Started on	Wednesday, 26 May 2021, 15:50
State	Finished
Completed on	Tuesday, 1 June 2021, 13:21
Time taken	5 days 21 hours
Grade	**30.83** out of 130.00 (**24**%)

Question 1

Partially correct | Mark 2.50 out of 10.00

In the `main` method of `Factorial` class in the lecture:

```
public static void main(String[] args) {
    new Thread(new Runnable() { // create a thread using an
        public void run() {     // anonymous Runnable
            computeFact(99);
        }
    }).start();
    computeFact(100);
}
```

Which of the following are possible interleavings ?

Select one or more:

- [ ] i. The call to computeFact(99) finishes before the call to computeFact(100) starts
- [x] ii. The call to computeFact(100) starts before the call to computeFact(99) starts
- [ ] iii. The call to computeFact(99) starts before the call to computeFact(100) starts
- [ ] iv. The call to computeFact(100) finishes before the call to computeFact(99) starts

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: The call to computeFact(99) finishes before the call to computeFact(100) starts, The call to computeFact(100) starts before the call to computeFact(99) starts, The call to computeFact(99) starts before the call to computeFact(100) starts, The call to computeFact(100) finishes before the call to computeFact(99) starts

Question 2

Incorrect | Mark 0.00 out of 10.00

Now consider this different `main` that runs three factorial computations:

```
public static void main(String[] args) {
    computeFact(98);
    Thread t = new Thread(new Runnable() {
        public void run() {
            computeFact(99);
```

```
        });
        computeFact(100);
        t.start();
    }
```

Which of the following are possible ordering of events?

Select one or more:

- i. The call to computeFact(98) starts before the call to computeFact(99) starts
- ii. The call to computeFact(99) starts before the call to computeFact(100) starts
- iii. The call to computeFact(99) finishes before the call to computeFact(98) starts
- iv. The call to computeFact(100) finishes before the call to computeFact(99) starts

Your answer is incorrect.
In this code, `computeFact(98)` first executes serially, from start to end. Then a thread for `computeFact(99)` is created *but not started yet*.
Then `computeFact(100)` executes serially from start to end, and finally thread `t` is started to finally run `computeFact(99)`. The `main` method then returns and the main thread finishes running, but the `t` thread continues until it finishes `computeFact(99)` and returns from its `run` method.

The effect of this code is actually three serialized calls: `computeFact(98)` followed by `computeFact(100)` followed by `computeFact(99)` So the first and fourth ordering above *always* happen, but the second and third *never* happen.

Be careful to understand the difference between creating a `Thread` object and `start`ing it.

The correct answers are: The call to computeFact(98) starts before the call to computeFact(99) starts, The call to computeFact(100) finishes before the call to computeFact(99) starts

Question 3

Partially correct | Mark 3.33 out of 10.00

Here's part of the pinball simulator example from the lecture:

```
public class PinballSimulator {

    private static PinballSimulator simulator = null;

    // invariant: there should never be more than one PinballSimulator object created

    public static PinballSimulator getInstance() {
/* 1 */     if (simulator == null) {
/* 2 */         simulator = new PinballSimulator();
            }
/* 3 */     return simulator;
    }
}
```

We want to have the invariant that only one simulator object is created.

Suppose two threads are running `getInstance()`.

One thread is about to execute one of the numbered lines above; the other thread is about to execute the other.

For each pair of possible line numbers, is it possible the invariant will be violated?

Select one or more:

- i. about to execute lines 1 and 3
- ii. about to execute lines 1 and 2
- iii. about to execute lines 1 and 1

Your answer is partially correct.
You have correctly selected 1

The correct answers are: about to execute lines 1 and 3, about to execute lines 1 and 2, about to execute lines 1 and 1

Question 4

Partially correct | Mark 2.50 out of 10.00

Consider this class's rep:

```
public class Building {
    private final String buildingName;
    private int numberOfFloors;
    private final int[] occupancyPerFloor;
    private final List<String> companyNames = Collections.synchronizedList(new ArrayList<>());
    private final Set<String> roomNumbers = new HashSet<>();
    private final Set<String> floorplan = Collections.synchronizedSet(roomNumbers);
    ...
}
```

Which of these variables refer to a value of a threadsafe data type?

Select one or more:

- [x] i. buildingName
- [x] ii. numberOfFloors
- [x] iii. occupancyPerFloor
- [ ] iv. companyNames
- [ ] v. roomNumbers
- [ ] vi. floorplan

Your answer is partially correct.

`buildingName` has type `String`, and `numberOfFloors` has type `int`. Both types are immutable, so the values of the types are threadsafe.

`occupancyPerFloor` has type `int[]`, which is mutable and not threadsafe.

`companyNames` has type `List<String>`, which is not automatically threadsafe, but the implementation of `List<String>` used here is a synchronized list wrapper, and `companyNames` is final so it can never be assigned to a different `List`, so this type is threadsafe.

Similarly, the actual type of the `roomNumbers` value is `HashSet`, which is not threadsafe.

But the synchronized set wrapper is threadsafe, so `floorplan` points to a value of a threadsafe data type.

You have correctly selected 2.
The correct answers are: buildingName, numberOfFloors, companyNames,
floorplan

Question 5

Correct | Mark 10.00 out of 10.00

Consider this class's rep:

```
public class Building {
    private final String buildingName;
    private int numberOfFloors;
    private final int[] occupancyPerFloor;
    private final List<String> companyNames = Collections.synchronizedList(new ArrayList<>());
    private final Set<String> roomNumbers = new HashSet<>();
    private final Set<String> floorplan = Collections.synchronizedSet(roomNumbers);
    ...
}
```

Which of these variables are safe for use by multiple threads?

Select one or more:

- [x] i. buildingName
- [ ] ii. numberOfFloors
- [ ] iii. occupancyPerFloor
- [x] iv. companyNames
- [ ] v. roomNumbers
- [ ] vi. floorplan

Your answer is correct.

Not only does the variable's type have to be threadsafe, but the variable itself should be unreassignable. `buildingName` and `companyNames` satisfy that, but reads and writes of `numberOfFloors` may have race conditions.

When using a synchronized collection wrapper, you have to be sure not to keep any aliases to the underlying collection. So `companyNames` is safe because no other variables hold a reference to the underlying `ArrayList`, but `floorplan` is not safe because `roomNumbers` points to the same `HashSet`.

The correct answers are: buildingName, companyNames

Question 6

Incorrect | Mark 0.00 out of 10.00

Consider this class's rep:

```
public class Building {
    private final String buildingName;
    private int numberOfFloors;
    private final int[] occupancyPerFloor;
    private final List<String> companyNames = Collections.synchronizedList(new ArrayList<>());
    private final Set<String> roomNumbers = new HashSet<>();
    private final Set<String> floorplan = Collections.synchronizedSet(roomNumbers);
    ...
}
```

Which of these variables **cannot** be involved in any race condition?

Select one or more:

- [ ] i. buildingName
- [ ] ii. numberOfFloors
- [x] iii. occupancyPerFloor
- [x] iv. companyNames
- [ ] v. roomNumbers
- [ ] vi. floorplan

Your answer is incorrect.

`buildingName` is unreassignable and immutable, so it can't be involved in any race condition.

`companyNames` might still be involved in a race condition caused by (otherwise safe) mutations to the list, e.g.:

```
if (companyNames.size() > 0) { String firstCompany = companyNames.get(0); }
```

If another thread could empty the `companyNames` list between the size check and the `get` call, then this code will fail.

The correct answer is: buildingName

Question 7

Incorrect | Mark 0.00 out of 10.00

If thread B tries to acquire a lock currently held by thread A:

What happens to thread A?

Select one:

- a. blocks until B acquires the lock
- b. blocks until B releases the lock
- c. throws an exception
- d. nothing

Your answer is incorrect.

The correct answer is: nothing

Question 8

Incorrect | Mark 0.00 out of 10.00

If thread B tries to acquire a lock currently held by thread A:

What happens to thread B?

Select one:

- a. blocks until A acquires the lock
- b. blocks until A releases the lock
- c. throws an exception
- d. nothing

Your answer is incorrect.

The correct answer is: blocks until A releases the lock

Question 9

Partially correct | Mark 2.50 out of 10.00

Suppose `list` is an instance of `ArrayList<String>`.

What is true while thread A is in a `synchronized (list) { ... }` block?

Select one or more:

- i. it owns the lock on `list`

i. it owns the lock on `list`

ii. it does not own the lock on `list`

iii. no other thread can use observers of `list`

iv. no other thread can use mutators of `list`

v. no other thread can acquire the lock on `list`

vi. no other thread can acquire locks on elements in `list`

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: it owns the lock on `list`, no other thread can acquire the lock on `list`

Question 10

Incorrect Mark 0.00 out of 10.00

Suppose we run this code:

```
synchronized (obj) {
    // ...
    synchronized (obj) { // 1
        // ...
    }
    // 2
}
```

On the line `// 1` do we experience deadlock?

If we don't deadlock, on the line `// 2`, does the thread own the lock on obj?

Select one:

a. No, we do not experience deadlock; Yes, the thread owns the lock on obj

b. Yes, we experience deadlock; No, the thread does not own the lock on obj

c. Yes, we experience deadlock; Yes, the thread owns the lock on obj

d. No, we do not experience deadlock; No, the thread does not own the lock on obj

Your answer is incorrect.

In Java, a thread is allowed to re-acquire a lock it already owns. The technical term for this is *reentrant locks*.

Acquire and release come in pairs, and `synchronized` blocks on the same object can be safely nested inside each other. This means that a lock actually stores a *counter* of the number of times that its owner has acquired it without yet releasing it. The thread continues to own it until each acquire has had its corresponding release, and the counter has fallen to zero. So on the "do we own the lock" line, the thread does still have a lock on `obj`.

Nested synchronization on the same lock happens frequently, e.g. if a `synchronized` method is recursive, or if one `synchronized` method calls another `synchronized` method on `this`.

The correct answer is: No, we do not experience deadlock; Yes, the thread owns the lock on obj

Question 11

Correct Mark 10.00 out of 10.00

In the code below three threads 1, 2, and 3 are trying to acquire locks on objects `alpha`, `beta`, and `gamma`

In the code below three threads 1, 2, and 3 are trying to acquire locks on objects `alpha`, `beta`, and `gamma`.

Thread 1

```
synchronized (alpha) {
    // using alpha
    // ...
}

synchronized (gamma) {
    synchronized (beta) {
        // using beta & gamma
        // ...
    }
}
// finished
```

Thread 2

```
synchronized (gamma) {
    synchronized (alpha) {
        synchronized (beta) {
            // using alpha, beta, & gamma
            // ...
        }
    }
}
// finished
```

Thread 3

```
synchronized (gamma) {
    synchronized (alpha) {
        // using alpha & gamma
        // ...
    }
}

synchronized (beta) {
    synchronized (gamma) {
        // using beta & gamma
        // ...
    }
}
// finished
```

This system is susceptible to deadlock.

For each of the scenarios below, determine whether the system is in deadlock if the threads are currently on the indicated lines of code.

Scenario A

Thread 1 inside `using alpha`
Thread 2 blocked on `synchronized (alpha)`
Thread 3 finished

Scenario B

Thread 1 finished
Thread 2 blocked on `synchronized (beta)`
Thread 3 blocked on 2nd `synchronized (gamma)`

Select one:

- a. A not deadlock, B deadlock
- b. A deadlock, B deadlock
- c. A deadlock, B not deadlock
- d. A not deadlock, B not deadlock

Your answer is correct.

Scenario A : Thread 1 will exit the top synchronized block, release the lock on `alpha`, and the system will continue.

Scenario B : Thread 2 has acquired the lock on `gamma` and is awaiting `beta`. Thread 3 has `beta` and wants `gamma`. Deadlock.

The correct answer is: A not deadlock, B deadlock

Question 12

Incorrect | Mark 0.00 out of 10.00

In the code below three threads 1, 2, and 3 are trying to acquire locks on objects `alpha`, `beta`, and `gamma`.

Thread 1

```
synchronized (alpha) {
    // using alpha
    // ...
}

synchronized (gamma) {
    synchronized (beta) {
        // using beta & gamma
        // ...
    }
}
// finished
```

Thread 2

```
synchronized (gamma) {
    synchronized (alpha) {
        synchronized (beta) {
            // using alpha, beta, & gamma
            // ...
        }
    }
}
// finished
```

Thread 3

```
synchronized (gamma) {
    synchronized (alpha) {
        // using alpha & gamma
        // ...
    }
}

synchronized (beta) {
    synchronized (gamma) {
        // using beta & gamma
        // ...
    }
}
// finished
```

This system is susceptible to deadlock.

For each of the scenarios below, determine whether the system is in deadlock if the threads are currently on the indicated lines of code.

Scenario C

Thread 1 running `synchronized (beta)`
Thread 2 blocked on `synchronized (gamma)`
Thread 3 blocked on 1st `synchronized (gamma)`

Scenario D

Thread 1 blocked on `synchronized (beta)`
Thread 2 finished
Thread 3 blocked on 2nd `synchronized (gamma)`

Select one:

- a. C not deadlock, D deadlock
- b. C deadlock, D deadlock
- c. C deadlock, D not deadlock
- d. C not deadlock, D not deadlock

Your answer is incorrect.

Scenario C : Thread 1 can successfully acquire the lock on `beta`, then exit the synchronized block, and one of the other threads will be able to acquire the lock on `gamma`. (As we saw in scenario B, they could deadlock later!)

Scenario D : Thread 1 has acquired the lock on `gamma` and is awaiting `beta`. Thread 3 has `beta` and wants `gamma`. Deadlock.

The correct answer is: C not deadlock, D deadlock

Question 13

Incorrect | Mark 0.00 out of 10.00

In the code below three threads 1, 2, and 3 are trying to acquire locks on objects `alpha`, `beta`, and `gamma`.

Thread 1

```
synchronized (alpha) {
```

Thread 2

```
synchronized (gamma) {
```

Thread 3

```
synchronized (gamma) {
```

```
    // using alpha
    // ...
}

synchronized (gamma) {
    synchronized (beta) {
        // using beta & gamma
        // ...
    }
}
// finished
```

```
    synchronized (alpha) {
        synchronized (beta) {
            // using alpha, beta, & gamma
            // ...
        }
    }
}
// finished
```

```
    synchronized (alpha) {
        // using alpha & gamma
        // ...
    }
}

synchronized (beta) {
    synchronized (gamma) {
        // using beta & gamma
        // ...
    }
}
// finished
```

This system is susceptible to deadlock.

In the previous problem, we saw deadlocks involving `beta` and `gamma`.

What about `alpha`?

Select one:

- a. there are no deadlocks involving `alpha`
- b. there is a possible deadlock where thread 1 owns the lock on `alpha`
- c. there is a possible deadlock where thread 2 owns the lock on `alpha`
- d. there is a possible deadlock where thread 3 owns the lock on `alpha`

Your answer is incorrect.

We can reason about it this way: in order to encounter deadlock, threads must try to acquire locks in different orders, creating a cycle in the graph of who-is-waiting-for-who.

So we look at `alpha` vs. `beta`: are there two threads that try to acquire these locks in the opposite order? No. Only thread 2 acquires them both at the same time.

Next we look at `alpha` vs. `gamma`: are there two threads that try to acquire these locks in the opposite order? No. Both thread 2 and thread 3 acquire both locks, but both of them acquire `gamma` first, then `alpha`.

The correct answer is: there are no deadlocks involving `alpha`

Finish review

Lab 13 Recording Jump to... Lab Exercise 13.1 ARBinHeap C