## Section 9: Concurrency

# 1. User Profile

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:

```
1  class UserProfile {
2      static int id_counter;
3      int id; // unique for each account
4      int[] friends = new int[9999]; // horrible style
5      int numFriends;
6      Image[] embarrassingPhotos = new Image[9999];
7
8      UserProfile() { // constructor for new profiles
9          id = id_counter++;
10         numFriends = 0;
11     }
12
13     synchronized void makeFriends(UserProfile newFriend) {
14         synchronized(newFriend) {
15             if(numFriends == friends.length
16             || newFriend.numFriends == newFriend.friends.length)
17                 throw new TooManyFriendsException();
18                 friends[numFriends++] = newFriend.id;
19                 newFriend.friends[newFriend.numFriends++] = id;
20         }
21     }
22
23     synchronized void removeFriend(UserProfile frenemy) {
24         ...
25     }
26 }
```

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

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

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.

# 2. Bubble Tea

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.

```
1  public class BubbleTea {
2      private Stack<String> drink = new Stack<String>();
3      private Stack<String> toppings = new Stack<String>();
4      private final int maxDrinkAmount = 8;
5
6      // Checks if drink has capacity
7      public boolean hasCapacity() {
8          return drink.size() < maxDrinkAmount;
9      }
10
11     // Adds liquid to drink
12     public void addLiquid(String liquid) {
13         if (hasCapacity()) {
14             if (liquid.equals("Milk")) {
15                 while (hasCapacity()) {
16                     drink.push("Milk");
17                 }
18             } else {
19                 drink.push(liquid);
20             }
21         }
22     }
23
24     // Adds newTop to list of toppings to add to drink
25     public void addTopping(String newTop) {
26         if (newTop.equals("Boba") || newTop.equals("Tapioca")) {
27              toppings.push("Bubbles");
28         } else {
29             toppings.push(newTop);
30         }
31     }
32 }
```

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!

**a race condition**

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!

**a race condition**

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 things slightly worse.

# 3. Phone Monitor

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

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

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

**a race condition, a data race**

There is a data race on `phoneOn`. Thread 1 (not needing to hold any locks) could be at line 11 reading phoneOn, while Thread 2 is at line 27 (holding both of the locks) writing `phoneOn`. A data race is by definition a type of race condition.

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. Be specific!

**a race condition, potential for deadlock, a data race**

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 13 holding the `accessesLock` lock and trying to get the `minutesLock` lock. Thread 2 could now call `checkLimits` and be at line 24, holding the `minutesLock` lock and trying to get the `accessesLock` lock. Therefore, now there is also potential for deadlock.

# 4. TimeMachine

**5) [16 points total] Concurrency:** The `TimeMachine` class (code for entire class shown below) manages the CSE 332 staff's time machine. Multiple threads can access the same `TimeMachine` object.

```
1  public class TimeMachine {
2     private int now = 1985;
3     private int future = 2015;
4     private int energy = 100;
5
6  ReentrantLock energyLock = new ReentrantLock();
7  ReentrantLock futureLock = new ReentrantLock();
8
9     public boolean hasEnergy() {
10       energyLock.lock();
11       return energy >= 100; boolean result = energy >= 100;
12       energyLock.unlock(); return result;
13    .}
14
15    public void adjustEnergy(int charge) {
16       energyLock.lock();
17       if (energy + charge < 0 ) { // energy should never be negative
18          energyLock.unlock();
19          return;
20       }
21       energy = energy + charge;
22       energyLock.unlock();
23    }
24
25    public void setFuture(int newFuture) {
26       futureLock.lock();
27       future = newFuture;
28       futureLock.unlock();
29    }
30 }
```

a) **[4 pts]** Does the `TimeMachine` class as shown above have (circle **all** that apply):

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

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

**There are multiple data races. A thread could be in `hasEnergy` reading `energy` at line 11 while another thread is at line 21 in `adjustEnergy` writing `energy`. Two threads could also be at line 21 in `adjustEnergy` writing `energy`. Two threads could also be at line 27 in `setFuture` writing `future`. A data race by definition is a type of race condition.**

**5) (Continued)**

**b) [4 pts]** We now add this method to the `TimeMachine` class:

```
28  public boolean backToTheFuture() {
29     energyLock.lock(); futureLock.lock();
30     if (!hasEnergy() && now != future) {
31        energyLock.unlock(); futureLock.unlock();
32        return false;
33
34     }
35
36     now = future;
37
38     energy = energy - 100;
39
40     System.out.println("Heading to:" + future + " Energy remaining:" + energy);
41     energyLock.unlock(); futureLock.unlock();
42     return true;
43
44  }
```

Does this modified `TimeMachine` class have (circle **all** that apply):

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

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

**This adds several new data races. A data race by definition is a type of race condition. Here are a few of the new data races:**

- **A thread could be in `hasEnergy` reading `energy` at line 11 while another thread is at line 38 in `backToTheFuture` writing `energy`. Similarly a thread could be in `adjustEnergy` reading `energy` at line 17 or 21 while another thread is at line 38 in `backToTheFuture` writing `energy`.**
- **Two threads could also be at line 38 in `backToTheFuture` both writing `energy`, or one reading and one writing `energy` both on line 38. A thread could also be at line 40 in `backToTheFuture` reading `energy`, while another thread is at line 38 in `backToTheFuture` writing `energy`.**
- **Threads could be in `adjustEnergy` writing `energy` while a thread it reading `energy` at line 38 or 40 in `backToTheFuture`.**

- **A thread could be at line 27 in `setFuture` writing `future`, while a thread is at line 30 or line 36 or 40 in `backToTheFuture` reading `future`.**

c) **[8 pts]** Modify the **code above in part b) and on the previous page** to use locks to ***allow the most concurrent access*** and to avoid all of the potential problems listed above. **For full credit you must allow the most concurrent access possible without introducing any errors or extra locks.** Create locks as needed. Use any reasonable names for the locking methods you call. **DO NOT use `synchronized`**. You should create re-entrant lock objects as follows:

```
ReentrantLock lock = new ReentrantLock();
```