Concurrency

CSE 332 – Section 9

Slides by James Richie Sulaeman

Where to Get Help

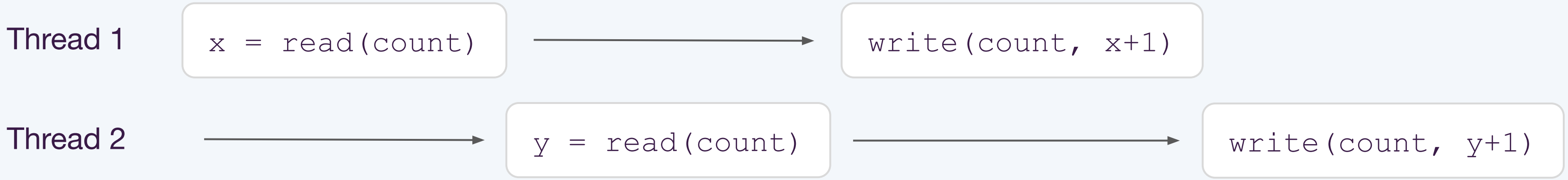
- **Office Hours** → homework and clarification
- **Conceptual office hours** → theory and reasoning
- **Ed board** → ask here for a quick answer from students or staff
- **1-1 meetings** → get individualized help

Concurrency Errors

Concurrency Errors

A **race condition** occurs when the result of your program depends on how threads are scheduled/interleaved

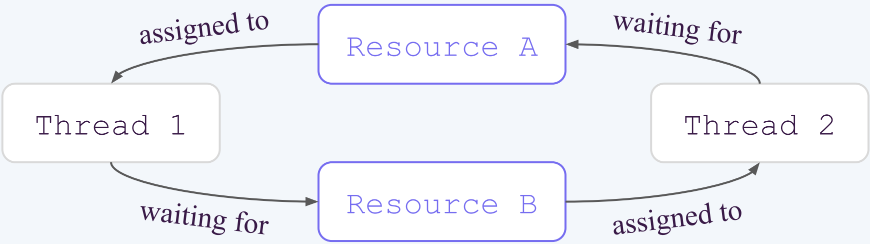
- A **data race** occurs when two threads access the same variable at the same time
  - **Write-write**: two threads writing to the same variable at the same time
  - **Write-read**: one thread writing to a variable while another reads from it
  - Note: read-reads do not cause a data race since they do not modify variables
- A **bad interleaving** occurs when the interleaving of threads result in bad and unexpected intermediate states
  - e.g. two threads are trying to increment the variable `count` at the same time



Concurrency Errors

A **deadlock** occurs when a cycle of threads are waiting on each other

- Thread 1 is waiting on a resource held by Thread 2
- Thread 2 is waiting on a resource held by Thread 1



A piece of code is considered to have a concurrency error **if there exists any execution sequence** that can lead to a race condition or deadlock

- It is not necessary for the code to always execute in this bad sequence
- The possibility of such a sequence occurring is sufficient

Problem 1

(10sp final)

Problem 1a

The constructor has a concurrency error. What is it and how would you fix it?

- There is a data race on `id_counter`
- Two accounts could get the same `id` if they are created at the same time by different threads
- To fix this, you could synchronize on a lock for `id_counter`

```java
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) {
        ...
    }
}
```

Note: the `synchronized` keyword on a method locks `this` object. elsewhere, it locks the specified object

Problem 1b

The `makeFriends` method has a concurrency error. What is it and how would you fix it?

- There is a potential deadlock
- Suppose there are two `UserProfile` objects called `obj1` and `obj2`
    - One thread calls `obj1.makeFriends(obj2)`
    - Another thread calls `obj2.makeFriends(obj1)`
    - Both threads execute line 13 at the same time and deadlock at line 14
- To fix this, acquire locks in a consistent order (e.g. in order of `id` fields)

```
 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             }
19             friends[numFriends++] = newFriend.id;
20             newFriend.friends[newFriend.numFriends++] = id;
21         }
22     }
23
24     synchronized void removeFriend(UserProfile frenemy) {
25         ...
26     }
27 }
```

Note: the `synchronized` keyword on a method locks `this` object. elsewhere, it locks the specified object

Problem 2

(18au final)

Problem 2a

Does the `BubbleTea` class have:

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

- There is the potential for bad interleaving
- Suppose two threads call `addLiquid()` at the same time
  - Both threads satisfy the `hasCapacity()` condition with a value of 7 for `drink.size()`
  - Both threads then push onto the `drink` stack, exceeding `maxDrinkAmount`

```java
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);
        }
    }
}
```

Note: a “thread-safe” stack prevents data races on itself since only one thread can modify it at a time

Problem 2b

Suppose we made the `addTopping` method synchronized. Does this modified `BubbleTea` class have:

a race condition | potential for deadlock

a data race | none of these

- This does not fix the problem
- Modifying `addTopping()` still allows for the same pattern of execution in `addLiquid()` as described earlier
- However, this change reduces the effective concurrency in the code, so it makes things slightly worse

```java
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 synchronized void addTopping(String newTop) {
        if (newTop.equals("Boba") || newTop.equals("Tapioca")) {
             toppings.push("Bubbles");
        } else {
            toppings.push(newTop);
        }
    }
}
```

Note: a “thread-safe” stack prevents data races on itself since only one thread can modify it at a time

Problem 3

(19wi final)

Problem 3a

Does the `PhoneMonitor` class have:

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

- There is a data race on `phoneOn`. By definition, this is also a race condition
- Thread 1 could be at line 11 reading `phoneOn`, while Thread 2 is at line 27 writing `phoneOn`
  - This is a write-read data race

```java
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 }
```

Note: the synchronized keyword is reentrant. The thread holds the lock, not the function call.

Problem 3b

Suppose we made the `checkLimits` method public.
Does this modified `PhoneMonitor` class have:

a race condition | **potential for deadlock**

a data race | none of these

- Same data race on `phoneOn` still exists
- However, there is now also the potential for deadlock
- Suppose two threads call `accessPhone()` and `checkLimits()` at the same time
    - Thread 1 calls `accessPhone()` and acquires `accessesLock`
    - Thread 2 calls `checkLimits()` and acquires `minutesLock`
    - Now Thread 1 wants to acquire `minutesLock`, while Thread 2 wants to acquire `accessesLock`

```java
public class PhoneMonitor {
    private int numMinutes = 0;
    private int numAccesses = 0;
    private int maxMinutes = 200;
    private int maxAccesses = 10;
    private boolean phoneOn = true;
    private Object accessesLock = new Object();
    private Object minutesLock = new Object();

    public void accessPhone(int minutes) {
        if (phoneOn) {
            synchronized (accessesLock) {
                synchronized (minutesLock) {
                    numAccesses++;
                    numMinutes += minutes;
                    checkLimits();
                }
            }
        }
    }

    private void checkLimits() {
        synchronized (minutesLock) {
            synchronized (accessesLock) {
                if (numAccesses >= maxAccesses
                  || numMinutes >= maxMinutes) {
                    phoneOn = false;
                }
            }
        }
    }
}
```

Note: the synchronized keyword is reentrant. The thread holds the lock, not the function call.

Problem 4

(19au final)

Problem 4

Does the TimeMachine class as shown above have (circle all that apply):

a race condition potential for deadlock

a data race none of these

- 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.

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

Problem 4

We now add this method to the TimeMachine class.
Does this modified TimeMachine 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. Refer to line numbers in your explanation. Be specific!

```
28 public boolean backToTheFuture() {
29
30 if (!hasEnergy() && now != future) {
31
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
42 return true;
43
44 }
```

Problem 4

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.

```
28 public boolean backToTheFuture() {
29
30 if (!hasEnergy() && now != future) {
31
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
42 return true;
43
44 }
```

Problem 4

Modify the code to use locks to allow the most concurrent access and to avoid all of the potential problems listed above.

You should create re-entrant lock objects as follows:

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

```
public class TimeMachine {
private int now = 1985;
private int future = 2015;
private int energy = 100;




public boolean hasEnergy() {

return energy >= 100;

}

public void adjustEnergy(int charge) {

if (energy + charge < 0 ) { // energy should never be negative

return;
}
energy = energy + charge;

}

public void setFuture(int newFuture) {

future = newFuture;

}
}
```

Problem 4

```java
public class TimeMachine {
      private int now = 1985;
      private int future = 2015;
      private int energy = 100;

      ReentrantLock energyLock = new ReentrantLock();
      ReentrantLock futureLock = new ReentrantLock();

      public boolean hasEnergy() {
            energyLock.lock();
            return energy >= 100;
            boolean result = energy>=100;
            energyLock.unlock(); return result;
      }

      public void adjustEnergy(int charge) {
            energyLock.lock();
            if (energy + charge < 0 ) {
                  energyLock.unlock();
                  return;
            }
            energy = energy + charge;
            energyLock.unlock();
      }
```

```java
      public void setFuture(int newFuture) {
            futureLock.lock();
            future = newFuture;
            futureLock.unlock();
      }
}
```

```java
      public boolean backToTheFuture() {
            energyLock.lock(); futureLock.lock();
            if (!hasEnergy() && now != future) {
                  energyLock.unlock();
                  futureLock.unlock();
                  return false;

            }

            now = future;

            energy = energy - 100;

            System.out.println("Heading to:" +
                  future + " Energy remaining:" +
                  energy);
            energyLock.unlock();
            futureLock.unlock();
            return true;

      }
```

Thank You!