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

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.
True, since they share the same address space, they have the same vpn->pfn translations and the same TLB entries are valid.
2. Threads that are part of the same process share the same code, heap, and stack.
False, they share the same code and heap but they have separate stacks.
3. If a lock will only be used on a uniprocessor then it should always block instead of spin.
True, on a uniprocessor if a thread can't acquire a lock then it would be best that the thread holding the lock be scheduled so it can more quickly release the lock.
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.
False, if the thread blocks, it will waste the time of a context-switch; if the thread had just used spin-waiting, it would have wasted less time than a context-switch.
5. The hardware atomic exchange instruction requires that interrupts are disabled during that instruction.
False, this instruction has nothing to do with disabling interrupts, it is an atomic 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.
False, the OS can still perform a context switch. Locks simply prevent other threads from executing their critical sections if they do not hold the lock.
7. It is required to hold the mutex lock when calling pthread_cond_wait().
True, the lock must be held when calling wait(). The lock will be released before placing the calling thread on the CV queue.
8. It is required to hold the mutex lock when calling pthread_cond_signal().
False, a thread can call signal() without holding the lock, but it is advisable to hold it otherwise you may have lost wakeups. Take a look at this stack overflow example
https://stackoverflow.com/questions/4544234/calling-pthread-cond-signal-without-locking-mutex
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().
True, being signalled is just a hint that the state of the program may be such that the thread can run, but it is no guarantee.
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.
True, producer threads need to wait until their is a free location to store the produced value 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).
True, that is how reader/writer locks work.
12. Deadlock cannot occur if one thread never attempts to acquire any locks.
False, that one thread may never try to acquire a lock and so may run to completion, but other threads may contend for locks and reach deadlock between those threads.

13. Semaphores are often used instead of locks because they cannot cause deadlock.
False, semaphores have the same properties as locks for mutual exclusion.
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.
True, livelock is similar to deadlock in that all threads make no progress. The difference is that no threads remain blocked on a resource. Instead the threads are cycling in a way that they are still blocking eachother from getting the resources they need to make progress.
15. Disabling interrupts as a means of implementing locks does not work on a multiprocessor.
True, since multiple threads could be executing at the same time on different processors so multiple threads could enter their critical section concurrently. Disabling interrupts only works as a locking mechanism on a uniprocessor.
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.
True or False accepted as correct, Threshold is the maximum value that any one thread could be before copying to the global count. The global count could be off by num_threads*threshold. If there were a single local count, which would occur on a uniprocessor, then the maximum value could be threshold.
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.
False, it does make the data structure thread-safe but it may not scale well as only one thread at a time may interact with the data structure.
18. It is always correct to use broadcast instead of signal when using Mesa semantics.
True, since the waking thread will re-check the program state, waking additional threads will only slow performance but will still operate correctly.
19. A semaphore used as a lock is called a binary semaphore.
True, that is the meaning of a binary semaphore, only one thread will have simultaneous access to the critical section of code.
20. A semaphore used to create mutual exclusion for critical sections should be initialized to 0.
False, it should be initialized to 1 and sem_wait() before a critical section will drop the value to 0 (letting one thread through) and sem_post() after the critical section will raise the value and waken one thread (if there is one) to allow it to enter the critical section.
21. The call to signal() must only be made if a thread is waiting, otherwise the call to signal() will fail.
False, signal() will not fail if no threads are waiting, but no threads will be awoken since none are waiting.
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).
True, think about the ticket lock discussed in lecture.
23. To implement a thread_join() operation with a condition variable, the thread_exit() code will call cond_wait().
False, thread_exit() calls cond_signal().
24. As the amount of code a mutex protects increases, the amount of concurrency in the application increases.
False, with more code within a mutex, concurrency decreases.
25. Deadlock is impossible if threads acquire all potentially needed locks atomically.
True, this breaks the hold-and-wait condition which is necessary for deadlock.
26. Deadlock is impossible if all threads acquire needed locks in the same order.
True, this breaks the circular wait condition which is necessary for deadlock.

27. Semaphores can be used exclusively -- they can replace the need for locks and condition variables.
True, semaphores can be used to create the equivalent of 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.
True, This breaks the no-preemption condition which is necessary for deadlock.
29. Ordering problems in multi-threaded applications can be most easily fixed with locks.
False, condition variables or semaphores are usually used to fix ordering problems.
30. Multi-threaded applications can take advantage of parallelism on a multiprocessor system.
True, that is one of the major advantages of theading. The second is allowing a process to continue to make progress even if a thread is blocked waiting on I/O.

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
B (3) -- After the first fork() 2 processes are running. Only the child process calls fork() again creating a third process. All processes reach the print statement.
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
B (2) -- "a" is a local variable. Each process will get its own copy.
Parent process: a=0; a++ (after first fork()); a++(in else); -> a=2
Child process: a=0 (after first fork()); a++ (after second fork()); a++; -> a=2
Grandchild: a=1 (upon creation after second fork()); a++; -> a=2

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
D (4), the main thread and threads p1, p2, and p3.
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
B (200), Note that none of the created threads run concurrently with one another.
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
D (600), the balance variable is shared across threads -- each child thread increments balance by 200.

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
C (400), the main thread may be scheduled before p3 has run at all which would result in 400 being printed.

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
C (3)
38. What is the contents of memory address 2000?
A. 1
B. 2
C. 3
D. 4
E. None of the above
C (3)
39. What is the contents of memory address 2000?
A. 1
B. 2
C. 3
D. 4
E. None of the above
D (4)
40. What is the contents of memory address 2000?
A. 1
B. 2
C. 3
D. 4
E. None of the above
E (5)
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
E (6), as both threads increment the addr 3 times starting from 0.

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

E, Race conditions don't cause memory address 2000 to hold any random garbage. The race conditions that can happen in this program will not lead to any values greater than 6, since there are only 6 increments from 0. However, some of the increments across the two threads can be missed by the other.

The worst case actually occurs when both threads copy the initial value of 0 into ax, then one thread does all iterations except for the last, then the second thread does one iteration, moving the value of 1 to memory; at this point, the first thread grabs the value of 1 into ax, the second thread does all of its iterations and moves its value to memory, but then the first thread completes its last iteration (in which it adds one to 1 and moves the value of 2 to memory). Thus, values >=2 and <= 6 are possible (E). Tricky!

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
E (none of the above), The code only has a single condition variable and as such a producer could incorrectly wake a producer or a consumer could incorrectly wake a consumer. That is not one of the options, thus 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
E (nothing is wrong), the code works properly, thus none of the above.

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
A (0), initially the buffer is empty and all consumer threads should block when calling sem_wait(&full).
46. What should the semaphore `empty` be initialized to?
A. 0
B. 1
C. -1
D. MAX
E. None of the above
D (MAX), up to MAX values can be placed into the buffer before it is full, then producers would need to wait for a consumer thread to remove a value.
47. What should the semaphore `mutex` be initialized to?
A. 0
B. 1
C. -1
D. MAX
E. None of the above
B (1), this semaphore should act like a mutex lock, allowing one thread to execute the criticial section of code at a time.

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
E (No threads could deadlock), since no threads run concurrently, nothing 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
E (No threads could deadlock), since only T1 and T2 run concurrently and they do not share two locks on which they could deadlock, they can't deadlock so they will complete and then T3 will complete.
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
D (All threads may deadlock), since T2 and T3 share locks L1 and L3, they may deadlock waiting on eachother and T1 may be deadlocked waiting on T2 if it holds the L1 lock.
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
E (No threads could deadlock), since T1 and T3 do not share two locks, they could not 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
D (All threads may deadlock), T2 and T3 could deadlock on locks L1 and L3 so they may not finish and so T1 will never start.
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
A (T1 completes, T2&T3 may deadlock). Since T2 and T3 share locks L1 and L3 they may 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
A (1, 0, 0) -- The mutex is a binary semaphore so should be initialized to 1. Both the OKtoRead and OKtoWrite should be initialized to 0 to not allow any threads through calls to sem_wait() without first a call to sem_post().
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)
D, W0 acquires the writer lock and continues past the shown code
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)
C, R0 did not call sem_post(OKToRead) so it will be stuck on Sem_wait()
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)
C, R1 did not call sem_post(OKToRead) so it will be stuck on Sem_wait()
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)
D, W0 will finish with release_writelock and continues past shown code. As part of release_writelock() it will call sem_post(OKToRead) multiple times to wake all sleeping reader threads.
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)
D, since W0 called sem_post(OKToRead) multiple times, R0 will finish acquire_readlock and continue past the shown code.
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)
C, W1 will also not call sem_post(OKToWrite) and so will also be stuck at sem_wait()