CS 537: Introduction to Operating Systems
Fall 2023: Midterm Exam #2 SOLUTION

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

1. Threads that are part of the same process share the same stack.
False, each thread has its own stack pointer and own stack.
2. Threads that are part of the same process share the same page table.
True, threads share the same address space
3. A context switch between threads of the same process requires flushing the TLB or tracking the ASID.
False, since the threads share the same address space, they share the same VPN->PPN mappings
4. One advantage of multi-threaded programming is parallelism, allowing the program to take advantage of multiple CPUs.
True, threads can run concurrently on multiple CPUs.
5. Another advantage of multi-threaded programming is with IO-bound workloads since a thread can run the IO-bound instructions, not blocking the other threads.
True, one thread blocking does not cause other threads to block.
6. Locks prevent the OS scheduler from performing a context switch during a critical section.
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. A lock implementation should always spin instead of block if it is known that the lock will always be used only on a uniprocessor.
False, on a uniprocessor a spinning process cannot acquire the lock since the other thread must run to release the lock. This is wasted CPU time.
8. A lock implementation on a multiprocessor should always spin instead of blocking if it is known that the lock will be available before the time of a context-switch.
True, then less time will be wasted then performing a context switch.
9. A spin-wait type lock can provide fairness across the threads (i.e. threads receive the lock in the order they requested it).
True, this is demonstrated in the ticket-lock implementation.
10. When performing thread traces, you can assume locks will be acquired in the order they are requested.
False, There is no guarantee in the general locking theory about the order that locks are acquired.
11. Failing to surround a critical section of code with a lock may result in a deadlock situation.
True, if the critical section is the obtaining of other locks and threads try to obtain those other locks in different orders. Recall the prevention lock that surrounds the obtaining of lock L1 and L2 from lecture.
12. A condition variable contains a queue of waiting threads.
True, calling wait() places a thread on the queue and calling signal() wakes one of the threads.
13. While not required, it is best practice to hold the mutex lock when calling pthread_cond_wait().
False, the lock must be held when calling wait(). The lock will be released before placing the calling thread on the CV queue.
14. While not required, it is best practice to hold the mutex lock when calling pthread_cond_signal().
True, a thread can call signal() without holding the lock, but it is advisable to hold it because sometimes
15. The performance of broadcasting to a condition variable decreases as the number of waiting threads increases.
True, with more waiting threads, more will be awoken and need scheduling, all costing a context switch.

16. One can safely modify the state associated with a condition variable without holding the mutex lock.
False, a race condition could occur between a thread modifying the state and a thread reading the state, causing a thread to sleep forever.
17. When condition variable signaling follows Mesa semantics it is not necessary to recheck the program state when a thread returns from a wait().
False, since the signaled thread is not guaranteed to run immediately it is necessary to recheck the program state since another thread may run between the signaling and the waking and change the program state.
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. When a thread returns from a cond_wait() it is safe to assume that it holds the corresponding mutex lock.
True, the lock is released when the thread goes to sleep and it reacquires the lock before returning from the wait call.
20. In the attempted solution to the producer/consumer problem using a single condition variable, a consumer thread may signal another consumer thread to awake which would cause corruption of the data in the buffer.
False, using a single condition variable may cause a consumer thread to awaken another consumer thread, but it may cause a deadlock situation not a data corruption situation.
21. In the solution to the producer/consumer problem with a finite sized circular shared buffer, consuming threads must wait until there is an empty element of the buffer.
False, consumers must wait until there is a full element.
22. Semaphores can be used instead of mutex locks and instead of condition variables.
True, both other primitives can be replaced by semaphores that have the proper starting value.
23. A semaphore used as a condition variable is called a binary semaphore.
False, a binary semaphore is one used like a lock.
24. To implement the thread_join operation with a semaphore, the semaphore is initialized to the value of 0 and the thread_exit() code calls sem_wait().
False, thread_join() calls sem_wait() and thread_exit() calls sem_post().
25. Deadlock can be avoided by using semaphores instead of locks for mutual exclusion.
False, semaphores can be used in place of locks but they also suffer from deadlock.
26. A semaphore used to create mutual exclusion is initialized to 1.
True, if initialized to 1 and sem_wait() before a critical section and sem_post() after the critical section will ensure one thread at a time may be in the critical section.
27. In the approximate counter algorithm, the threshold parameter that controls the transfer from local to global counter controls the tradeoff between accuracy and performance.
True, the higher the threshold, the update to the global variable will occur less often (less accurate) but will also require the global lock less frequently (higher performance).
28. Increasing the number of locks that are used to access a thread-safe data structure (like the hashtable compared to the linked-list example from the book) is guaranteed to increase concurrency and performance.
False, the acquiring and releasing locks may slow down performance.
29. With reader-writer locks as implemented in the textbook, a write thread must wait until all reader threads have released their lock. With many busy reader threads the writer thread may starve.
True, one thread that makes no progress is an example of starvation.
30. Livelock is when a thread holds a resource but can't acquire the remaining needed resources and so must release the resource it holds.
False, livelock is when threads cycle continuously trying to acquire their needed resources and releasing them if they can't all be acquired and none of the threads make progress.

Part 2. Forks and Threads

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;

int main() {
   a++;
   fork();
   a++;
   fork();
   a++;
   if (fork() == 0) {
      printf("Hello!\n");
   } else {
      printf("Goodbye!\n");
   }
   a++;
   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
C, You can type this in and run it to check. The first call to fork() results in two processes; each of those then calls fork() which results in processes; the next call to fork() results in eight processes, but only the children processes (of which there are four) will print "Hello".
32. What will be the final value of "a" as displayed by the final line of the program?
A. The value of "a" may be printed multiple times in different orders with different values.
B. 4
C. 8
D. 13
E. None of the above
B, Each process has its own copy of the variable, with no sharing across processes. Each process will increment a 4 times, resulting in an identical value of 4 for all processes.

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 result = 0;
   result = result + 200;
   balance = balance + 200;
   printf("Result is %d\n", result);
   printf("Balance is %d\n", balance);
   return NULL;
}
```

```
int main() {
   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);
}
```

33. How many total threads are part of this process?
A. 1
B. 2
C. 3
D. 4
E. None of the above
C, Main thread plus the two created threads.
34. 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
C, 200 since "result" is a local variable on the stack, each thread has its own copy so there are no race conditions.
35. 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
A, balance is a shared variable between the threads and since there is no lock controlling access so race conditions may cause different values to be stored in the variable.
36. 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
A, balance is a shared variable between the threads and since there is no lock controlling access so race conditions may cause different values to be stored in the variable.

Part 3. Scheduling Without Locks (assembly code)

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) on the next page.

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

37. A, contents of addr = 1
38. A, contents of addr = 1
39. B, contents of addr = 2
40. B, contents of addr = 2

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

6 so E, 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 poss*ible 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. Non-Blocking Data Structures

Your project partner has written the following correct implementation of `insert(int val)` for a linked list using traditional locks for mutual exclusion:

```
typdef struct {
   int val;
   node_t *next;
} node_t;

node_t *head;
```

```
void insert(int val) {
   node_t *n = malloc(sizeof(*n));
   n->val = val;
   lock(&m);
   n->next = head;
   head = n;
   unlock(&m);
}
```

You decide you would like to replace the locks with calls to the atomic hardware instruction `CompareAndSwap(int *addr, int expect, int new)` which returns 0 on failure and 1 on successful swap. Recall that it changes the value of *addr to new if *addr = expect. You know that insert() modified to use CompareAndSwap() looks something like the following:

```
node_t *head; //assume points to existing list
void insert(int val) {
   node_t *n = malloc(sizeof(*n));
   n->val = val;
   do {
      n->next = head;
   } while (???);
}
```

Which of the following are correct replacements for ??? in the code above? Select (A) True, or (B) False.

48 True, rest False, Intuition: need a solution that ensures n->next still equals head (which means a different thread didn't race in and change the value of head), while atomically updating head to point to n.

First 4 solutions all set n->next equal to something, whereas we want to set head to something. Last two set head equal to n->next, whereas we need to set it to n.

Finally, need !CompareAndSwap() so the while loop continues while we failed.

43. CompareAndSwap(&n->next, head, n)

44. !CompareAndSwap(&n->next, head, n)

45. CompareAndSwap(&n->next, n, head)

46. !CompareAndSwap(&n->next, n, head)

47. CompareAndSwap(&head, n->next, n)

48. !CompareAndSwap(&head, n->next, n)

49. CompareAndSwap(&head, n, n->next)

50. !CompareAndSwap(&head, n, n->next)

Part 5. Spin-Locking with Atomic Hardware Instructions

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

```
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
}

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

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

A. 0
B. 1
C. 2
D. 0 or 2
E. 0 or 1

B, Since release() sets lock->flag to 1, this is how the code shows the lock is not currently held. The lock should be initialized so it is not held, which means 1.

52. To what value or values could lock->flag be compared to acquire() (shown as **xyz** above) to obtain a correct implementation?

A. 0
B. 1
C. 2
D. 0 or 2
E. 0 or 1

D, Any integer but 1 so D. to indicate that the lock is acquired, we can use any value other than that showing the lock is released (1).

Part 6. Reusable Barrier Lock

Imagine a problem where many threads are executing the same piece of sequential code and all the threads need to complete the section of code before moving on to the next section of code. This rendezvousing needs to occur multiple times as the code is executed. So the sequential code looks something like this:

P1
barrier()
P2
barrier()
P3

...

All threads must complete P1 before any thread continues on to P2. All threads must then complete P2 before any thread continues on to P3. This process occurs over and over as the sequential code executes.

It is safe to assume you know the total number of running threads and that all threads will try and enter the barrier. You have created a partial implementation of a barrier lock that uses condition variables and mutex locks as shown below:

```
typedef struct __barrier_t {
   pthread_mutex_t lock;
   pthread_cond_t cond;
   int at_barrier;
   int total_threads;
} barrier_t;

void init(barrier_t *b, int num_threads) {
   pthread_mutex_init(&b->lock, NULL);
   pthread_cond_init(&b->cond, NULL);
   b->total_threads = num_threads;
   b->at_barrier = 0;
}

void barrier(barrier_t *b) {
   pthread_mutex_lock(&b->lock);
   b->at_barrier++;
   if (b->at_barrier < b->total_threads) {
      pthread_cond_wait(&b->cond, &b->lock);
      //FINISH POINT 1
   }
   //FINISH POINT 2
   pthread_mutex_unlock(&b->lock);
}
```

For each of the following attempted completions of the barrier() function select which is most accurate about the solution.

53. **FINISH POINT 1 contains:**

```
//no additional code
```

FINISH POINT 2 contains:

```
pthread_cond_signal(&b->cond);
b->at_barrier--;
```

A. Correct Implementation
B. Correct implementation but more signaling than necessary
C. Atomicity Violation
D. Order Violation

E. Deadlock

D, imagine Thread T1 and T2. T1 waits at first barrier(), T2 makes it through first barrier() and may continue on through remainder of barrier() calls, not waiting for T1 to rendezvous while T1 may not have left first barrier() yet.

54. **FINISH POINT 1 contains:**

```
return;
```

FINISH POINT 2 contains:

```
pthread_cond_broadcast(&b->cond);
b->at_barrier=0;
```

A. Correct Implementation
B. Correct implementation but more signaling than necessary
C. Atomicity Violation
D. Order Violation
E. Deadlock

E, since awoken threads do not release the mutex lock, all subsequent calls to acquire lock will block causing all subsequent calls to barrier() to deadlock.

55. **FINISH POINT 1 contains:**

```
//no additional code
```

FINISH POINTS 2 contains:

```
pthread_cond_broadcast(&b->cond);
b->at_barrier=0;
```

A. Correct Implementation
B. Correct implementation but more signaling than necessary
C. Atomicity Violation
D. Order Violation
E. Deadlock

D, imagine Thread T1 and T2. T1 waits at first barrier(), T2 makes it through first barrier() and may continue on to second barrier() where it will wait(). when T1 returns from wait() in first barrier T2 will be signaled (in second barrier) and exit second barrier before T1 has reached second barrier.

56. **FINISH POINT 1 contains:**

```
pthread_mutex_unlock(&b->lock);
return;
```

FINISH POINT 2 contains:

```
pthread_cond_broadcast(&b->cond);
```

A. Correct Implementation
B. Correct implementation but more signaling than necessary
C. Atomicity Violation
D. Order Violation

E. Deadlock

D, since the at_barrier is never reset to 0, all subsequent calls to barrier() will skip the if statement.

57. **FINISH POINT 1 contains:**

```
pthread_mutex_unlock(&b->lock);
return;
```

FINISH POINT 2 contains:

```
pthread_cond_broadcast(&b->cond);
b->at_barrier=0;
```

A. Correct Implementation
B. Correct implementation but more signaling than necessary
C. Atomicity Violation
D. Order Violation
E. Deadlock

A, last thread into barrier() awakens all earlier threads to enter and resets the at_barrier to its original value. The next entrance to barrier() will behave as the first entrance to barrier() since state was reset and all threads are awake and leaving first barrier and unlocking the mutex.

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
}
```

58. What value should the semaphore mutex be initialized to?
A. 0
B. 1
C. The number of reader threads
D. The number of writer threads
E. None of the above
B, One thread should be able to acquire the mutex lock
59. What value should the semaphore OKToRead be initialized to?
A. 0
B. 1
C. The number of reader threads
D. The number of writer threads
E. None of the above
A, a thread needs to call sem_post() to let a reader thread continue past the sem_wait().
60. What value should the semaphore OKToWrite be initialized to?
A. 0
B. 1
C. The number of reader threads
D. The number of writer threads
E. None of the above
A, a thread needs to call sem_post() to let a writer thread continue past the sem_wait()
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();
```

61. 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)
D, R0 acquires the reader lock and continues past the shown code
62. 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)
C, W0 did not call sem_post(OKToWrite) so it will be stuck on Sem_wait()
63. 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 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).

64. 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)
D, R0 will finish with release_readlock and continues past shown code. As part of release_readlock() it will call sem_post(OKToWrite).
65. 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, since R0 called sem_post(OKToWrite), W0 will finish acquire_writelock and continue past the shown code.
66. 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, nothing has changed for R1, it is still waiting to acquire the readerlock

67. 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)
C, R2 will also not call sem_post(OKToRead) and so will also be stuck at sem_wait()
68. 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, R2 will continue to wait
B. Thread R1 or R2 will acquire the readlock, but it is unknown which (the other waits)
C. First R1 will acquire the readlock, and then R2 will acquire the readlock, while R1 still also has the readlock
D. Both threads R1 and R2 will acquire the readlock concurrently, but the order the threads return from acquire_readlock() is unknown
E. None of the above (including non-deterministic locations)
D, when W0 calls release_writelock() it will call sem_post(OKToRead) two times, therefore both R1 and R2 will be able to continue with the readlock. It is unknown the order they will be awoken.
You have reached the end of the exam!