CS 537: Introduction to Operating Systems
Spring 2024: Midterm Exam #2

This exam is closed book, closed notes.

No calculators may be used. All cell phones must be turned off and put away.

You have 1 hour 30 minutes to complete this exam.

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

- CANVAS LAST NAME - fill in your last (family) name starting at the leftmost column
- CANVAS FIRST NAME - fill in the first five letters of your first (given) name
- IDENTIFICATION NUMBER - This is your UW Student WisCard ID number
- ABC of SPECIAL CODES - Write your lecture number as a three digit value:
    - 001 (1:00-2:15 Afternoon Lecture in Social Sciences)
    - 002 (9:30-10:45 Morning Lecture in Science Hall)

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

You may separate the pages of this exam if it helps you.

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

- The OS manages a single uniprocessor (single core)
- All memory is byte addressable
- The terminology lg means $\log_2$
- 2^{10} bytes = 1KB
- 2^{20} bytes = 1MB
- 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).
- Hex numbers are represented with a proceeding “0x”

This exam has 60 questions. Each question has the same number of points.

Good luck!

Part 1. Designate if the statement is True (A) or False (B).

1. Threads that are part of the same process can access the same TLB entries.

2. Threads that are part of the same process share the same code, heap, and stack.

3. If a lock will only be used on a uniprocessor then it should always block instead of spin.

4. On a multiprocessor, a lock implementation should block instead of spin if it is known that the lock will be available before the time required for a context-switch.

5. The hardware atomic exchange instruction requires that interrupts are disabled during that instruction.

6. If a thread holds a lock then other threads that are attempting to acquire the lock cannot be scheduled to run by the OS.

7. It is required to hold the mutex lock when calling pthread_cond_wait().

8. It is required to hold the mutex lock when calling pthread_cond_signal().

9. When condition variable signaling follows Mesa semantics (more common; allows spurious wakeups) it is necessary to recheck the program state when a thread returns from a wait().

10. With producer threads communicating with consumer threads via a finite-sized circular shared buffer, producer threads must wait until there is an empty element in the buffer.

11. With a reader/writer lock either multiple readers can hold the lock or a single writer can hold the lock (or no-one holds the lock).

12. Deadlock cannot occur if one thread never attempts to acquire any locks.

13. Semaphores are often used instead of locks because they cannot cause deadlock.

14. Livelock is when threads acquire some of the locks they need but cannot acquire all of them and so release the locks they hold and try again; yet no thread acquires all the locks they need and so no progress is made.

15. Disabling interrupts as a means of implementing locks does not work on a multiprocessor.

16. With the approximate counter as discussed in lecture and the textbook, the most that the counter can be off by is the user-provided threshold value.

17. Adding a single lock around accesses to a data structure makes the data structure both thread safe and this scales for many concurrent accesses.

18. It is always correct to use broadcast instead of signal when using Mesa semantics.

19. A semaphore used as a lock is called a binary semaphore.

20. A semaphore used to create mutual exclusion for critical sections should be initialized to 0.

21. The call to signal() must only be made if a thread is waiting, otherwise the call to signal() will fail.

22. A lock implementation that performs spin-waiting can provide fairness across threads (i.e. threads receive the lock in the order they requested the lock).

23. To implement a thread_join() operation with a condition variable, the thread_exit() code will call cond_wait().

24. As the amount of code a mutex protects increases, the amount of concurrency in the application increases.

25. Deadlock is impossible if threads acquire all potentially needed locks atomically.

26. Deadlock is impossible if all threads acquire needed locks in the same order.

27. Semaphores can be used exclusively -- they can replace the need for locks and condition variables.

28. One way to prevent deadlock in the dining philosophers problem is to have philosophers put back down a fork they picked up if the second fork they need to eat is not available.

29. Ordering problems in multi-threaded applications can be most easily fixed with locks.

30. Multi-threaded applications can take advantage of parallelism on a multiprocessor system.

Part 2. Forks and Threads -- Select the one best answer, A - E.

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.

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

31. How many times will the message "Hello!\n" be displayed?
A. 2
B. 3
C. 4
D. 6
E. None of the above

32. What is the largest value of "a" that could be displayed by the program?
A. Due to race conditions, "a" may have different values on different runs of the program.
B. 2
C. 3
D. 5
E. None of the above

The next set of questions looks at creating new threads. For the next questions, assume the following code is compiled and run on a modern Linux machine. Assume 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 i;
   for (i = 0; i < 200; i++) {
      balance++;
   }
   printf("Balance is %d\n", balance);
   return NULL;
}

int main(int argc, char *argv[]) {
   pthread_t p1, p2, p3;
   pthread_create(&p1, NULL, mythread, "A");
   pthread_join(p1, NULL);
   pthread_create(&p2, NULL, mythread, "B");
   pthread_join(p2, NULL);
   pthread_create(&p3, NULL, mythread, "C");
   pthread_join(p3, NULL);
   printf("Final Balance is %d\n", balance);
}
```

33. How many total threads are part of this process?
A. 1
B. 2
C. 3
D. 4
E. None of the above

34. What will thread p1 print is the value of balance?
A. Due to race conditions, "balance" may have different values on different runs of the program.
B. 200
C. 400
D. 600
E. None of the above

35. What will the main parent thread print as the final value of balance?
A. Due to race conditions, "balance" may have different values on different runs of the program.
B. 200
C. 400
D. 600
E. None of the above

36. If the final pthread_join() call was not there, what is the smallest value that the main parent thread could print as the final value of balance?
A. Due to race conditions, "balance" may have different values on different runs of the program.
B. 200
C. 400
D. 600
E. None of the above

Part 3. Scheduling Without Locks (assembly code) -- Select the one best answer, A - E.

For the next questions, assume that two threads are running the following code on a uniprocessor (This is the same looping-race-nolocks.s code from homework simulations). 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.

```
# assumes %bx has loop count in it
.main
.top
# critical section
mov 2000, %ax  # get 'value' at address 2000
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 scheduler runs the two threads producing the following order of instructions (the first column shows the address of the executed instruction).

```
       Thread 0                Thread 1
1000 mov 2000, %ax
1001 add $1, %ax
1002 mov %ax, 2000
------ Interrupt ------  ------ Interrupt ------
                         1000 mov 2000, %ax
                         1001 add $1, %ax
                         1002 mov %ax, 2000
                         1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
1003 sub  $1, %bx
1004 test $0, %bx
1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
                         1004 test $0, %bx
                         1005 jgt .top
                         1000 mov 2000, %ax
                         1001 add $1, %ax
------ Interrupt ------  ------ Interrupt ------
1000 mov 2000, %ax
1001 add $1, %ax
1002 mov %ax, 2000                                  37)
1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
                         1002 mov %ax, 2000          38)
                         1003 sub  $1, %bx
                         1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
1004 test $0, %bx
1005 jgt .top
1000 mov 2000, %ax
------ Interrupt ------  ------ Interrupt ------
                         1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
1001 add $1, %ax
1002 mov %ax, 2000                                 39)
1003 sub  $1, %bx
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                         1000 mov 2000, %ax
                         1001 add $1, %ax
                         1002 mov %ax, 2000         40)
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
1006 halt
----- Halt;Switch -----  ----- Halt;Switch -----
------ Interrupt ------  ------ Interrupt ------
                         1003 sub  $1, %bx
                         1004 test $0, %bx
                         1005 jgt .top
                         1006 halt
```

For each of the lines designated above with a question numbered 37-40, determine the contents of the memory address 2000 **AFTER** that assembly instruction executes.

37. What is the contents of memory address 2000?
A) 1
B) 2
C) 3
D) 4
E) None of the above

38. What is the contents of memory address 2000?
A. 1
B. 2
C. 3
D. 4
E. None of the above

39. What is the contents of memory address 2000?
A. 1
B. 2
C. 3
D. 4
E. None of the above

40. What is the contents of memory address 2000?
A. 1
B. 2
C. 3
D. 4
E. None of the above

41. What would be the expected value for the final content of address 2000 if there had been no race conditions between the two threads?
A. 1
B. 2
C. 3
D. 4
E. None of the above

42. Assume looping-race-nolocks.s is run with an unknown scheduler and some random interleaving of instructions occurs across threads 1 and 2 (i.e., not just the interleaving shown above). For an arbitrary, unknown schedule, what contents of the memory address 2000 *are possible* when the two threads are done and the program is completed?
A. Any values >=0 and <=6
B. Any values >=1 and <=6
C. Any values >=3 and <=6
D. Any values >=4 and <=6
E. None of the above

Part 4. Producer / Consumer Problem

The following code is an attempt to solve the producer / consumer problem using condition variables and locks. Remember that `put(i)` increments `numfull` while `get()` decrements it.
The producer code:

```
mutex_lock(&m);           // P1
while (numfull == max)    // P2
  cond_wait(&cond, &m);   // P3
put(i);                   // P4
cond_signal(&cond);       // P5
mutex_unlock(&m);         // P6
```

The consumer code:

```
mutex_lock(&m);           // C1
while (numfull == 0)      // C2
   cond_wait(&cond, &m); // C3
int tmp = get();          // C4
cond_signal(&cond);       // C5
mutex_unlock(&m);         // C6
```

43. There is something wrong with the implementation. What is it?
A. Code uses while loop instead of if statement
B. Lines C2 and P2 are switched
C. Producer can incorrectly wake a consumer
D. Consumer can incorrectly wake a producer
E. None of the above

An attempt is made to fix the broken code from above and the following version is written:
The producer code:

```
mutex_lock(&m);           // P1
while (numfull == max)    // P2
   cond_wait(&bug, &m);   // P3
put(i);                   // P4
cond_signal(&patterns);   // P5
mutex_unlock(&m);         // P6
```

The consumer code:

```
mutex_lock(&m);               // C1
while (numfull == 0)          // C2
   cond_wait(&patterns, &m);  // C3
int tmp = get();              // C4
cond_signal(&bug);            // C5
mutex_unlock(&m);             // C6
```

44. What is wrong with the code now?
A. Too much signaling and waiting
B. Signal signals the wrong CV; wait waits on the wrong one
C. Producer can incorrectly wake a producer
D. Something else is wrong other than A-C
E. None of the above, code is correct

Part 5. Semaphores as Locks and Condition Variables

Here is a working version of the producer/consumer problem again, but using semaphores rather than locks and condition variables. Assume the buffer size is MAX.

```
void *producer(void *arg) { // core of producer
   for (i = 0; i < num; i++) {
      sem_wait(&empty);
      sem_wait(&mutex);
      put(i);
      sem_post(&mutex);
      sem_post(&full);
   }
}

void *consumer(void *arg) { // core of consumer
   while (!done) {
      sem_wait(&full);
      sem_wait(&mutex);
      int tmp = get(i);
      sem_post(&mutex);
      sem_post(&empty);
      // do something with tmp ...
   }
}
```

45. What should the semaphore `full` be initialized to?
A. 0
B. 1
C. -1
D. MAX
E. None of the above

46. What should the semaphore `empty` be initialized to?
A. 0
B. 1
C. -1
D. MAX
E. None of the above

47. What should the semaphore `mutex` be initialized to?
A. 0
B. 1
C. -1
D. MAX
E. None of the above

Part 6. Deadlock

One way to avoid deadlock is to schedule threads carefully. Assume the following characteristics of threads T1, T2, and T3, and that this is the order each thread acquires its locks in.

T1 (at some point) acquires and releases locks L1, L2.
T2 (at some point) acquires and releases locks L1, L3.
T3 (at some point) acquires and releases locks L3, L1, and L4.

For each of the following schedules, which of the listed deadlocks are possible?

48. T1 runs to completion, then T2 to completion, then T3 runs.
A. T1 cannot deadlock, T2 & T3 may deadlock
B. T2 cannot deadlock, T1 & T3 may deadlock
C. T3 cannot deadlock, T1 & T2 may deadlock
D. All threads may deadlock
E. No threads could deadlock

49. T1 and T2 run concurrently and after they complete T3 runs.
A. T1 cannot deadlock, T2 & T3 may deadlock
B. T2 cannot deadlock, T1 & T3 may deadlock
C. T3 cannot deadlock, T1 & T2 may deadlock
D. All threads may deadlock
E. No threads could deadlock

50. T1, T2, and T3 run concurrently.
A. T1 cannot deadlock, T2 & T3 may deadlock
B. T2 cannot deadlock, T1 & T3 may deadlock
C. T3 cannot deadlock, T1 & T2 may deadlock
D. All threads may deadlock
E. No threads could deadlock

51. T1 and T3 run concurrently and after they complete T2 runs.
A. T1 cannot deadlock, T2 & T3 may deadlock
B. T2 cannot deadlock, T1 & T3 may deadlock
C. T3 cannot deadlock, T1 & T2 may deadlock
D. All threads may deadlock
E. No threads could deadlock

52. T2 and T3 run concurrently and after they complete T1 runs.
A. T1 cannot deadlock, T2 & T3 may deadlock
B. T2 cannot deadlock, T1 & T3 may deadlock
C. T3 cannot deadlock, T1 & T2 may deadlock
D. All threads may deadlock
E. No threads could deadlock

53. T1 runs to completion then T2 and T3 run concurrently.
A. T1 cannot deadlock, T2 & T3 may deadlock
B. T2 cannot deadlock, T1 & T3 may deadlock
C. T3 cannot deadlock, T1 & T2 may deadlock
D. All threads may deadlock
E. No threads could deadlock

Part 7. Reader/Writer Locks with Semaphores

The following is a correct implementation of reader/writer locks that uses semaphores and incorporates a technique so writer threads will not starve waiting for reader threads. Assume the code compiles and works 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 (line 1 of Release_writelock)
   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
}
```

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).

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

54. What should `mutex`, `OKtoRead`, and `OKtoWrite` be initialized to?
A. 1, 0, 0 respectively
B. 1, 1, 0 respectively
C. 1, 1, 1 respectively
D. 0, 0, 0 respectively
E. None of the above

55. 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)

56. Where will thread R0 be in the code?
A. AR1
B. AR4
C. AR8
D. Beyond AR8
E. None of the above (including non-deterministic locations)

57. 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, now assume thread W0 calls release_writelock() 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).

58. Where will thread W0 be in the code?
A. RW1
B. RW6
C. RW10
D. Beyond RW11
E. None of the above (including non-deterministic locations)

59. Where will thread R0 be in the code?
A. AR1
B. AR4
C. AR8
D. Beyond AR8
E. None of the above (including non-deterministic locations)

60. Continuing the same execution stream, assume another write thread W1 begins and calls acquire_writelock() and executes as far along as it can before it must block. Where will thread W1 be in the code?
A. AW1
B. AW4
C. AW7
D. Beyond AW7
E. None of the above (including non-deterministic locations)