1. River Con-current-ency

Batman and Robin share a jig of water. Rather than going to the store and buying a new bottle whenever they run out (so wasteful!), they continually refill the same jug, In addition, they keep track of how many cups are left in the jug on a sticky note so that they don't have to look in the jug before adding water to it or pouring themselves a delicious glass of water.

```
private int cupsOfWater = 0;
private int maximumWater = 8;
private Stack<Water> water;

private int checkWater() {
  return cupsOfWater;
}

public Water getWater() {
  if (checkWater() > 0) {
      return pourWater()
  }
  // cryDeeply()
}

public void addWater(Water e) {
  if (cupsOfWater != maximumWater) {
      water.push(e);
      cupsOfWater++;
  }
}
```

(a) Let's say that each person is a thread. Provide an interleaving of the two people that causes the maximum amount of embarrassment.

Solution:

if (cupsOfWater != maximumWater)	
	if (cupsOfWater != maximumWater)
water.push(e)	
	water.push(e)

(b) Let's say we put a lock around every time cupsOfWater and water is called. Does this make our scenario thread-safe?

Solution:

Nope, you can still have bad interleavings.

(c) Where would you put the lock to prevent concurrency problems?

Solution:

Put a lock around each method because each method will either read or write from the water stack.

(d) Due to budget cuts, we also share one singular cup now, and having learned our lesson from last time, we decide put a new lock every time someone tries to access this cup. What problems could this cause?

Solution:

Deadlock (if we synchronize both the cup and the jug)

Name: ______________________________

7) **[19 points]** The following class implements a dictionary storing (int key, E data) pairs as a "move to front" unsorted linked list. It assumes no duplicate keys will be inserted.

```
public class MoveToFrontList<E> {
     private Node front = null;
     // Remove Node containing key from the list & return data
     // associated with key or null if key not found.
     synchronized E delete(int key){…}
     // Insert (key, data) at the front of the list.
     synchronized void insert(int key, E data){
          front = new Node(key, data, front);
     }
     // Find (key, data) and move to the front of the list.
     // Return data associated with key or null if key not found.
     E findAndMove(int key){
          E tempData = delete(key);
          if (tempData != null) insert(key, tempData);
          return tempData;
     }
```

a) Does the code above have (circle all that apply):

potential for deadlock, a data race, [✓ a race condition,] none of these

b) If possible, show (using code as done in class) an interleaving of two or more threads where a value that ***is*** in the list would not be found. If not possible, explain why not.

```
findAndMove (a) {   <- finds a                    Doesn't find a!
  tempData <= a's Data  //deletes a  ---->  findAndMove(a) {
                                              tempData = null;    a not there!
                                              return null;
                                            }
  insert (a, tempData)
  return tempData
}
```

c) *If we change the* findAndMove *method to be* **synchronized**, now, does the code above have (circle all that apply) (Note: **synchronized** uses re-entrant locks):

potential for deadlock, a data race, a race condition, [✓ none of these]

d) Say we added another find method to this class that only finds values, but does not move them (does not modify the list). (See the code for find on the next page.) If *all other methods* are **synchronized**, but our new find method is not **synchronized**, does the code above plus this new find method have (circle all that apply):

potential for deadlock, [✓ a data race,] [✓ a race condition,] none of these

find reads front while someone else could be writing it