Section 8: Concurrency (Solutions)

0. You are designing a new social-networking site to take over the world. To handle all the volume you expect, you want to support multiple threads with a fine-grained locking strategy in which each user's profile is protected with a different lock. At the core of your system is this simple class definition:

```
class UserProfile {
   static int id_counter;
   int id; // unique for each account
   int[] friends = new int[9999]; // horrible style
   int numFriends;
   Image[] embarrassingPhotos = new Image[9999];
   UserProfile() { // constructor for new profiles
     id = id_counter++;
     numFriends = 0;
   }
   synchronized void makeFriends(UserProfile newFriend) {
      synchronized(newFriend) {
        if(numFriends == friends.length
           || newFriend.numFriends == newFriend.friends.length)
          throw new TooManyFriendsException();
        friends[numFriends++] = newFriend.id;
        newFriend.friends[newFriend.numFriends++] = id;
      }
   }
   synchronized void removeFriend(UserProfile frenemy) {
      ...
   }
}
```

(a) The constructor has a concurrency error. What is it and how would you fix it? A short English anwser is enough – no code or details required.

(b) The `makeFriends` method has a concurrency error. What is it and how would you fix it? A short English anwser is enough – no code or details required.

Solution:

(a) There is a data race on `id_counter`. Two accounts could get the same `id` if they are created simultaneously by different threads. Or even stranger things could happen. You could synchronize on a lock for `id_counter.`

(b) There is a potential deadlock if there are two objects `obj1` and `obj2` and one thread calls `obj1.makeFriends(obj2)` when another thread calls `obj2.makeFriends(obj1)`. The fix is to acquire locks in a consistent order based on the `id` fields, which are unique.

1. **Concurrency:** The `BubbleTea` class manages a bubble tea order assembled by multiple workers. Multiple threads could be accessing the same `BubbleTea` object. Assume the `Stack` objects ARE THREAD-SAFE, have enough space, and operations on them will not throw an exception.

```
public class BubbleTea {
	private Stack<String> drink = new Stack<String>();
	private Stack<String> toppings = new Stack<String>();
	private final int maxDrinkAmount = 8;



	// Checks if drink has capacity
	public boolean hasCapacity() {

		return drink.size() < maxDrinkAmount;

	}

	// Adds liquid to drink
	public void addLiquid(String liquid) {

		if (hasCapacity()) {

			if (liquid.equals("Milk")) {

				while (hasCapacity()) {

					drink.push("Milk")
				}

			} else {

				drink.push(liquid);

			}

		}

	}

	// Adds newTop to list of toppings to add to drink
	public void addTopping(String newTop) {

		if (newTop.equals("Boba") || newTop.equals("Tapioca")) {

			toppings.push("Bubbles");

		} else {

			toppings.push(newTop);

		}
	}
}
```

a) Does the `BubbleTea` class above have (circle all that apply):

a race condition, potential for deadlock, a data race, none of these

If there are any problems, give an example of when those problems could occur. Be specific!

Assuming stack is thread-safe, a race condition still exists. If two threads attempt to call `addLiquid()` at the same time, they could potentially both pass the `hasCapacity()` test with a value of 7 for `drink.size()`. Then both threads would be free to attempt to push onto the drink stack, exceeding `maxDrinkAmount`.

Although this is not a data race, since a thread-safe stack can't be modified from two threads at the same time, it is definitely a bad interleaving (because exceeding `maxDrinkAmount` violates the expected behavior of the class).

b) Suppose we made the `addTopping` method **`synchronized`**, and changed nothing else in the code. Does this modified `BubbleTea` class above have (circle all that apply):

a race condition, potential for deadlock, a data race, none of these

If there are any FIXED problems, describe why they are FIXED. If there are any NEW problems, give an example of when those problems could occur. Be specific!

Assuming stack is thread-safe, a race condition still exists as described above.

This change does reduce the effective concurrency in the code, however, so it actually makes thing slightly worse.

2) **Concurrency:** The `PhoneMonitor` class tries to help manage how much you use your cell phone each day. Multiple threads can access the same `PhoneMonitor` object. Remember that synchronized gives you reentrancy.

```
1
2  public class PhoneMonitor {
3     private int numMinutes = 0;
4     private int numAccesses = 0;
5     private int maxMinutes = 200;
6     private int maxAccesses = 10;
7     private boolean phoneOn = true;
8     private Object accessesLock = new Object();
9     private Object minutesLock = new Object();
10
11    public void accessPhone(int minutes) {
12
13       if (phoneOn) {
14
15          synchronized (accessesLock) {
16
17             synchronized (minutesLock) {
18
19
20
21                numAccesses++;
22                numMinutes += minutes;
23                checkLimits();
24             }
25          }
26       }
27    }
28
29    private void checkLimits() {
30
31       synchronized (minutesLock) {
32
33          synchronized (accessesLock) {
34
35             if ( (numAccesses >= maxAccesses) ||
36                   (numMinutes >= maxMinutes) ) {
37                      phoneOn = false;
38             }
39          }
40       }
41    }
42 }
```

a) Does the `PhoneMonitor` class as shown above have (circle **all** that apply):

a race condition, potential for deadlock, **a data race**, none of these

Justify your answer. Refer to line numbers in your explanation. Be specific!

There is a data race on `phoneOn`. Thread 1 (not needing to hold any locks) could be at line 13 reading `phoneOn`, while Thread 2 is at line 35 (holding both of the locks) writing `phoneOn`. A data race is by definition a type of race condition.

2) (**Continued**)

b) Suppose we made the `checkLimits` method **`public`**, and changed nothing else in the code. Does this modified `PhoneMonitor` class have (circle **all** that apply):

a race condition, **potential for deadlock**, **a data race**, none of these

If there are any FIXED problems, describe why they are FIXED. If there are any NEW problems, give an example of when those problems could occur. Refer to line numbers in your explanation. Be specific!

The same data race still exists, and thus so does the race condition.

By making `checkLimits` method `public`, it is possible for Thread 1 to call `accessPhone` and be at line 17 holding the `accessesLock` lock and trying to get the `minutesLock` lock. Thread 2 could now call `checkLimits` and be at line 31, holding the `minutesLock` lock and trying to get the `accessesLock` lock