CS 537: Introduction to Operating Systems
Fall 2015: Midterm Exam #2
SOLUTIONS

This exam is closed book, closed notes. All cell phones must be turned off. No calculators may be used.

You have two hours to complete this exam.

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

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

Please use the following Special Codes on the accu-scan form:

A: Unofficial discussion section 301
B: Unofficial discussion section 302
C: Unofficial discussion section 303
D: Unofficial discussion section 304

There are 83 multiple choice questions on this exam.

For all questions except question 71, pick the **one** best answer.

Good luck!

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

1) **Cooperative multi-tasking** requires hardware support for a timer interrupt.

False: if cooperative, only switch when process enters OS through other means (e.g., system calls)

2) Two processes reading from the **same virtual address** will access the same contents.

False: processes have different address spaces, so different contents

3) The **convoy effect** occurs when longer jobs must wait for shorter jobs.

False: occurs when shorter jobs must wait for longer jobs

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

True.

5) A **RR scheduler** may preempt a previously running job.

True.

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

False; longer time-slices make it more like FIFO since job has more time to run to completion.

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

False; each process, not thread, has its own address space.

8) An **instruction pointer register** is identical to a program counter.

True.

9) With **dynamic relocation**, hardware dynamically translates an address on every memory access.

True.

10) The OS may **manipulate** the contents of an MMU.

True.

11) With pure segmentation (and no other support), fetching and executing an instruction that performs an add of a constant value to a register will involve exactly **two memory references**.

False, just one memory reference; pure segmentation implies physical address can be computed by just determing the segment and adding the virtual offset within this segment to the physical base for this segment (i.e., no memory references needed to calculate physical address); fetching the instruction then takes one memory reference and the add itself does not perform more memory references.

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

False; internal fragmentation increases as the size of the page increases.

13) The **number** of virtual pages is always identical to the number of physical pages.

False; it doesn't have to (though the size of each type of page must be identical)

14) A TLB **caches** translations from virtual page numbers to logical page numbers.

False; caches translations from virtual page numbers to physical (not logical).

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

True.

16) A **TLB miss** is usually slower to handle than a **page miss**.

False; TLB miss just requires memory references to walk the page tables; a page miss requires fetching that page from disk.

17) A single page can be **shared** across two address spaces by having each process use the same page table.

False; to share a page, just share one entry of a page table (using the same page table would share the whole address space).

18) If the **present bit** is clear (equals 0) in a PTE needed for a memory access, the running process is likely to be killed by the OS.

False; present bit = 0 means the page is on disk and not in main memory, so the page just needs to be fetched.

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

True; without a TLB, with multi-level page tables, many memory references are needed to translate each address.

20) When a page fault occurs, it is more expensive to replace a **clean** page than a dirty page.

False; more expensive to replace a dirty page since the OS must write out those page contents that do not match the copy on disk.

Part 2: True/False from Concurrency [3 points each]

Designate if the statement is True (a) or False (b).

21) The **clock frequency** of CPUs has been increasing **exponentially** each year since 1985.

False; clock frequency has not been increasing exponentially in more recent years, requiring multiple cores to achieve the same type of speed-up.

22) Threads that are part of the same process share the same **stack pointer**.

False; each thread has its own stack and stack pointer so it can have own call stack and local

23) Threads that are part of the same process share the same **page table base register.**

True; the threads have the same address space.

24) Threads that are part of the same process share the same **general-purpose registers**.

False; each thread must have own registers (saved and restored when that thread is not executing).

25) **Context-switching** between threads of the same process requires flushing the TLB or tracking an ASID in the TLB.

False; since the threads share the same address space, they share the same VPN->PPN mappings.

26) With **kernel-level threads**, if one thread of a process blocks, all the threads of that process will also be blocked.

False; with kernel-level threads, the OS is aware of the state of each thread and just that thread will be blocked.

27) The **hardware atomic exchange** instruction requires that interrupts are disabled during that instruction.

False; this instruction has nothing to do with disabling interrupts.

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

True; the only question that most people answered incorrectly. Spin-waiting has poor efficiency, but it can still provide fairness. Think about the implementation we described for ticket locks… uses spin-waiting but is still fair.

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

True; must block to relinquish processor to thread currently holding lock so that thread will relinquish lock.

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

True; the amount of wasted time is then less than the amount of time wasted for a context switch.

31) Periodically **yielding** the processor while spin-waiting reduces the amount of wasted time to be proportional to the duration of a time-slice.

False; the wasted time is proportional to the cost of a context-switch.

32) A **semaphore** can be used to provide **mutual exclusion**.

True; initialize the semaphore to 1.

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

True; the thread relinquishes the mutex while in cond_wait() but acquires it again before it returns.

34) A thread calling **cond_signal()** will block until the corresponding mutex is available.

False; cond_signal doesn't need to do anything with the mutex.

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

False; consumers must wait until there is a full element.

36) The performance of **broadcasting** to a condition variable decreases as the number of waiting threads increases.

True; with more waiting threads, more threads are awoken which must be scheduled (all incurring a context-switch).

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

False; thread_exit() calls cond_signal().

38) A goal of **a reader/writer lock** is to ensure that either multiple readers hold the lock or just one writer holds the lock.

True.

39) Building a condition variable on top of a semaphore is **easier** than building a semaphore over condition variables and locks.

False; this is difficult…

40) 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 (imagine one big lock around all the code).

41) **Ordering** problems in multi-threaded applications can be most easily fixed with locks.

False; ordering is fixed with condition variables or semaphores.

42) A **wait-free** algorithm relies on condition variables instead of locks.

False; wait-free algorithms use atomic hardware instructions.

43) Deadlock can be avoided if threads acquire all potentially needed locks **atomically.**

True; this breaks the hold-and-wait condition which is necessary for deadlock.

44) Deadlock can be avoided by having only a **single lock** within a multi-threaded application.

True; this breaks the hold-and-wait condition which is necessary for deadlock.

Part 3. Fork Processes vs. Creating Threads [20 points]

The first set of questions looks at creating new processes with fork(). For the next two questions, assume the following code is compiled and run on a modern Linux machine. Assume any irrelevant details have been omitted and that no routines, such as fork(), ever fail.

```
int a = 0;

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

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

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

Answer: 4 → c. You can type this in and run it if you don't believe. The first call to fork() results in two processes; each of those two processes then calls fork() which results in a total of four processes; the next call to fork() results in eight processes, but only the child process (of which there are four) prints "Hello".

46) What will be the **final** value of "a" as displayed by the final line of the program?

a) The value for "a" may be printed multiple times in different orders with different values.
b) 3
c) 4
d) 8
e) None of the above

Answer: 3 → b. Each process has its own copy of the variable a, with no sharing across processes. Each process will increment 'a'three times, resulting in an identical value of 'a=3' for all 8 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 any irrelevant details have been omitted and that no routines, such as pthread_create() or pthread_join(), ever fail.

```
volatile int balance = 0;

void *mythread(void *arg) {
      int result = 0;
```

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

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

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

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

Answer: 3 → c. Main thread plus two created threads.

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

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

Answer: 200-> c. 'result' is a local variable allocated on the stack; each thread has its own private copy which only it increments, therefore there are no race conditions.

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

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

Mark as answer 84. Answer: a. balance is allocated on the heap and shared between the two threads that are each accessing it without locks; there is a race condition.

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

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

Answer: a. balance is allocated on the heap and shared between the two threads that are each accessing it without locks; there is a race condition.

END of PART 3.

Part 4. Impact of scheduling without locks (assembly code) [21 points]

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

```
# assumes %bx has loop count in it
.main
.top
mov 2000, %ax  # get the value at the address
add $1, %ax    # increment it
mov %ax, 2000  # store it back

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

This code is incrementing a variable (e.g., a shared balance) many times in a loop. Assume that the %bx register begins with the value 3, so that each thread performs the loop 3 times. Assume the code is loaded at address 1000 and that the memory address 2000 originally contains the value 0. Assume that the scheduler runs the two threads producing the following order of instructions (the first column shows the address of the executed instruction). The code continues on the next page.

Answer: We find it useful to track the value of ax, which is separate for each thread, along with addr 2000 which is shared across the two threads. Instructions involving register bx and jumping can be ignored.

```
Thread 0                  Thread 1
1000 mov 2000, %ax                                  Contents of ax = 0
1001 add $1, %ax                                    Contents of ax = 1
------ Interrupt ------  ------ Interrupt ------
                         1000 mov 2000, %ax         Contents of ax = 0
------ Interrupt ------  ------ Interrupt ------
1002 mov %ax, 2000                                  50) Contents of addr = 1
------ Interrupt ------  ------ Interrupt ------
                         1001 add $1, %ax           Contents of ax = 1
                         1002 mov %ax, 2000         51) Contents of addr = 1
                         1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
                         1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                         1005 jgt .top
                         1000 mov 2000, %ax         Contents of ax = 1
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
1000 mov 2000, %ax                                  Contents of ax = 1
1001 add $1, %ax                                    Contents of ax = 2
------ Interrupt ------  ------ Interrupt ------
                         1001 add $1, %ax           Contents of ax = 2
                         1002 mov %ax, 2000         52) Contents of addr = 2
------ Interrupt ------  ------ Interrupt ------
1002 mov %ax, 2000                                  53) Contents of addr = 2
```

```
------ Interrupt ------  ------ Interrupt ------
                         1003 sub  $1, %bx
                         1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
1003 sub  $1, %bx
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                         1005 jgt .top
                         1000 mov 2000, %ax          Contents of ax = 2
                         1001 add $1, %ax            Contents of ax = 3
------ Interrupt ------  ------ Interrupt ------
1005 jgt .top
1000 mov 2000, %ax                                   Contents of ax = 2
------ Interrupt ------  ------ Interrupt ------
                         1002 mov %ax, 2000          54) Contents of addr = 3
                         1003 sub  $1, %bx
                         1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
1001 add $1, %ax                                     Contents of ax = 3
1002 mov %ax, 2000                                   55) Contents of addr = 3
1003 sub  $1, %bx
------ Interrupt ------  ------ Interrupt ------
                         1005 jgt .top
------ Interrupt ------  ------ Interrupt ------
1004 test $0, %bx
------ Interrupt ------  ------ Interrupt ------
                         1006 halt
----- Halt;Switch -----  ----- Halt;Switch -----
1005 jgt .top
1006 halt
```

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

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

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

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

Answer: 6->d. Each of the two threads would add 3 to address 2000 (which started at 0).
End of Part 4.

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

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

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

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

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

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

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

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

Answer: 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.

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

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

Answer: (any integer but 1)->d. To indicate that the lock is acquired, we can use any value other than that showing the lock is released (1).

End of Part 5.

Part 6. Impact of scheduling multi-threaded C code

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

Assume a **uniprocessor**.

The scheduler runs each thread in the system such that each **line** of the given C-language code executes in one scheduler tick, or interval. For example, if there are two threads in the system, T and S, we tell you which thread was scheduled in each tick by showing you either a "T" or a "S" to designate that **one line** of C-code was scheduled for the corresponding thread; for example, TTTSS means that 3 lines of C code were run from thread T followed by 2 lines from thread S.

Some lines of C code require special rules, as follows.

Assume each **test** of a `while()` loop or an `if()` statement requires one scheduler tick. Assume jumping to the correct code does not take an additional tick (e.g., jumping either inside or outside the while loop or back to the while condition does not take an extra tick; jumping to the **then** or the **else** branch of an **if** statement does not take an extra tick).

Assume **function calls** whose internals are not shown and that do not require synchronization, such as qadd(), qremove(), qempty(), and malloc(), require one scheduling tick.

Function calls that may need to **wait for another thread** to do something (e.g., mutex_lock() and cond_wait()) may consume an arbitrary number of scheduling ticks and are treated as follow.

For **mutex_lock()**, assume that the function call to mutex_lock() requires one scheduling interval if the lock is available. If the lock is not available, assume the call spin-waits until the lock is available (e.g., you may see a long instruction stream TTTTTTT that causes no progress for this thread). Once the lock is available, the next scheduling of the acquiring thread causes that thread to obtain the lock (e.g., after a thread S releases the lock, the next scheduling of the waiting thread T will complete mutex_lock(); note that T does need to be scheduled for one tick with the lock released for mutex_lock() to complete).

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

If a thread continues to be scheduled after it reaches the **end** of the shown code, assume it continues running lines that have no interaction with other threads.

Part 6a. Implementation of thread_join() and thread_exit() with CVs [24 points]
Using the scheduling model that has been described, imagine there is a parent thread, P, starting the call to thread_join() and a child thread, C, starting the call to thread_exit(). Assume the done variable has been initialized to 0. Imagine you have the following implementation of thread_join() and thread_exit():

```
void thread_join() {
        Mutex_lock(&m);          // p1
        while (done == 0)        // p2
            Cond_wait(&c, &m);   // p3
        Mutex_unlock(&m);        // p4
}
void thread_exit() {
        Mutex_lock(&m);          // c1
        done = 1;                // c2
        Cond_signal(&c);         // c3
        Mutex_unlock(&m);        // c4
}
```

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

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

Answer: p2 (b). P will have acquired the lock, which means it will have finished mutex_lock().

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

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

Answer: c1 (a). C must wait to acquire the lock since it is currently held by P.

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

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

Answer: p3(c). Since done = 0, P will execute while p2 and p3; it is stuck in p3 until signaled. Note that P relinquishes the lock while in p3.

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

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

Answer: c4 (d). C finishes c1, executes c2, and then c3. The next time C runs, it will execute c4. At this point, the condition variable has been signaled, but the mutex is still locked.

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

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

Answer: p3 (c). P cannot return from cond_wait() until it is able to acquire the lock; since the lock is held by C, P stays in cond_wait().

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

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

Answer: (e). C executes c4 and then code beyond c4.

65) Assume the scheduler continues on with PPP (the full instruction stream is PCPPPCCCPPCCPPP). Which line will the parent execute when it is scheduled again?

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

Answer: (e). P finishes p3, then rechecks p2 (the while loop), and then executes p4. Next time, it will execute code beyond p4.

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

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

Answer: (a) No problems with this code…

End of Part 6a.

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

Assume the following code for inserting keys into a shared linked list (identical to code in class):

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

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

67) Schedule: TTTTSSSS

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

Answer: (b). Each call to List_Insert() runs to completion; key 2 is inserted first at the head of the list, then 6, which gives the list: 6, 2, 3, 4, 5

68) Schedule: SSSSTTTT

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

Answer: (a). Each call to List_Insert() runs to completion; key 6 is inserted first, then 2, which gives the list: 2, 6, 3, 4, 5

69) Schedule: SSTTTTSS

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

Answer: (b) A context switch away from S occurs after new->key has been set to 6; then key 2 is completely and correctly added to the list; then key 6 is added to the list.

70) Schedule: SSSTTTTS

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

Answer: d. A context switch occurs away from S after `new->next = L->head` *(without key 2); Key 2 is then added to the head of the list. However, when S is resumed, it executes* `L->head = new;` *this causes the list to point to 6 (which points to 3, 4, 5); thus key 2 is lost from the list.*

71) If the above code has a race condition, how could it be fixed? **Indicate all that apply**.

a) Code does not have a race condition
b) Lock
c) Condition variable
d) Semaphore
e) Atomic compare and swap instruction

Official Answer: Need the two instruction assignment of new->next and L->head to be atomic. Code can be fixed with a lock, a semaphores (which can be used as mutex by initializing to 1), AND with an atomic compare and swap instruction. Correct answers should have marked all 3, but everyone was given credit for this question.

End of Part 6b

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

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

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

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

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

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

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

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

Answer: a2->b. Since l->guard starts as false, statement a1 is executed once; T will resume with a2.

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

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

Answer: a8->c. First T executes a2. Since l->lock is false, T next executes a7. Next statement will be a8.

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

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

Answer: a1->a. Since guard=true, S will keep looping on this first statement.

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

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

Answer: d. Executes a8 and continues past a8.

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

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

Answer: a4->b. The guard=false now, so executes a1 one more time, then a2. In a2 sees that lock=true, so executes a3. Next time will execute a4.

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

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

Answer: R1->a. S is still holding the guard, so T will be stuck trying to acquire the guard in the release() code.

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

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

Answer: c. The guard must be acquired before l->lock can be examined or set.

End of Part 6c.

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

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

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

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

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

Answer: d. Mutex is not locked, numfull != max, so just runs through all lines: p1, p2, p4, p5, p6. Next time it is scheduled, it will run a line after P6.

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

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

Answer: d. Mutex is not locked, numfull is 1, so just runs through lines: c1, c2, c4, c5, c6. Next instruction will be after c6.

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

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

Answer: c4 -> c. Mutex is not locked, but numfull is 0, so executes c1, c2, c3. Now, we see that the consumer is incorrectly waiting on a semaphore fill that was initialized to numfull. As a result, the consumer does NOT end up waiting here and the next time it runs it will execute c4.

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

a) p1
b) p3

c) p5
d) Some line after p6
e) None of the above

Answer: p1->a. Since the mutex is currently held by O, R will be stuck waiting to acquire the lock.

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

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

Answer: d or e (points given after initial grading). A common answer was to say that the code would work if we corrected how the semaphores were initialized – but this is actually not enough! Semaphores do NOT release the mutex when sema_wait() is called; therefore, the system will be in a deadlock if a process calls sema_wait() because it will hold the mutex and the process that needs to call sema_post() will never be able to grab the lock and run. The only correct way to fix the code is to use condition variables (which do not require initialization) AND change the if() loop to a while() loop.

End of Exam! Congratulations!