CS 537: Introduction to Operating Systems
Fall 2016: Midterm Exam #2
November 9, 2016 - SOLUTIONS

This exam is closed book, closed notes.

All cell phones must be turned off and put away.

No calculators may be used.

You have two hours to complete this exam.

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

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

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

- The OS manages a single uniprocessor
- All memory is byte addressable
- The terminology lg means $\log_2$
- 2^{10} bytes = 1KB
- 2^{20} bytes = 1MB
- Page table entries require 4 bytes
- Data is allocated with optimal alignment, starting at the beginning of a page
- Assume leading zeros can be removed from numbers (e.g., 0x06 == 0x6).

This exam has multiple versions. To make sure you are graded with the correct answer key, you must identify this exam version with a Special Code in Column A on your accu-tron sheet. Be sure to fill in the corresponding bubble as well. Your special code is

There are 107 questions on this exam.

Good luck!

Part 1: Straight-forward True/False about Virtualization [2 points each]
Designate if each statement is True (a) or False (b).

1) Two processes reading from the same physical address will access the same contents.

   True – they must have a mapping so share a page so potentially different virtual addresses in each of their address spaces point to the same physical address.

2) Stacks are used for procedure call frames, which include local variables and parameters.

   True.

3) A SJF scheduler may preempt a previously running longer job.

   False – Shortest Job First is non-preemptive – it lets a running job run to completion no matter what type of new jobs arrive (unlike STCF which is preemptive).

4) If all jobs have identical run lengths, a RR scheduler (with a time-slice much shorter than the jobs' run lengths) provides better average turnaround time than FIFO.

   False – with RR, identical jobs will all finish at nearly the same time (at the very end of the workload time), which has very poor average turnaround time.

5) The longer the time slice, the more a RR scheduler gives similar results to a FIFO scheduler.

   True – In the extreme, when the time slice is >= the length of the job, RR degenerates to FIFO (or FCFS).

6) The OS provides the illusion to each thread that it has its own address space.

   False – Each process has its own address space, but threads in the same address space share that address space (e.g., they use the same code and heap).

7) The OS may manipulate the contents of an MMU.

   True – the OS changes the contents of the memory management unit on a context switch across processes.

8) With a single-level page table (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.

   True – 1st memory reference is to page table to translate vpn to ppn (no TLB support); 2nd memory reference is to fetch the instruction.

9) Given a constant number of bits in a virtual address, the size of a linear page table decreases with larger pages.

   True – Larger pages → Fewer pages → Fewer entries in page table → Smaller page table

10) If a physical address is 32 bits and each page is 4KB, the top 18 bits exactly designate the physical page number.

   False; 4 KB pages → 12 bits for offset → 20 bits for ppn.

11) A multi-level page table typically reduces the amount of memory needed to store page tables, compared to a linear page table.

True; with multiple levels, the portions of the page table that correspond to invalid entries may not need to be allocated at all.

12) Paging approaches suffer from internal fragmentation, which decreases as the size of a page decreases.

True; internal fragmentation (the amount of wasted space) decreases if smaller units of allocation are used.

13) The size of a virtual page is identical to the size of a physical page.

True.

14) TLB reach is defined as the number of TLB entries multiplied by the size of a page.

True.

15) If the valid bit is clear (equals 0) in a PTE needed for a memory access, the desired page will be swapped in from the backing store.

False; if the present bit is 0, the page resides on the backing store; if the valid bit is 0, this isn't a valid virtual address at all.

16) TLBs are more beneficial with multi-level page tables than with linear (single-level) page tables.

True, because the cost of walking a multi-level page table is higher than walking a single level (i.e., more memory accesses are needed)

17) When the dirty bit is clear (equals 0) in a PTE needed for a memory access, an identical copy of the desired page resides in the backing store.

True; if dirty bit is clear, then the page is clean, which means the page in memory can be discarded on replacement because its contents reside on the backing store (disk).

18) LRU with N+1 pages of memory always performs better than LRU with N pages of memory.

False; LRU performs as well or better with N+1 pages (might perform the same).

19) FIFO with N+1 pages of memory always performs better than FIFO with N pages of memory.

False; FIFO with N+1 pages can even perform worse.

Part 2: Process States [1 point each]
Assume you have a system with three processes (X, Y, and Z) and a single CPU. Process X has the highest priority, process Z has the lowest, and Y is in the middle. Assume a priority-based scheduler (i.e., the scheduler runs the highest priority job, performing preemption as necessary). Processes can be in one of five states: RUNNING, READY, BLOCKED, not yet created, or terminated. Given the following cumulative timeline of process behavior, indicate the state the specified process is in AFTER that step, and all preceding steps, have taken place. **Assume the scheduler has reacted to the specified workload change.**

For all questions in this Part, use the following options for each answer:
a. RUNNING
b. READY
c. BLOCKED
d. Process has not been created yet
e. Not enough information to determine OR None of the above

Step 1: Process X is loaded into memory and begins; it is the only user-level process in the system.
20) Process X is in which state?

a. Running. X is only process so it will be scheduled.

Step 2: Process X calls fork() and creates Process Y.
21) Process X is in which state?
a. Running. X is highest priority process so it is scheduled.

22) Process Y is in which state?
b. Ready. Y could be scheduled, but it is lower priority than X.

Step 3: The running process issues an I/O request to the disk.
23) Process X is in which state?
c. Blocked. X is waiting for I/O to complete.
24) Process Y is in which state?
a. Running. Y is now the only available ready process to schedule.

Step 4: The running process calls fork() and creates process Z.
25) Process X is in which state?
C. Blocked. X is waiting for I/O to complete.

26) Process Y is in which state?
A. Running. Y is higher priority than Z.

27) Process Z is in which state?
B. Ready. Z is lower priority than Y.

Step 5: The previously issued I/O request completes.
28) Process X is in which state?
A. Running. X is ready and at higher priority than others.
29) Process Y is in which state?
B. Ready. Y could be scheduled, but it is lower priority than X.
30) Process Z is in which state?
B. Ready. Z is lower priority than X.

Step 6: The running process completes.

31) Process X is in which state?

E. X is in terminated state.

32) Process Y is in which state?

A. Running. Y is highest priority runnable process.

33) Process Z is in which state?

B. Ready. Z is lower priority than Y.

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

34) The clock frequency of CPUs has been increasing exponentially each year since 2005.

False – clock frequency has not been increasing that dramatically recently, leading to the need for multiple cores to improve performance.

35) Threads that are part of the same process share the same stack.

False – each thread has its own stack (specifically its own stack and frame pointer) although the stacks are placed in the same address space.

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

True – since they share an address space, they have the same vpn->ppn translations and the same TLB entries are valid.

37) With kernel-level threads, multiple threads from the same process can be scheduled on multiple CPUs simultaneously.

True – this is the benefit of kernel-level threads (true thread support from OS); we could not do this with user-level threads.

38) Locks prevent the OS scheduler from performing a context switch during a critical section.

False – the OS scheduler can still perform context switches whenever it wants; there is no coordination between the scheduler and the lock implementation. Locks simply ensure that IF the scheduler schedules a second thread that also wants to enter the same critical section as the first thread, that the second thread cannot acquire the lock until the first thread releases it.

39) Peterson's algorithm uses the atomic fetch-and-add instruction to provide mutual exclusion for two threads.

False – Peterson's algorithm (using a turn variable and an per-thread intention variable) is based entirely on atomic word loads and stores (and is not used in current systems).

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

True – the ticket lock implementation we looked at in lecture is fair and can use spin-waiting.

41) A lock implementation should block instead of spin if it will always be used only on a uniprocessor.

True – on a uniprocessor, if a thread can't acquire a lock, we'd like the thread holding the lock to have a chance to be scheduled (so it can more quickly release the lock).

42) On a multiprocessor, a lock implementation should block instead of spin if it is known that the lock will 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 than the time for a context-switch.

43) Periodically yielding the processor while spin waiting reduces the amount of wasted time to be proportional to the duration of a context switch.

True; instead of spin-waiting for an entire time slice, the thread will just waste the time for the context switch each time it is scheduled and cannot acquire the lock.

44) When a thread returns from a call to cond_wait() it can safely assume that it holds the corresponding mutex.

True; cond_wait() releases the corresponding mutex lock when it is called, but reacquires the lock before it returns (after being signaled).

45) When a thread returns from a call to cond_wait() it can safely assume that the situation it was waiting for is now true.

False; after this thread has been signaled (meaning the situation it was waiting for is true; e.g., a buffer has been produced) but BEFORE it reacquires the lock, a different (related) thread could acquire the lock and change the situation (e.g., consume the one buffer); thus, when the first thread later reacquires the lock, it must check that the situation it was waiting for is actually true.

46) The call cond_signal() releases the corresponding mutex.

False; cond_signal() doesn't do anything with the mutex.

47) With producer/consumer relationships and a finite-sized circular shared buffer, producing threads must wait until there is an empty element of the buffer.

True; producers must have an empty buffer.

48) To implement a 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().

49) The safety property for dining philosophers states that it is not the case that there exists a philosopher who is hungry and his/her neighbors are not eating.

False; this is more like the liveness property, ensuring that as much progress as possible takes place; the safety property ensures that nothing bad can happen (e.g., two neighboring philosophers are both eating).

50) 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 a reader/writer lock.

51) A thread can hold only one lock at a time.

False, a thread can hold/acquire any number of locks simultaneously.

52) Deadlock can be avoided by using semaphores instead of locks for mutual exclusion.

False; if you use a semaphore for mutual exclusion, it has all the same properties as a traditional lock.

53) Deadlock can be avoided if one thread does not acquire any locks. (REMOVED QUESTION)

Any answer accepted because this wasn't clear enough. Meant to ask something like "Deadlock is guaranteed to not exist as long as one of the threads does not acquire any locks." Which would be false…

Part 4: Straight-forward True/False about I/O Devices [2 points each]
Designate if each statement is True (a) or False (b).

54) The peripheral bus used by hard disk drives tends to provide lower bandwidth than the memory bus.

True; busses that are further away from the main CPU tend to have lower bandwidth and the devices that are connected to them tend to be slower (than RAM).

55) A device driver runs on the microcontroller that is part of the external device.

False; the device driver is part of the OS and runs on the main CPU (not the external device).

56) When interacting with a fast device, it can be better to spin wait than to use interrupts.

True; if the device responds very quickly, the response might come back faster than the time required to context-switch to another process and back again.

57) To use DMA, the OS must inform the device of the address for the relevant data in main memory.

True, with direct memory access, the OS lets the device transfer the actual data.

58) The interface that modern hard disk drives expose to the OS is that of tracks on surfaces.

False, the interface is a simple linear array of sectors (or blocks). The OS doesn't know exactly where tracks are allocated on different surfaces.

59) A hard disk drive can have more surfaces than platters.

True; a disk usually has two surfaces on each platter (but could have just 1 surface per platter).

60) A hard disk drive can have more r/w heads than surfaces.

False; doesn't make sense to have a r/w head with no corresponding surface to read/write from.

61) A seek on a modern disk drive could take around 1 second.

False; seeks are on the order of 10 ms or so (while 10ms is slow, it is two orders of magnitude faster than 1 second).

62) With a 7200 RPM disk, the average rotation time for a random read is expected to be around 8.3 ms.

Time to rotate: 1 minute / 7200 rotations → 8.3 ms / rotation. Average rotation distance is ½. So average rotation time is 4.15 ms.

63) On a modern disk, a workload of sequential reads will be about twice as fast as a workload of random writes.

False; sequential accesses are much faster than just twice that of random accesses. For example, sequential performance on the Cheetah disk is about 125 MB/s. We calculated random performance as about 2.5 MB/s.

64) Track skew accounts for the amount a disk rotates while the r/w head seeks for one track.

True; sectors on adjacent tracks are skewed so that sequential accesses can be performed without the disk rotating past the next sector when we move to the next track.

65) On a disk with zones, large files that will be accessed sequentially should be placed on the outer tracks instead of the inner tracks.

True; with zones, disks get better sequential bandwidth on the outer tracks.

66) Shortest-positioning-time-first can be implemented more accurately inside of a disk than within the OS.

True; the disk knows the exact geometry and layout of sectors to tracks and can calculate exactly how much rotation and seek time are required for each access; the OS can only roughly approximate these times.

67) Shortest-positioning-time-first treats I/O requests from different processes fairly.

False; SPTF doesn't know about different processes and doesn't try to do anything fairly.

68) The elevator algorithm takes rotational time into account when scheduling I/O requests.

False; the elevator algorithm optimizes seek costs while avoiding starvation; it doesn't know anything about the rotation time for different requests.

69) An anticipatory scheduler may leave the disk idle even when there are waiting I/O requests.

True.

Part 5. Fork and Thread_Create() [4 points each]

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):

```
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);
}
```

70) Assuming `fork()` never fails, how many times will the message "Hello!\n" be displayed?

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

After first fork(): 2 processes.
Only child calls second fork() creating a third process (call 'grandchild').

71) What will be the **largest** value of "a" 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

Parent process: a = 0; a++; a++; → a=2.
Child process: a=0 (after fork()); a++; a++; → a=2.
Grandchild: a=1 (after fork()); a++; → a =2.

Part 5 continued.

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):

```
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);
}
```

72) Assuming none of the system calls fail, when thread p1 prints "Balance is %d\n", what will p1 say 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

Note that none of the 3 created threads, A, B, and C, run concurrently with one another! The main thread waits for one to finish (using pthread_join()) before it creates the next thread. So, there is concurrency and no race conditions!

73) Assuming none of the system calls fail, when the main parent thread prints "Final Balance is %d\n", what will the parent thread say is the value of `balance`?

a) Due to race conditions, "balance" may have different values on different runs of the program.
b) 200
c) 400
d) 600
e) None of the above

Part 6. Impact of scheduling without locks (assembly code) [2 points each]
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.

For each of the lines designated below with a question numbered 74-79, 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

```
Thread 0                Thread 1                    ax_t2 ax_t1
1000 mov 2000, %ax                                  0     ?
1001 add $1, %ax                                    1     ?
1002 mov %ax, 2000                                  74) Contents of addr 2000? 1
------ Interrupt ------  ------ Interrupt ------
                        1000 mov 2000, %ax          1     1
                        1001 add $1, %ax            1     2
                        1002 mov %ax, 2000          75) Contents of addr 2000? 2
                        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          1     2
                        1001 add $1, %ax            1     3
------ Interrupt ------  ------ Interrupt ------
1000 mov 2000, %ax                                  2     3
1001 add $1, %ax                                    3     3
```

```
1002 mov %ax, 2000                                  76) Contents of addr 2000? 3
1003 sub  $1, %bx
------ Interrupt ------   ------ Interrupt ------
                         1002 mov %ax, 2000         77) Contents of addr 2000? 3
                         1003 sub  $1, %bx
                         1004 test $0, %bx
------ Interrupt ------   ------ Interrupt ------
1004 test $0, %bx
1005 jgt .top
1000 mov 2000, %ax                                  3     3
------ Interrupt ------   ------ Interrupt ------
                         1005 jgt .top
------ Interrupt ------   ------ Interrupt ------
1001 add $1, %ax                                    4     3
1002 mov %ax, 2000                                  78) Contents of addr 2000? 4
1003 sub  $1, %bx
1004 test $0, %bx
------ Interrupt ------   ------ Interrupt ------
                         1000 mov 2000, %ax         4     4
                         1001 add $1, %ax           4     5
                         1002 mov %ax, 2000         79) Contents of addr 2000? 5
------ 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
```

80) Assume `looping-race-nolock.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

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.

What is the maximum number of increments by one thread that could be missed? Mistakenly, credit was given for Option c, incorrectly assuming that the worst case occurs when one thread copies the initial value of 0 into ax, then thread 2 runs to completion, incrementing the value to 3; then, the first thread continues back with the incorrect value of 0, and increments it to 3.

However, 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 6: Wait-Free Algorithms [12 total points]

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

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

node_t *head; // assume points to an existing list
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 `CmpAndSwap(int *addr, int expect, int new)`, which returns 0 on failure and 1 on success. The exact behavior of atomic `CmpAndSwap()` is defined as in class.

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

```
node_t *head; // assume points to an 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?
For questions 81-88, mark the suggested replacement for ??? as Possible (a) or Not Possible (b).

81) `CmpAndSwap(&n->next, head, n)`
82) `!CmpAndSwap(&n->next, head, n)`
83) `CmpAndSwap(&n->next, n, head)`
84) `!CmpAndSwap(&n->next, n, head)`
85) `CmpAndSwap(&head, n->next, n)`
86) `!CmpAndSwap(&head, n->next, n)`
87) `CmpAndSwap(&head, n, n->next)`
88) `!CmpAndSwap(&head, n, n->next)`

Answer 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.

The first 4 solutions all set n->next equal to something, whereas we want to set head equal to something.

The last 2 solutions set head equal to n->next, whereas we need to set it to n.

Finally, we need !CmpAndSwap() so the while loop continues while we failed (and another thread changed the value of head).

Part 7: Lock implementation with blocked threads [21 total points]

Your project partner started to write the code for implementing lock `acquire()` and lock `release()`, but the code isn't quite working yet. Your job is to finish it up. The following code is very similar to the code in lecture, except it has a problem.

```
typedef struct {
   bool lock = false;
   bool guard = false;
   queue_t q;
} LockT;
void acquire(LockT *l) {                    // A0
   while (TAS(&l->guard, true));            // A1
   if (l->lock) {                           // A2
        qadd(l->q, tid);                    // A3
        setpark(); // notify of plan        // A4
        park(); // unless unpark()          // A5
   } else {                                 // A6
        l->lock = true;                     // A7
   }                                        // A8
}
void release(LockT *l) {
   while (TAS(&l->guard, true));
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
   l->guard = false;
}
```

The problem is that `acquire()` does not set `l->guard=false`. Where in the code could the statement `l->guard=false` be correctly placed? Note that the statement may need, or may be able, to be placed in multiple locations. **For questions 89-95, mark each location as Possible (a) or Not Possible (b) to provide a correct implementation for both lock acquire() and release().**

89) Between A0 and A1

b. Not Possible. This would wrongly set guard to false even if another thread currently had the guard lock.

90) Between A1 and A2

b. Not possible. This would make the critical section have nothing in it.

91) Between A2 and A3

b. Not possible. Before thread gets chance to add self to queue, thread releasing lock could mistakenly think queue is empty.

92) Between A3 and A4

b. Not possible. Before we call setpark(), thread calling release() could try to unpark us and we would then lose that unpark signal and be stuck on park() forever.

93) Between A4 and A5

a. Possible. Correct location.

94) Between A6 and A7

b. Not possible. Releases guard before lock is set to true; thus, another thread calling acquire() could see lock hasn't been set yet and both would incorrectly acquire lock.

95) Between A7 and A8

a. Possible. Correct location; critical section includes test of lock and set of lock to true.

Although it was correct, you do not like the structure of two lines of code in `release()`:

```
if (qempty(l->q)) l->lock=false;
```

```
   else unpark(qremove(l->q));
```
and you decide to restructure those two lines to:
```
   if (!qempty(l->q)) unpark(qremove(l->q));
   l->lock=false;
```

96) What additional changes do you now need to make in order for all the code to be correct?
   a) No other changes are needed
   b) Place `l->lock=true` between A3 and A4
   c) Place `l->lock=true` between line A4 and A5
   d) Place `l->lock=true` after line A5
   e) Other changes beyond options b, c, and d are needed to make the code correct

   This is a little tricky. This modification is setting lock to false in the (new) situation where we are handing off the lock to the thread we are unparking. So, you might think that setting lock to true after line A5 would work, but it won't work because there is a race condition.

   We can't guarantee after we unpark the thread (say thread A) that it will be the next thread scheduled; it is possible thread C calls acquire() right at that point, sees that lock is false, acquires the lock, and enters the critical section. The problem is that when thread A is then scheduled, it doesn't recheck the condition (i.e., that lock is false) and it blindly believes it should be able to get the lock and sets lock to true.

Part 8. Scheduling multi-threaded C code [2 points each]
The following problem 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. This is identical to what was presented in previous exams and examples.

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 by the corresponding thread; for example, TTTSS means that 3 lines were run from thread T followed by 2 lines from thread S.

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

Function calls that may have to wait for something to happen (e.g., sem_wait()) are treated specially.

For sem_wait(), assume that the function call and return of sem_wait() requires exactly one scheduling interval if the calling process does not need to wait (i.e., it completes with just "T"). If the semaphore requires some other operation to occur in order for sem_wait() to complete, then assume the call spin-waits until that other operation occurs (e.g., you may see a long instruction stream "TTTTTTT" that causes no progress for this thread); once the other operation occurs, the next scheduling of the waiting thread causes that thread to finish the call to sem_wait() (i.e., it completes then with just "T").

Consider the following code for implementing rw locks; it continues on the next page. It is identical to the code presented in lecture.

```
typedef struct _rwlock_t {
```

```
     sem_t lock;
     sem_t writelock;
     int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
      rw->readers = 0;
      sem_init(&rw->lock, ???);
      sem_init(&rw->writelock, ???);
}
```

```
void rwlock_acquire_readlock(rwlock_t *rw) {
      sem_wait(&rw->lock);                          // AR1
      rw->readers++;                                // AR2
      if (rw->readers == 1)                         // AR3
         sem_wait(&rw->writelock);                  // AR4
      sem_post(&rw->lock);                          // AR5
}

// 1 line of Critical Section C1 after acquire

void rwlock_release_readlock(rwlock_t *rw) {
      sem_wait(&rw->lock);                          // RR1
      rw->readers--;                                // RR2
      if (rw->readers == 0)                         // RR3
         sem_post(&rw->writelock);                  // RR4
      sem_post(&rw->lock);                          // RR5
}

rwlock_acquire_writelock(rwlock_t *rw){sem_wait(&rw->writelock);}// AW1

// 1 line of Critical Section C1 after acquire

rwlock_release_writelock(rwlock_t *rw){sem_post(&rw->writelock);}// RW1
```

Assume `rwlock_init()` has already been called. Assume there are 3 threads, W, R, and S that want to execute the following requests:

W: rwlock_acquire_writelock(rw); 1 line of critical section C1 (not shown); rwlock_release_writelock(rw);
R: rwlock_acquire_readlock(rw); 1 line of critical section C1 (not shown); rwlock_release_readlock(rw);
S: rwlock_acquire_readlock(rw); 1 line of critical section C1 (not shown); rwlock_release_readlock(rw);

For example, after thread R completes `rwlock_acquire_readlock(rw)` (specifically, after it finishes line AR5), it will execute in its critical section, executing one line, C1; it will then call rwlock_release_readlock(rs) (and the next line it will execute will be RR1).

The exact ordering of these operations across W, R, and S will depend upon the scheduler as described below.

97) To begin, how should the semaphore rw->lock be initialized in the statement `sem_init(&rw->lock, ???);`

a) -1
b) 0
c) 1
d) 2
e) None of the above

98) How should rw->writelock be initialized in `sem_init(&rw->writelock, ???);`

a) -1
b) 0
c) 1
d) 2
e) None of the above

99) Now let us consider what happens when the scheduler starts running these three threads. Assume the scheduler runs one line from W; that is, W calls sem_wait() within rwlock_acquire_writelock(). After the instruction stream "W" (i.e., after the scheduler runs one line), which line of W's will be run when W is scheduled again?

a) AW1 (hint: here if the condition for sem_wait(&rw->writelock) was not satisfied)
b) C1 (hint: here if the condition for sem_wait(&rw->writelock) was met)
c) RW1 (hint: don't know how this would happen)
d) Code byond RW1 (hint: don't know how this would happen)
e) Nothing else makes sense for W

100) Assume the scheduler continues on with RRRR (i.e., the scheduler runs 4 lines for one of the reader threads, and the full instruction stream is WRRRR). Which line will R execute when R is scheduled again?

a) AR1
b) AR3
c) AR4
d) AR5
e) None of the above

Runs following code; stuck waiting on sem_wait(writelock);

```
sem_wait(&rw->lock);                    // AR1
rw->readers++;                          // AR2
if (rw->readers == 1)                       // AR3
      sem_wait(&rw->writelock);             // AR4
```

101) Assume the scheduler continues on with SSSS (i.e., the scheduler runs 4 lines for another reader thread, S, and the full instruction stream is WRRRRSSSS). Which line will S execute when S is scheduled again?

a) AR1 – stuck on sem_wait(&rw->lock) held by R
b) AR3
c) AR4
d) AR5
e) None of the above

102) Assume the scheduler continues on with WWW (i.e., the scheduler runs 3 lines for the original write thread, and the full instruction stream is WRRRRSSSSWWW). Which line will W execute when W is scheduled again?

a) AW1
b) C1
c) RW1
d) Code beyond RW1
e) Nothing else makes sense

Completes C1, releases its write lock (which posts to writelock), and continues on beyond RW1

103) Assume the scheduler continues on with SSSS (i.e., the scheduler runs 4 lines for reader thread,

S, and the full instruction stream is WRRRRSSSSWWWSSSS). Which line will S execute when S is scheduled again?

a) AR1 – S is still stuck waiting on sem_wait(&rw->lock) held by R
b) AR3
c) AR4
d) AR5
e) None of the above

104) Assume the scheduler continues on with RRRRR (i.e., the scheduler runs 5 lines for reader thread, R, and the full instruction stream is WRRRRSSSSWWWSSSSRRRRR). Which line will R execute when R is scheduled again?

a) AR1
b) AR3
c) AR4
d) AR5
e) None of the above

Finishes acquiring the read lock:

```
      sem_wait(&rw->writelock);                    // AR4
   sem_post(&rw->lock);                            // AR5

   1 line of critical section

   then releases read lock:
   sem_wait(&rw->lock);                            // RR1
   rw->readers--;                                  // RR2

   Next line will be RR3
```

105) Assume the scheduler continues on with SS (i.e., the scheduler runs 2 lines for reader thread, S, and the full instruction stream is WRRRRSSSSWWWSSSSRRRRRSS). Which line will S execute when S is scheduled again?

a) AR1 – Still stuck on lock held by R
b) AR3
c) AR4
d) AR5
e) None of the above

106) Assume the scheduler continues on with RRR (i.e., the scheduler runs 3 lines for reader thread, R, and the full instruction stream is WRRRRSSSSWWWSSSSRRRRRSSRRR). Which line will R execute when R is scheduled again?

a) RR1
b) RR2
c) RR3
d) RR4
e) None of the above

```
      if (rw->readers == 0)                        // RR3
```

```
    sem_post(&rw->writelock);                    // RR4
sem_post(&rw->lock);                             // RR5

Next will be 1 line of critical section
```

107) Assume the scheduler continues on with SSSS (i.e., the scheduler runs 4 lines for reader thread, S, and the full instruction stream is WRRRRSSSSWWWSSSSRRRRRSSRRRSSSS). Which line will S execute when S is scheduled again?

a) RR1
b) RR2
c) RR3
d) RR4
e) None of the above – AR5

S finally gets to acquire the lock and complete statement AR1. Executes these 4 statements:

```
sem_wait(&rw->lock);                             // AR1
rw->readers++;                                   // AR2
if (rw->readers == 1)                            // AR3
   sem_wait(&rw->writelock);                     // AR4

Then will execute AR5 next time.
```

Congratulations on finishing! See you in lecture tomorrow to talk about file systems!