Section 09: Concurrency Solutions

0. 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.

1. 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.

2. 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

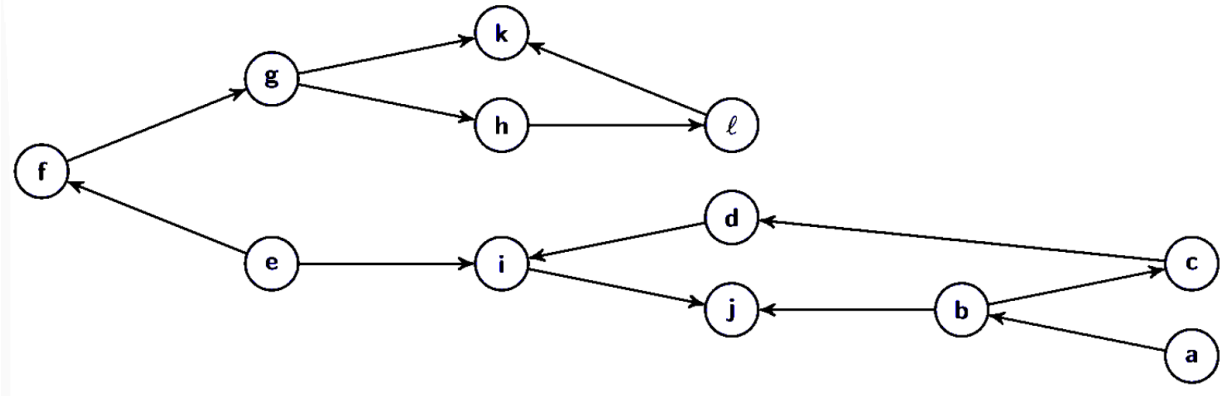
If there are any FIXED problems, describe why they are FIXED. If there are any NEW problems, give an example of when those problems could occur. Be specific!

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.

3. It Rhymes with Flopological Sort

Consider the following graph:



a) Does this graph have a topological sort? Explain why or why not. If you answered that it does not, remove the MINIMUM number of edges from the graph necessary for there to be a topological sort and carefully mark the edge(s) you are removing. Otherwise, just move on to the next part.
Yes it does. This is a DAG (ie. it has no cycles).

For the remaining parts, work with this (potentially) new version of the graph.

b) Find a topological sort of the graph. Do not bother showing intermediary work.
There are many! One example is **e, f, g, h, l, k, a, b, c, d, i, j.**

Snow Day

After 4 snow days last year, UW has decided to improve its snow response plan. Instead of doing "late start" days, they want an "extended passing period" plan. The goal is to clear enough sidewalks that everyone can get from every classroom to every other \textbf{eventually} but not necessarily very quickly.

Unfortunately, UW has access to only one snowplow. Your goal is to determine which sidewalks to plow and whether it can be done in time for the first 8:30 AM lectures.

You have a map of campus, with each sidewalk labeled with the time it will take to plow to clear it.

a) What will the vertices of your graph be?
Have a vertex for each building.

b) What will the edges be? You should at least say whether your edges are directed or not and whether they're weighted or not.
Have an edge for each section of sidewalk. The edges should be undirected, and weighted by the time it will take the snowplow to clear it.

c) What algorithm will you run on your graph?
Run an MST algorithm (either **Kruskal's** or Prim's).

d) How will you interpret the output of your algorithm? (i.e. which sidewalks to plow “in the real world" instead of just in graph terms).
Whatever edges are chosen are the sidewalks the plow should clear.

e) Briefly (2-4 sentences) explain why your model works. You should at least address why you ran the algorithm you did (e.g., why are you looking for a shortest path/MST/topological ordering/etc.) and how you are ensuring your algorithm will be able to produce an "extended passing period" plowing plan.
We want an MST because our goal is to connect everything cheaply (not find the shortest route from A to B). Look at the weight of the MST. That's how long it will take to plow. If the plow can start in time to finish by 8:30, then we can start on time!