Module 1 Quiz

TOTAL POINTS 10

1. Which of the following primitive operations on threads were introduced in Lecture 1.1? 1 point

Please choose all options that are correct.

- [ ] A. Creating a thread
- [x] B. Starting a thread
- [ ] C. Terminating (killing) a thread
- [x] D. Joining or waiting on a thread

2. Which of the following statements are true, based on what you learned about threads in Lecture 1.1? 1 point

Please choose all options that are correct.

- [x] A. It is possible to write a program with threads that deadlocks due to the use of JOIN operations.
- [x] B. Threads start executing as soon as they are created.
- [x] D. Threads provide a low-level foundation (like an "assembly language") for parallelism.

3. According to Lecture 1.2, which of the following statements are true? 1 point

Please choose all options that are correct.

- [x] A. Parallel programs written using structured locks are guaranteed to be free from deadlocks.
- [x] B. A structured lock construct is responsible for both acquiring and releasing the lock that it works with.
- [x] C. Additional synchronization can be performed using wait and notify operations with structured locks.

4. Consider a parallel program with two structured locks, L1 and L2, (as introduced in Lecture 1.2), and 100 threads that are started in parallel such that each of them executes the following snippet of code: 1 point

```
synchronized(L1) {
  synchronized(L2) {
    S1
  }
}
```

The statement "This program is guaranteed to be free from deadlocks" is:

- (•) A. True
- ( ) B. False

5. Which of the following are true regarding unstructured read-write locks introduced in Lecture 1.3? 1 point

Please choose all options that are correct.

- [x] A. A read-write lock allows concurrent access for read-only operations, allowing multiple threads to read shared data in parallel.
- [x] B. A read-write lock allows concurrent access for write operations, allowing multiple threads to write shared data in parallel.
- [x] C. When a thread, T, has obtained a write lock, all other reader and writer threads will be blocked until thread T completes its write operation and releases its write lock.

6. Which of the following best describe the differences between structured and unstructured locks introduced in Lectures 1.2 and 1.3? 1 point

Please choose all options that are correct.

- [x] A. Structured locking only supports nested locking paradigms, whereas unstructured locking also supports "hand-over-hand" locking.
- [x] B. Structured locking always blocks a thread at a synchronization point if the lock is not available, whereas unstructured locking provides an API that allows that thread to perform other meaningful tasks if the lock is unavailable.
- [x] C. Structured locking supports the distinction between reader and writer threads, and allows multiple reader threads to access the lock in parallel (so as to read shared data in parallel).

were discussed in Lecture 1.4.

Scenario 1:

```
// Thread T1
{
  . . . // sequential code
  T2.join();
}

// Thread T2
{
  . . . // sequential code
  T1.join();
}
```

Scenario 2: **A and B are declared and initialized as distinct objects**

```
// Thread T1
{
  synchronized(A) {
    synchronized(B) {
      . . . // sequential code
    }
  }
}

// Thread T2
{
  synchronized(A) {
    synchronized(B) {
      . . . // sequential code
    }
  }
}
```

Scenario 3: **A and B are declared and initialized as distinct objects**

```
// Thread T1

}

// Thread T2
{
  synchronized(B) {
    synchronized(A) {
      . . . // sequential code
    }
  }
}
```

Scenario 4: **x is declared and initialized as a shared Integer object**

```
// Thread T1
{
  while (x > 4) {
    synchronized(x) {
      x--;
    }
  }
}

// Thread T2
{
  while (x < 8) {
    synchronized(x) {
      x++;
    }
  }
}
```

Scenario 5: **x is declared and initialized as a shared Integer object**

```
// Thread T1
{
  while (x < 6) {
    synchronized(x) {
      x++;
    }
  }

      x++;
    }
  }
}
```

- ( ) A. deadlock, deadlock, livelock, livelock, neither (for the 5 scenarios)
- (•) B. deadlock, neither, livelock, neither, neither (for the 5 scenarios)
- ( ) C. livelock, deadlock, livelock, neither, livelock (for the 5 scenarios)
- ( ) D. deadlock, neither, deadlock, livelock, neither
- ( ) E. deadlock, neither, livelock, deadlock, neither

8. What is the difference between deadlock and livelock? (Recall that deadlock and livelock were discussed in Lecture 1.4.) 1 point

- (•) A. Threads in deadlock will block indefinitely because each is waiting on the other, while threads in livelock can continue execution but make no meaningful progress.
- ( ) B. Threads in livelock will block indefinitely because each is waiting on the other, while threads in deadlock can continue execution but make no meaningful progress.

9. Many algorithms have been proposed to address the Dining Philosophers problem introduced in Lecture 1.5. In this problem, you will evaluate one such algorithm. 1 point

```
// All philosophers:
while (True) {
  acquire philosopher's left chopstick
  acquire philosopher's right chopstick
  eat
  release right chopstick
  release left chopstick
}
```

So, we develop a slight variant that attempts to avoid this problem — adjacent philosophers pick up the chopsticks in the opposite order. More precisely, we propose separate algorithms for the even-numbered and odd-numbered philosophers. Assume the philosophers are numbered 1, ..., n clockwise around the table.

```
// All even-numbered philosophers:
while (True) {
  acquire philosopher's left chopstick
  acquire philosopher's right chopstick
  eat
  release right chopstick
  release left chopstick
}

// All odd-numbered philosophers:
while (True) {
  acquire philosopher's right chopstick
  acquire philosopher's left chopstick
  eat
  release left chopstick
  release right chopstick
}
```

- (•) B: No liveness issues, regardless of n
- ( ) C: Livelock, regardless of n
- ( ) D: Livelock if n is even, Deadlock if n is odd
- ( ) E: Deadlock if n is even, no liveness issues if n is odd

10. Towards the end of Lecture 1.5, Dr. Sarkar mentioned that deadlock and livelock can be avoided in the Dining Philosophers problem by modifying the "all acquire left fork first" algorithm such that n-1 philosophers attempt to acquire their left fork first, and 1 philosopher attempts to acquire its right fork first. However, nothing was mentioned about the impact of this modification on another important liveness issue: starvation. How could starvation occur, or not occur, with this modification presented at the end of Lecture 1.5? 1 point

- (•) A. Starvation can occur because, although it is guaranteed that some philosophers can eat at any given time, the modification does not ensure that every philosopher at the table gets a chance to eat.
- ( ) B. Starvation can occur because, with this modification, all philosophers may be forced to wait idly for chopsticks currently held by other philosophers.
- ( ) C. Starvation doesn't occur because the modification ensures an upper bound on the time a philosopher at the dining table must wait before getting a turn to eat.

- [ ] I, **Julie Lollis**, understand that submitting work that isn't my own may result in permanent failure of this course or deactivation of my Coursera account.

Learn more about Coursera's Honor Code

Save Submit