Java Programming 2
Thread interference and deadlock

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk

“juice” by Lindsey Turner
https://flic.kr/p/4npCkA

Thread interference

Interference happens when two operations running in different threads but on the same data **interleave**

Two operations have multiple steps, and the steps overlap

Seemingly simple statements can translate to multiple steps in the virtual machine:

`i++` turns into:

1. Retrieve the current value of `i`
2. Increment the retrieved value by 1
3. Store the incremented value back into `i`

Example

```
public class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

Thread interference

Possible sequence with two threads both accessing memory:

1. Thread A: Retrieve `i`
2. Thread B: Retrieve `i`.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in `i`; `i` is now 1.
6. Thread B: Store result in `i`; `i` is now -1.

Avoiding interference: impose an ordering

Establish a **happens-before** relationship between two statements

Actions that create **happens-before**:

- Every statement before a **Thread.start()** happens before every statement executed by that thread
- When a thread terminates and causes **Thread.join()** to return, every statement in the terminated thread happens before every statement following the join

```
int counter = 0;

counter++;

System.out.println
      (counter);
```

Synchronized methods

Additional keyword: **synchronized**

Add to method header

Ensures that:

- Two calls to **synchronized** methods **on the same object** cannot interleave
- When a synchronized method exits, it **happens-before** any other **synchronized** method calls **on the same object**

Constructors cannot be synchronized

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Under the hood: **Intrinsic locks**

Every Java object has a lock associated with it

A thread that needs consistent access to an object fields must **acquire** the lock before access, and **release** the lock when it is done

- In between, the thread **owns** the lock – no other thread can acquire it (will block on attempt)
- Note that a thread can access the same lock multiple times (**re-entrant**)

Synchronized methods make implicit use of the lock

More fine-grained option: **synchronized statements**

Synchronized statements example

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 =
      new Object();
    private Object lock2 =
      new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Atomic access

Atomic action

- Effectively happens all at once – cannot stop in the middle

Reads and writes are atomic for most types (except long/double)

Increments like c++ are **not** atomic

Avoids need for synchronized code

Liveness problems

Liveness: concurrent program's ability to execute in a timely fashion

Potential problems:

- **Deadlock**: two or more threads are blocked forever, waiting for each other
- **Starvation:** a thread cannot gain access to a shared resource and is unable to make progress
- **Livelock:** threads too busy responding to each other to make progress

Deadlock is by far the most common problem

Simple deadlock example

```
transferMoney(accountOne,
  accountTwo, amount);
```

```
transferMoney(accountTwo,
  accountOne, amount);
```

```
public void transferMoney(Account fromAccount,
                          Account toAccount,
                          DollarAmount amountToTransfer) {
   synchronized (fromAccount) {
     synchronized (toAccount) {
       if (fromAccount.hasSufficientBalance(amountToTransfer) {
         fromAccount.debit(amountToTransfer);
         toAccount.credit(amountToTransfer);
       }
     }
   }
 }
```

https://www.infoworld.com/article/2075692/avoid-synchronization-deadlocks.html

How to fix it?

Easiest (but not always practical): don’t ever acquire more than one lock at a time

Or: impose a **consistent ordering** on the acquisition of locks

```
public void transferMoney(Account fromAccount,
                           Account toAccount,
                           DollarAmount amountToTransfer) {
   Account firstLock, secondLock;
   if (fromAccount.accountNumber() == toAccount.accountNumber())
     throw new Exception("Cannot transfer from account to itself");
   else if (fromAccount.accountNumber() < toAccount.accountNumber()) {
     firstLock = fromAccount;
     secondLock = toAccount;
   }
   else {
     firstLock = toAccount;
     secondLock = fromAccount;
   }
```

```
synchronized (firstLock) {
     synchronized (secondLock) {
       if (fromAccount.hasSufficientBalance(amountToTransfer) {
         fromAccount.debit(amountToTransfer);
         toAccount.credit(amountToTransfer);
       }
     }
   }
 }
```