides the illusion to each thread that it has its own address space.
8) An **instruction pointer register** is identical to a program counter.
9) With **dynamic relocation**, hardware dynamically translates an address on every memory access.
10) The OS may **manipulate** the contents of an MMU.
11) With pure segmentation (and no other support), fetching and executing an instruction that performs an add of a constant value to a register will involve exactly **two memory references**.
12) Paging approaches suffer from **internal fragmentation**, which decreases as the size of a page increases.
13) The **number** of virtual pages is always identical to the number of physical pages.
14) A TLB **caches** translations from virtual page numbers to logical page numbers.
15) **TLB reach** is defined as the number of TLB entries multiplied by the size of a page.
16) A **TLB miss** is usually slower to handle than a **page miss**.
17) A single page can be **shared** across two address spaces by having each process use the same page table.
18) If the **present bit** is clear (equals 0) in a PTE needed for a memory access, the running process is likely to be killed by the OS.
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 more expensive to replace a **clean** page than a dirty page.

Part 2: Straight-forward True/False from Concurrency [3 points each]
Designate if the statement is True (a) or False (b).

21) The **clock frequency** of CPUs has been increasing **exponentially** each year since 1985.
22) Threads that are part of the same process share the same **stack pointer**.
23) Threads that are part of the same process share the same **page table base register.**
24) Threads that are part of the same process share the same **general-purpose registers**.
25) **Context-switching** between threads of the same process requires flushing the TLB or tracking an ASID in the TLB.

26) With **kernel-level threads**, if one thread of a process blocks, all the threads of that process will also be blocked.
27) The **hardware atomic exchange** instruction requires that interrupts are disabled during that instruction.
28) A lock that performs spin-waiting can provide **fairness** across threads (i.e., threads receive the lock in the order they requested the lock).
29) A lock implementation should always **block** instead of spin if it will always be used only on a uniprocessor.
30) On a multiprocessor, a lock implementation should **spin** instead of block if it is known that the lock will be available before the time of required for a context-switch.
31) Periodically **yielding** the processor while spin-waiting reduces the amount of wasted time to be proportional to the duration of a time-slice.
32) A **semaphore** can be used to provide **mutual exclusion**.
33) When a thread returns from a call to **cond_wait()** it can safely assume that it holds the corresponding mutex.
34) A thread calling **cond_signal()** will block until the corresponding mutex is available.
35) With **producer/consumer** relationships and a finite-sized circular shared buffer, consuming threads must wait until there is an empty element of the buffer.
36) The performance of **broadcasting** to a condition variable decreases as the number of waiting threads increases.
37) To implement a thread_join() operation with a condition variable, the thread_exit() code will call cond_wait().
38) A goal of **a reader/writer lock** is to ensure that either multiple readers hold the lock or just one writer holds the lock.
39) Building a condition variable on top of a semaphore is **easier** than building a semaphore over condition variables and locks.
40) As the **amount of code** a mutex protects increases, the **amount of concurrency** in the application increases.
41) **Ordering** problems in multi-threaded applications can be most easily fixed with locks.
42) A **wait-free** algorithm relies on condition variables instead of locks.
43) Deadlock can be avoided if threads acquire all potentially needed locks **atomically.**
44) Deadlock can be avoided by having only a **single lock** within a multi-threaded application.

Part 3. Fork and Thread_Create() [20 points]

For the next two questions, assume the following code is compiled and run on a modern Linux machine. Assume any irrelevant details have been omitted and that no routines, such as fork(), ever fail.

```
int a = 0;
fork();
a++;
fork();
a++;
if (fork() == 0) {
    printf("Hello!\n");
} else {
    printf("Goodbye!\n");
}
a++;
printf("a is %d\n", a);
```

45) How many times will the message "Hello!\n" be displayed?

a) 2
b) 3
c) 4
d) 6
e) None of the above

46) 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) 3
c) 4
d) 8
e) None of the above

For the next questions, assume the following code is compiled and run on a modern linux machine Assume any irrelevant details have been omitted and that no routines, such as pthread_create() or pthread_join(), ever fail.

```
volatile int balance = 0;

void *mythread(void *arg) {
      int result = 0;
      result = result + 200;
      balance = balance + 200;
      printf("Result is %d\n", result);
      printf("Balance is %d\n", balance);
      return NULL;
}

int main(int argc, char *argv[])
      pthread_t p1, p2;
      pthread_create(&p1, NULL, mythread, "A");
      pthread_create(&p2, NULL, mythread, "B");
      pthread_join(p1, NULL);
      pthread_join(p2, NULL);
      printf("Final Balance is %d\n", balance);
}
```

47) How many total threads are part of this process?

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

48) When thread p1 prints "Result is %d\n", what value of `result` will be printed?

a) Due to race conditions, "result" may have different values on different runs of the program.
b) 0
c) 200
d) 400
e) A constant value, but none of the above

48) When thread p1 prints "Balance is %d\n", what value of `balance` will be printed?

a) Due to race conditions, "balance" may have different values on different runs of the program.
b) 0
c) 200
d) 400
e) A constant value, but none of the above

49) When "Final Balance is %d\n is printed", what value of `balance` will be printed?

a) Due to race conditions, "balance" may have different values on different runs of the program.
b) 0
c) 200
d) 400
e) A constant value, but none of the above

Part 4. Impact of scheduling without locks (assembly code) [21 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 the 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). The code continues on the next page.

```
Thread 0                  Thread 1
1000 mov 2000, %ax
1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
                         1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
1002 mov %ax, 2000                                  50) Contents of addr 2000?
------ Interrupt ------  ------ Interrupt ------
                         1001 add $1, %ax
                         1002 mov %ax, 2000         51) Contents of addr 2000?
                         1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
                         1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                         1005 jgt .top
                         1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
1000 mov 2000, %ax
1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
                         1001 add $1, %ax
                         1002 mov %ax, 2000         52) Contents of addr 2000?
------ Interrupt ------  ------ Interrupt ------
1002 mov %ax, 2000                                  53) Contents of addr 2000?
------ Interrupt ------  ------ Interrupt ------
                         1003 sub  $1, %bx
```

```
                        1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
1003 sub  $1, %bx
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                        1005 jgt .top
                        1000 mov 2000, %ax
                        1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
                        1002 mov %ax, 2000          54) Contents of addr 2000?
                        1003 sub  $1, %bx
                        1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
1001 add $1, %ax
1002 mov %ax, 2000                                  55) Contents of addr 2000?
1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
                        1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                        1006 halt
----- Halt;Switch -----  ----- Halt;Switch -----
1005 jgt .top
1006 halt
```

For each of the lines designated above with a question numbered 50-55, determine the contents of the memory address 2000 AFTER that assembly instruction executes.

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

56) What would be the expected value for the final contents of address 2000 if there had been no race conditions between the two threads?

a) 3
b) 4
c) 5
d) 6
e) None of the above

Part 5. Spin-Locking with Atomic Hardware Instructions [10 points]

Consider the following partial implementation of a spin-lock. Examine it carefully because it is not identical to what was shown in lecture.

```
typedef struct __lock_t {
      int flag;
} lock_t;

void init(lock_t *lock) {
      lock->flag = abc;
}

void acquire(lock_t *lock) {
      while(xchg(&lock->flag, xyz) == xyz) ;
      // spin-wait (do nothing)
}

void release(lock_t *lock) {
      lock->flag = 1;
}
```

57) To what value or values could lock->flag be **initialized** to obtain a correct implementation (shown as `abc` above)?

a) 0
b) 1
c) 2
d) 0 or 2
e) 0 or 1

58) To what value or values could lock->flag be **compared** to in acquire() (shown as `xyz` above)?

a) 0
b) 1
c) 2
d) 0 or 2
e) 0 or 1

Part 6. Impact of scheduling multi-threaded C code

The following problems all ask you to step through C code according to a specific schedule of threads. To understand how the scheduler switches between threads, you must understand the following model.

Assume a **uniprocessor**. Imagine you have two threads, T,and S. The scheduler runs T and S such that each statement in the C-language language code (or **line** of code as written in our examples) is atomic. We tell you which thread was scheduled by showing you either a "T" or a "S" to designate that **one line** of C-code was scheduled for the corresponding thread; for example, TTTSS means that 3 lines were run from thread T followed by 2 lines from thread S.

Assume each **test** for a `while()` loop or an `if()` statement corresponds to one line of C-code.

Assume function calls that do not require synchronization such as qadd(), qremove(), qempty(), and malloc() require one scheduling interval.

Function calls that may need to wait for another thread to do something (e.g., mutex_lock() and cond_wait()) are treated specially….

For mutex_lock()), assume that the function call to mutex_lock() 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., you may see a long instruction stream TTTTTTT that causes no progress for this thread). Once the lock is available, the next scheduling of the acquiring thread causes that thread to obtain the lock (e.g., after thread S releases the lock, the next scheduling of T will complete mutex_lock(); note that T does need to be scheduled one time with the lock released for mutex_lock() to complete).

The rules for cond_wait() are similar. When one thread calls cond_wait(), if the work has not yet been done to complete cond_wait(), then no matter how long the scheduler runs this thread (e.g., TTTTT), this thread will be waiting in cond_wait(). After another thread runs and does the work necessary for the cond_wait() to complete, then the next scheduling of thread T will cause the cond_wait() line to complete; again, note that T does need to scheduled one time with the work completed for cond_wait() to complete).

Part 6a. Implementation of thread_join() and thread_exit() with CVs [24 points]
Imagine there is a parent thread, P, about to call thread_join() and a child thread, C, about to call thread_exit(). Assume the done variable has been initialized to 0. 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
}
```

59) After the instruction stream "P" (i.e., after the scheduler runs one line from the parent), which line of the parent's will be run when it is scheduled again?

a) p1 *(Hint to get you started: this line if the lock is already acquired and P must wait here)*
b) p2 *(Hint: here if the lock is not acquired and P continues to next line)*
c) p3 *(Hint: no idea how this could happen)*
d) p4 *(Hint: no idea how this could happen)*
e) Code beyond p4 *(Hint: no idea how this could happen)*

60) Assume the scheduler continues on with "C" (the full instruction stream is PC).
Which line will the child run when it is scheduled again?

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

61) Assume the scheduler continues on with PPPP (the full instruction stream is PCPPP). Which line will the parent run when it is scheduled again?

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

62) Assume the scheduler continues on CCC (the full instruction stream is PCPPPCCC). Which line will the child run when it is scheduled again?

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

63) Assume the scheduler continues on with PP (the full instruction stream is PCPPPCCCPP). Which line will the parent run when it is scheduled again?

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

64) Assume the scheduler continues on with CC (the full instruction stream is PCPPPCCCPPCC). Which line will the child run when it is scheduled again?

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

65) Assume the scheduler continues on with P (the full instruction stream is PCPPPCCCPPCCP). Which line will the parent run when it is scheduled again?

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

66) If the above code has a problem, how could it be fixed? Indicate one.

a) Code does not have a problem
b) Change how the done variable is initialized
c) Remove the mutex_lock() and mutex_unlock() calls
d) Change the while() loop to an if() statement
e) None of the above

Part 6b. Inserting into a linked list [15 points]

Assume you have the following code for inserting keys into a shared linked list (this is identical to code shown in class):

```
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;
Typedef struct __list_t {
    node_t *head;
} list_t;
Void List_Insert(list_t *L,  int key) {
    node_t *new = malloc(sizeof(node_t));
    new->key = key;
    new->next = L->head;
    L->head = new;
}
```

Assume a list `L` originally contains nodes with keys `3, 4, and 5`. Assume thread T calls `List_Insert(L, 2)` and thread S calls `List_Insert(L, 6)`. Assume malloc() does not fail. Given the following schedules of C-statements in the list_insert() routines for threads T and S, what will be the final contents of the List?

67) Schedule: TTTTSSSS
   a) 2, 6, 3, 4, 5
   b) 6, 2, 3, 4, 5
   c) 2, 3, 4, 5
   d) 6, 3, 4, 5
   e) Unknown results or None of the above

68) Schedule: SSSSTTTT
   a) 2, 6, 3, 4, 5
   b) 6, 2, 3, 4, 5
   c) 2, 3, 4, 5
   d) 6, 3, 4, 5
   e) Unknown results or None of the above

69) Schedule: SSTTTTSS
   a) 2, 6, 3, 4, 5
   b) 6, 2, 3, 4, 5
   c) 2, 3, 4, 5
   d) 6, 3, 4, 5
   e) Unknown results or None of the above

70) Schedule: SSSTTTTS
   a) 2, 6, 3, 4, 5
   b) 6, 2, 3, 4, 5
   c) 2, 3, 4, 5
   d) 6, 3, 4, 5
   e) Unknown results or None of the above

71) If the above code has a race condition, how could it be fixed? **Indicate all that apply**.
   a) Code does not have a race condition
   b) Lock
   c) Condition variable
   d) Semaphore
   e) Atomic compare and swap instruction

Part 6c. Lock implementation with blocked threads [21 points]

Assume you have the following code for acquiring and releasing a lock (this is identical to code shown in class):

```
Typedef struct {
      bool lock = false;
      bool guard = false;
      queue_t q;  // assume managed FIFO
} LockT;

void acquire(LockT *l) {
      while (TAS(&l->guard, true));                  // a1
      if (l->lock) {                                 // a2
                  qadd(l->q, tid);                   // a3
                  setpark();                         // a4
                  l->guard = false;                  // a5
                  park(); }                          // a6
      else {      l->lock = true;                    // a7
                  l->guard = false;}                 // a8
}

void release(LockT *l) {
      while (TAS(&l->guard, true));                  // r1
      if (qempty(l->q))                              // r2
            l->lock=false;                           // r3
      else unpark(qremove(l->q));                    // r4
      l->guard = false;                              // r5
}
```

Assume the initial state is that the lock is not held. Assume two threads T and S each call acquire() on the same shared lock.

72) After the instruction stream "T" (i.e., after the scheduler runs one line of acquire() for thread T), which line of acquire will be executed for T when T is scheduled again?

a) a1
b) a2
c) a3
d) a4
e) None of the above

73) Assume the scheduler continues with TT (the full instruction stream is TTT), which line will T execute when it is scheduled again?

a) A4
b) A7
c) A8
d) A line after A8
e) None of the above

74) Assume the scheduler continues with SSS (the full instruction stream is TTTSSS). Which line will thread S execute when it is scheduled again?

a) A1
b) A4
c) A5
d) A8
e) None of the above

75) Assume the scheduler continues on with TTT (the full instruction stream is TTTSSSTTT). Which line will thread T execute when it is scheduled again?

a) A4
b) A7
c) A8
d) A line after A8
e) None of the above

76) Assume the scheduler continues on with SSS (the full instruction stream is TTTSSSTTTSSS). Which line will thread S execute when it is scheduled again?

a) A1
b) A4
c) A5
d) A8
e) None of the above

77) Assume the thread T completes its critical section and then calls release(). If the scheduler runs TTT (the full instruction stream is TTTSSSTTTSSSTTT), which line will thread T execute when it is scheduled again?

a) R1
b) R2
c) R3
d) R4
e) None of the above

78) What is the purpose of the `l->guard` variable? Pick the one best answer.

a) To indicate whether or not the lock l is currently held by a caller of acquire()
b) To indicate whether a thread is currently on the queue waiting to acquire the lock
c) To ensure there is not a race condition between testing and setting of l->lock
d) To ensure there is not a race condition between setpark() and park()
e) To ensure there is not a race condition between park() and unpark()

Part 6d. Producer/consumer implementation with locks and semaphores [20 points]

Assume you have the following code for accessing a shared-buffer that contains `max` elements (for some very large value of max). Assume multiple producer and multiple consumer threads access these routines concurrently. Assume the initial state is that the mutex is not held and that all buffers are empty. Assume the semaphore `empty` is initialized to 0 and `fill` is initialized to `max`.

```
void *producer(void *arg) {
            Mutex_lock(&m);                          // p1
            if (numfull == max)                      // p2
                  sema_wait(&empty);                 // p3
            do_fill(i);                              // p4
            sema_post(&fill);                        // p5
            Mutex_unlock(&m);                        // p6
}
void *consumer(void *arg) {
            Mutex_lock(&m);                          // c1
            if (numfull == 0)                        // c2
                  sema_wait(&fill);                  // c3
            int tmp = do_get();                      // c4
            sema_post(&empty);                       // c5
            Mutex_unlock(&m);                        // c6
 }
```

79) After the instruction stream "PPPPP" (i.e., after the scheduler runs 5 lines producer() for a producer thread P), which line of acquire will be executed for P when P is scheduled again?

a) P1
b) P3
c) P5
d) Some line after P6
e) None of the above

80) Assume the scheduler continues on with CCCCC (i.e., the scheduler runs 5 lines of consumer() for a consumer thread C and the full instruction stream is PPPPPCCCCC). Which line will thread C execute when it is scheduled again?

a) c1
b) c3
c) c5
d) Some line after c6
e) None of the above

81) Assume the scheduler starts another consumer with OOO (i.e., the full instruction stream is PPPPPCCCCCOOO). Which line will thread O execute when it is scheduled again?

a) c1
b) c3
c) c4
d) Some line after c6
e) None of the above

82) Assume the scheduler starts another producer with RRR (i.e., the full instruction stream is PPPPPCCCCCOOORRR). Which line will thread R execute when it is scheduled again?

a) p1
b) p3
c) p5
d) Some line after p6
e) None of the above

83) If a problem exists in the above code, what would be the easiest solution to fix it?

a) There is no problem with this code
b) Remove the calls to mutex_lock() and mutex_unlock()
c) Correct how the two semaphores were initialized
d) Change the semaphores to condition variables
e) None of the above