Midterm2: Concurrency

Time:

Points: **Questions:**

Instructions

This exam is a closed book, closed notes. When you begin the exam, you will need to agree with the following statements: I understand that this exam must be worked on independently. I will not communicate or collaborate with any others during the exam in any way, for example:

- *I will not chat, text, or post any questions or information*
- *I will not ask for, receive, or provide help to others*
- *I will notify the instructor if I am aware of others violating this policy*

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

- *The OS manages a multiprocessor (multi-core) that can run multiple threads simultaneously*
- *Leading zeros can be removed from numbers (e.g., 0x06 == 0x6)*
- *Hex numbers are represented with a preceding "0x"*
- *All the code mentioned in the paper is assumed to compile properly. You can ignore any syntax error if there is any.*

You have 75 minutes to complete this exam.

Section		Question Type	No. of Questions	Total Points
Concurrency		True/False	39	39
		Multiple Choice	11	11
Multi-part (Concurrency)	Building Locks	Multiple Choice	5	5
	Producer & Consumer		8	8
	Thread join() and wait()		7	7
	Semaphores		5	5

Section - 1: In this first section, you will be answering True/False questions related to Concurrency. There are 39 questions, and each question is worth 1 point.

1. All threads share instruction pointer. [1 point]
    a. True
    b. False

2. Porting code with user-level threads compared to kernel-level threads is easy. [1 point]
    a. True
    b. False

3. By ensuring that the following critical section executes atomically, one can avoid race conditions in a uniprocessor environment. Assume 0x123 is the address of a global variable. [1 point]

    mov 0x123, %eax
    add %0x1, %eax
    mov %eax, 0x123

    a. True
    b. False

4. Given two processes P1 and P2, one can write a program that can guarantee mutual exclusion between the two by using only the load-store instructions that the modern hardware provides. [1 point]
    a. True
    b. False

5. Turning off the interrupts for critical sections can make one process keep control of a CPU for an arbitrary length of time. [1 point]
    a. True
    b. False

6. The OS CPU scheduler is aware of the lock(), and unlock() calls being called by the process. [1 point]
    a. True
    b. False

7. Ticket locks guarantee bounded waiting. [1 point]
    a. True
    b. False

8. While reasoning about locks, we can always assume a particular ordering of which processes will acquire the lock. [1 point]
    a. True
    b. False

9. If two threads in the same process access the same general-purpose hardware register, they will read/write the same register. [1 point]
    a. True
    b. False

10. If the runtime (total time taken by the process to complete) of a process is x, doubling the number of threads in the process will always lead to a runtime of x/2. Assume the process is running on a multi-core processor. [1 point]
    a. True
    b. False

11. Concurrency always produces deterministic results. [1 point]
    a. True
    b. False

12. A "critical section" of code is where a thread or process initializes non-shared variables. [1 point]
    a. True
    b. False

13. The page table base register (PTBR) is different for different threads belonging to the same process. [1 point]
    a. True
    b. False

14. Mutual Exclusion and Ordering are two objectives of synchronization. [1 point]
    a. True
    b. False

15. A condition variable is a queue of waiting threads. [1 point]
    a. True
    b. False

16. One can safely modify the state associated with condition variables without holding a mutex. [1 point]
    a. True
    b. False

17. After waking up from cond_wait(), there is no need to recheck the assumptions about the state. [1 point]
    a. True
    b. False

18. It is always correct (the code will do the right thing, even if it may be slower) to use a cond_broadcast() instead of a cond_signal() for code using while() loops. [1 point]
    a. True
    b. False

19. A cond_broadcast() call only wakes up one waiting thread. [1 point]
    a. True
    b. False

20. The counter value in semaphores can be directly accessed by the user program. [1 point]
    a. True
    b. False

21. Equivalence means semaphores can be used to address the same synchronization problems as condition variables + locks. [1 point]
    a. True
    b. False

22. While building a mutual exclusion lock using a semaphore, the semaphore value should be initiated to 1. [1 point]
    a. True
    b. False

23. Semaphores can be used for both mutual exclusion and ordering. [1 point]
    a. True
    b. False

24. Incorrect behavior cannot lead to catastrophic results. [1 point]
    a. True
    b. False

25. A deadlock can be eliminated by eliminating any one condition that leads to deadlocks. [1 point]
    a. True
    b. False

26. Encapsulation helps in preventing deadlocks. [1 point]
    a. True
    b. False

27. Using atomic instructions instead of using locks can help eliminate deadlocks. [1 point]
    a. True
    b. False

28. To avoid the Hold-and-Wait condition, all the locks must be acquired atomically. [1 point]
    a. True
    b. False

29. To avoid deadlock, if a thread cannot get the resources it wants, it may release all the resources it holds. [1 point]
    a. True
    b. False

30. Lock ordering is not important to address deadlocks. [1 point]
    a. True
    b. False

31. In a multi-threaded application with condition variables, after a thread returns from cond_wait(&cv, &mutex), it can assume that it holds the mutex lock. [1 point]
    a. True
    b. False

32. Two different condition variables can be associated with the same mutex lock; that is, wait could be called on two different condition variables when the same lock is held. [1 point]
    a. True
    b. False

33. With a reader-writer lock, either multiple readers or a single writer can hold a lock. [1 point]
    a. True
    b. False

34. Deadlock will always occur if thread A acquires two locks, X and Y, in the order X, Y while thread B acquires the lock in the order Y, X. [1 point]
    a. True
    b. False

35. Deadlock and Livelock are both the same problems - i.e., the techniques for fixing deadlock also fix livelock. [1 point]
    a. True
    b. False

36. In applications with multiple producers and consumer threads operating on a shared array, the number of producers should equal the number of elements in the array. [1 point]
    a. True
    b. False

37. For shorter critical sections, spinlocks are better as they avoid the overhead of context switches. [1 point]
    a. True
    b. False

38. When a lock developer uses a yield() call to relinquish the CPU while waiting to acquire the lock, the CPU cycles wasted (i.e., not doing useful work) by a lock can be calculated by determining the number of threads participating in the lock acquisition process and the context switch cost.
    a. True
    b. False

39. When the critical section is very large/long, it is always a good idea to block the waiting thread and put that thread on a waiting queue instead of spinning.
    a. True
    b. False

Section - 2: Congratulations on completing the True/False questions about Concurrency. In the second section, you will be answering Multiple Choice questions related to Concurrency. There is only one answer for each question. There are 11 questions, and each question is worth 1 point.

40. During which time, the CPU clock speed increased the least? [1 point]
    a. 1978-1986
    b. 1986-2003
    c. 2004-2011
    d. 2011-2018

41. Which is the most inefficient way to utilize multiple CPU's available in today's machine? [1 point]
    a. Run multiple applications on the machine
    b. Run multi-threaded application
    c. Run the single-threaded application
    d. None of the above.

42. Which of the following option (only one) is not true for user-level threads? [1 point]
    a. OS is not aware of the user-level threads.
    b. Operations are very fast.
    c. Must rely on the OS schedulers for scheduling.
    d. Entire process blocks when one thread blocks.

43. Condition variables are harder to use than semaphores for synchronization problems that: [1 point]
    a. Require the first thread that arrives at a section of code to wait
    b. Allow threads to wait for different conditions
    c. May wake up multiple threads at once
    d. Control access to a fixed count of resources

44. Which of the following option (only one) is not a part of the Lock Implementation Goals? [1 point]
    a. Only one thread in the critical section at a time.
    b. Must eventually allow each waiting thread to acquire the lock.
    c. Each thread gets the lock in some defined order.
    d. Fast to acquire the lock if no contention with other threads
    e. None of the above

45. Which of the following option (only one) is not true for Spinlock? [1 point]
    a. Spinlocks help in avoiding context switching costs.
    b. Spinning is wasteful, especially when locks are held for a long time.
    c. Spinlocks guarantee non-starvation free behavior and fairness.
    d. Spinlocks are better when the critical section size is small.

46. Which of the following option (only one) is not needed for a deadlock to happen? [1 point]
    a. Mutual exclusion
    b. Lock Ordering
    c. Hold-and-Wait
    d. No preemption
    e. Circular wait

47. For the following code:

```
void add (int *val, int amt) {
   do {
      int old = *val;
   } while(!CompAndSwap(val, old, old+amt);
}
```

Assuming *val = 0, if two threads call add(&val, 2) and add(&val, 3), what are the possible values after both threads return from add()? [1 point]

    a. 0, 3, 5
    b. 0, 2, 3
    c. 0, 2, 5
    d. None of the above

48. You and your project partner are working on a project and want to talk. Both of you decide to call at the exact same time and find the other person is busy. So, both of you hang up and call again simultaneously. Both of you keep on doing the same again and again. What is this condition called? [1 point]
    a. Deadlock
    b. Livelock
    c. Deadlock & Livelock
    d. Neither a deadlock nor a livelock.

49. How do you solve the condition specified in Question 48? [1 point]
    a. Stop talking to your partner and find a new partner.
    b. Random/Exponential back-off.
    c. There is no problem here – so nothing there to solve.
    d. Complain about the partner to the instructor and TA.

50. Which of the following statement is not true for Reader-Writer locks? [1 point]
    a. Writers wait while one or more readers are holding the lock in shared mode.
    b. Multiple readers can hold the lock along with a single writer.
    c. Only one writer is allowed to execute the critical section.
    d. Writers and readers need to execute in a mutually exclusive way.

Section - 3: Congratulations on completing the Multiple-Choice questions about Concurrency. In the third section, you will now have four multi-part questions covering different aspects of Concurrency.

Part 1. In this question, you will be building a lock using atomic instructions.

Consider the following spinlock code that you need to implement. The lock code will look as follows. [Total 5 points]

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

/* init function */
void init(lock_t *lock) {
        lock->flag = VALUE
}

/* Lock acquire function */
void acquire(lock_t *lock) {
        while (CONDITION) {
                <SOMETHING>
        }
}
```

```
/* Lock release function */
void release(lock _t *lock) }
        lock->flag = VALUE
}
```

Suppose that you are using XCHG instruction to implement the lock. **You can assume that the lock is held when the flag value of the lock is 0; while the lock is free when the flag value of the lock is 1. Note how the values are different than we studied in the class.**

The logic of the XCHG instruction is as follows:

```
int xchg(int * addr, int newval) {
        int old = *addr;
        *addr = newval;
        return old;
}
```

51. What should be the value of the VALUE in the init() and release() function respectively? [1 point]
    a. 0 and 0
    b. 1 and 1
    c. 1 and 0
    d. 0 and 1

52. What CONDITION should be there to ensure mutual exclusion is guaranteed using **xchg** instruction in acquire()? [1 point]
    a. while(xchg(&lock->flag, 1) == 1)
    b. while(xchg(&lock->flag, 0) == 1)
    c. while(xchg(&lock->flag, 1) == 0)
    d. while(xchg(&lock->flag, 0) == 0)

Suppose that you are using CompareAndSwap instruction to implement the lock. **You can assume that the lock is held when the flag value of the lock is 1; while the lock is free when the flag value of the lock is 0.**

The logic of the CompareAndSwap instruction is as follows:

```
int CompareAndSwap(int *addr, int expected, int new) {
        int actual = *addr;
        if (actual == expected)
                *addr = new;
        return actual;
}
```

53. What should be the value of the VALUE in the init() and release() function respectively? [1 point]
    a. 0 and 0
    b. 1 and 1
    c. 1 and 0
    d. 0 and 1

54. What CONDITION should be there to ensure mutual exclusion is guaranteed using **CompareAndSwap** instruction in acquire()? [1 point]
    a. while (CompareAndSwap(&lock->flag, 1, 1) == 1)
    b. while (CompareAndSwap(&lock->flag, 1, 0) == 0)
    c. while (CompareAndSwap(&lock->flag, 0, 1) == 1)
    d. while (CompareAndSwap(&lock->flag, 0, 0) == 0)

55. You are worried about the performance of the lock as the lock spins while waiting to acquire the lock. What line of code can you add in the acquire() function and replace <SOMETHING> with to improve the performance and let other threads run while the thread waits to acquire the lock? [1 point]
    a. park();
    b. yield();
    c. unpark();
    d. spin();

Part II. In this question, you will analyze the behavior of a multi-threaded producer-consumer application. [Total 8 points]

The following code is intended to be similar to what has been shown in lecture and should be correct; assume it is successfully compiled and run (e.g., ignore any capitalization problems). Everything is correctly initialized, including the mutex.

Assume there are two producer threads and two consumer threads in this application. This code is run with a shared buffer with four elements;.

- A global variable **numempty** is initialized to 4, and **numfull** is initialized to 0.
- **tmp** is a local per-thread variable.
- The producer routine **get_empty**() finds an "empty" element in the shared buffer, marks that element as being "in progress", and decrements numempty;
- **mark_full**() marks the designated element(tmp) as "full" and increments numfull.
- The consumer routines of **get_full**() and **mark_empty**() do the corresponding (opposite) operations.

The application has been running for some unknown amount of time and is in an unknown (but correct) state. It is up to you to determine where in the code it is possible for those 4 threads to currently be!

```
void *producer(void *arg) {
   while (1) {
      Mutex_lock(&m);  // p1
      while (numempty == 0) { // p2
         Cond_wait(&full, &m); // p3
       }
      tmp = get_empty(); //p4 -- decr numempty
      Mutex_unlock(&m);  //p5
      fill_buffer(tmp);  //p6
      Mutex_lock(&m);       // p7
      mark_full(tmp);          // p8 -- incr numfull
      Cond_signal(&empty); //p9
      Mutex_unlock(&m);   //p10
   }
}

void *consumer(void *arg) {
   while (1) {
      Mutex_lock(&m);    // c1
      while (numfull == 0) {  // c2
         Cond_wait(&empty, &m);// c3
      }
      tmp = get_full();  // c4 -- decr numfull
      Mutex_unlock(&m);  // c5
      use_buffer(tmp);   // c6
      Mutex_lock(&m);       // c7
      mark_empty(tmp);          // c8  -- incr numempty
      Cond_signal(&full);// c9
      Mutex_unlock(&m);  // c10
   }
}
```

Which of the following are possible locations in the code for the two producer threads, named Pa and Pb, and two consumer threads, named Ca and Cb? The locations denote that **the thread is about to start executing the line number specified**. For cond_wait(sleeping), the thread has executed the call and is currently sleeping.

56. Pa: p6(fill_buffer), Pb: p6(fill_buffer) Ca:c4 (get_full), Cb: c4 (get_full) [1 point]
    a. Possible
    b. Not possible

57. Pa: p1 (mutex_lock), Pb: p1 (mutex_lock), Ca: c1 (mutex_lock), Cb: c1 (mutex_lock) [1 point]
    a. Possible
    b. Not possible

58. Pa: p6 (fill_buffer), Pb: p4(get_empty), Ca: c6 (use_buffer), Cb: c6(use_buffer) [1 point]
    a. Possible
    b. Not possible

59. Pa: p3(cond_wait), Pb: p3(cond_wait), Ca: c8(mark_empty), Cb: c6(use_buffer) [1 point]
    a. Possible
    b. Not possible

60. Pa: p6(fill_buffer), Pb: p6(fill_buffer), Ca: c8(mark_empty), c9(cond_signal) [1 point]
    a. Possible
    b. Not possible

61. Pa: p3(cond_wait - sleeping), Pb: p3(cond_wait - sleeping), Ca: c6(use_buffer), Cb: c6(use_buffer) [1 point]
    a. Possible
    b. Not possible

62. Pa: p6(fill_buffer), Pb: p4(get_empty), Ca: c6(use_buffer), Cb: c4(get_full) [1 point]
    a. Possible
    b. Not possible

63. Pa: p3(cond_wait - sleeping), Pb: p6(fill_buffer), Ca: c6(use_buffer), Cb: c8(mark_empty) [1 point]
    a. Possible
    b. Not possible

Part III. In this question, you will explore the behavior of implementations of thread_join() and thread_exit() that may be incorrect. Whether or not the implementation could incur problems at run-time might depend on the exact scheduling of threads. [7 points]

To understand how the scheduler switches between threads, you must understand the following model. Imagine you have two threads, T and S. The scheduler runs T and S such that each statement in the C-language code (or line of code as written in our examples) is atomic, and each line of code takes one scheduler tick to complete. 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 three lines were run from thread T followed by two lines from thread S.

Assume jumping to the correct code does not take an additional tick beyond the test (e.g., jumping to the then branch of an if statement does not take an extra 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 follows.

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(); not that T must be scheduled for one tick with the lock released for mutex_lock() to complete).

The rules for cond_wait() are similar. When a thread calls cond_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 must be scheduled for one tick with the work completed for wait() to complete).

If the shown schedule results in one thread finishing the shown code, assume the thread continues executing code that is not related.

A function completes when the last line of code in the function finishes executing.

Assume the following code compiles correctly. Done is a shared variable and initialized correctly. Assume the parent thread has just called thread_join(); when it is next scheduled, it will execute p1. Assume the child thread is about to call thread_exit(); when it is next scheduled, it will execute c1. You can assume that a thread finishes a function when it executes the last line of code (i.e., it does not take an extra step to return from the function)

```
void thread_join() {
        mutex_lock(&m);             // p1
        if (done == 0)              //p2
                cond_wait(&c, &m); //p3
        mutex_unlock(&m);           //p4
}

void thread_exit() {
        mutex_lock(&m);             //c1
        done= 1;                    //c2
        mutex_unlock(&m);           //c3
        cond_signal(&c);            //c4
}
```

For each of the following schedules of the parent P, and the child, C, designate one of the options that apply.

64. CCCPPPC [1 point]
    a. The schedule does not yet cause a problem; thread_join or thread_exit is not completed.
    b. The schedule does not trigger any race conditions or problems; thread_join and thread_exit ran to completion, and the parent successfully waited for the child.
    c. The schedule caused a problem (Ordering): Parent will return from thread_join() before Child calls thread_exit().
    d. The schedule caused a problem (Deadlock): Parent and/or child will be stuck forever in thread_join or thread_exit().

65. CCCCPPP [1 point]
    a. The schedule does not yet cause a problem; thread_join or thread_exit is not completed.
    b. The schedule does not trigger any race conditions or problems; thread_join and thread_exit ran to completion, and the parent successfully waited for the child.
    c. The schedule caused a problem (Ordering): Parent will return from thread_join() before Child calls thread_exit().
    d. The schedule caused a problem (Deadlock): Parent and/or child will be stuck forever in thread_join or thread_exit().

66. CCPPPPPCC [1 point]
    a. The schedule does not yet cause a problem; thread_join or thread_exit is not completed.
    b. The schedule does not trigger any race conditions or problems; thread_join and thread_exit ran to completion, and the parent successfully waited for the child.
    c. The schedule caused a problem (Ordering): Parent will return from thread_join() before Child calls thread_exit().
    d. The schedule caused a problem (Deadlock): Parent and/or child will be stuck forever in thread_join or thread_exit().

67. PPCCPPPPC [1 point]
    a. The schedule does not yet cause a problem; thread_join or thread_exit is not completed.
    b. The schedule does not trigger any race conditions or problems; thread_join and thread_exit ran to completion, and the parent successfully waited for the child.
    c. The schedule caused a problem (Ordering): Parent will return from thread_join() before Child calls thread_exit().
    d. The schedule caused a problem (Deadlock): Parent and/or child will be stuck forever in thread_join or thread_exit().

68. PPCCCPPPP [1 point]
    a. The schedule does not yet cause a problem; thread_join or thread_exit is not completed.
    b. The schedule does not trigger any race conditions or problems; thread_join and thread_exit ran to completion, and the parent successfully waited for the child.
    c. The schedule caused a problem (Ordering): Parent will return from thread_join() before Child calls thread_exit().
    d. The schedule caused a problem (Deadlock): Parent and/or child will be stuck forever in thread_join or thread_exit().

69. PPPCCCCPP [1 point]
    a. The schedule does not yet cause a problem; thread_join or thread_exit is not completed.
    b. The schedule does not trigger any race conditions or problems; thread_join and thread_exit ran to completion, and the parent successfully waited for the child.
    c. The schedule caused a problem (Ordering): Parent will return from thread_join() before Child calls thread_exit().
    d. The schedule caused a problem (Deadlock): Parent and/or child will be stuck forever in thread_join or thread_exit().

70. PPPCCCPPPC [1 point]
    a. The schedule does not yet cause a problem; thread_join or thread_exit is not completed.
    b. The schedule does not trigger any race conditions or problems; thread_join and thread_exit ran to completion, and the parent successfully waited for the child.
    c. The schedule caused a problem (Ordering): Parent will return from thread_join() before Child calls thread_exit().
    d. The schedule caused a problem (Deadlock): Parent and/or child will be stuck forever in thread_join or thread_exit().

Part IV. In this question, you will explore the functioning of semaphores. [4 points]

Consider you are given a game with 3 players and 3 colors (orange, yellow, and red). Each player can be treated as a thread. You must ensure that the players only move pieces in a particular order of the colors. You write the following three routines for the players to call: MoveOrange(), MoveYellow(), and MoveRed(). You use three semaphores to maintain the proper order and initialize the values to X, Y, and Z.

semaphore orange = X
semaphore yellow = Y
semaphore red = Z

```
MoveOrange() {
   sem_wait(orange);
   printf("Orange color\n");
   sem_signal(yellow);
}
```

```
MoveYellow() {
   sem_wait(yellow);
   printf("Yellow color\n");
   sem_signal(red);
}
```

```
MoveRed() {
   sem_wait(red);
   printf("Red color\n");
   sem_signal(orange);
}
```

71. If the order of the printf statements is: [1 point]
    Red color
    Orange color
    Yellow color
    What is the value of the semaphore values X, Y, and Z?
    a. 1, 1, 1
    b. 1, 0, 0
    c. 0, 1, 0
    d. 0, 0, 1

72. You noticed that for a particular set of X, Y, and Z values, the printf statements can print in any order. Can you identify for which value of X, Y, and Z this is possible? [1 point]
    a. 0, 0, 0
    b. 1, 1, 1
    c. Both (a) and (b)
    d. Neither (a) nor (b)

73. With the right choice of X, Y and Z, is it possible to ensure the following is printed? [1 point]
    Red color
    Red color
    Orange color
    Orange color
    Yellow color
    Yellow color
    a. Yes
    b. No

74. You notice that the program did not print any output. For what values of X, Y, and Z, such a scenario is possible? [1 point]
    a. 1, 1, 1
    b. 0, 0, 0
    c. Both (a) and (b)
    d. None of the above

75. Can using MoveOrange(), MoveYellow(), and MoveRed() in any order lead to a deadlock? [1 point]
    a. Yes
    b. No