CSE 219
COMPUTER SCIENCE III

MULTITHREADING ISSUES

SLIDES COURTESY:

RICHARD MCKENNA

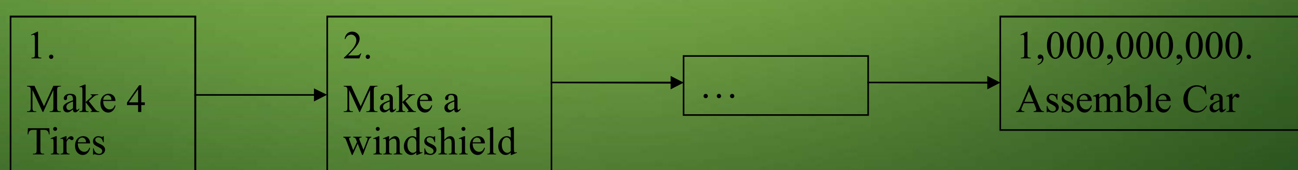
STONY BROOK UNIVERSITY

Multi-threaded Applications

- Can provide performance advantages. Why?
  - minimize IDLE time
  - think Diner Dash

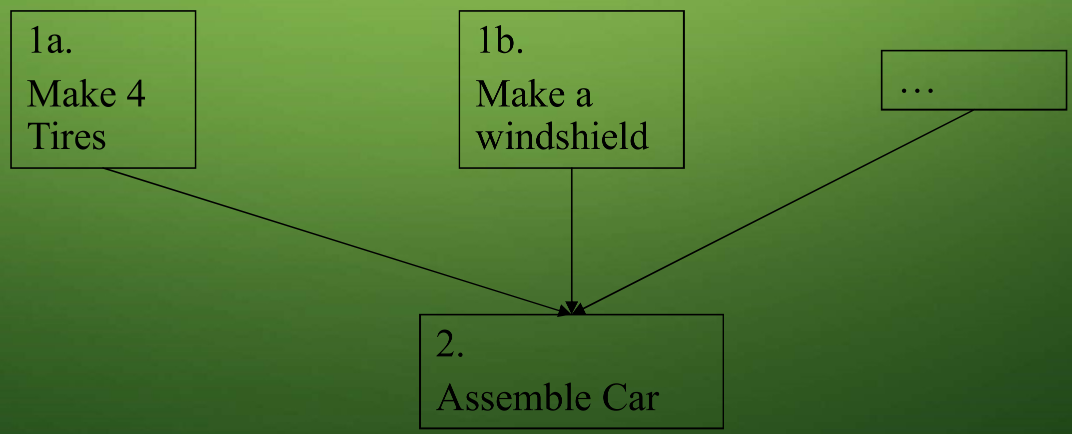
Let's make a CAR

- Sequential Approach:
  - Step 1: make 4 tires
  - Step 2: make a windshield
  - …
  - Step 1,000,000,000: Assemble Car



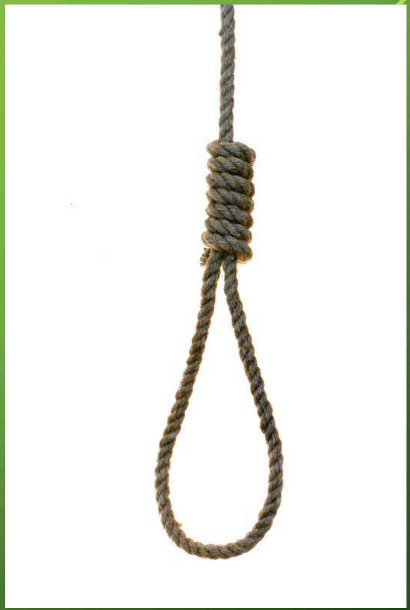
Before the end of eternity, please

- Parallel Approach:
  - Step 1: Simultaneously have different workers & suppliers make tires, windshield, etc.
  - Step 2: Assemble car as parts are available



So what could possibly go wrong?

- Lots:
  - race conditions: A race condition occurs when two threads "race" for access to a resource. For example, you may have an object that's used in two threads. If one thread tries to change a value in the object while another tries to do the same thing, a race condition can occur.
  - Deadlock: A deadlock occurs when two threads lock each other's resources. For example, one thread could lock an employee object and then wait for access to a department object. A second thread could have locked the department object and be waiting for the now locked employee object. The net result is that both threads stop dead in the water.
  - slower software production



- Why
  - threads can interfere with one another
  - threads require complex logic to avoid errors

Threads share data

- How?
  - instance variables, static variables, data structures
- So?
  - Thread A may *corrupt* data Thread B is using

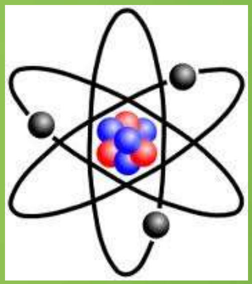
Consumers & Producers

- Some threads are Consumers
  - read shared data
- Some threads are Producers
  - write to shared data
- Some threads are both
  - read and write to shared data
- Danger for a variable when:
  - one thread is a Consumer
  - another thread is a Producer

Race Conditions

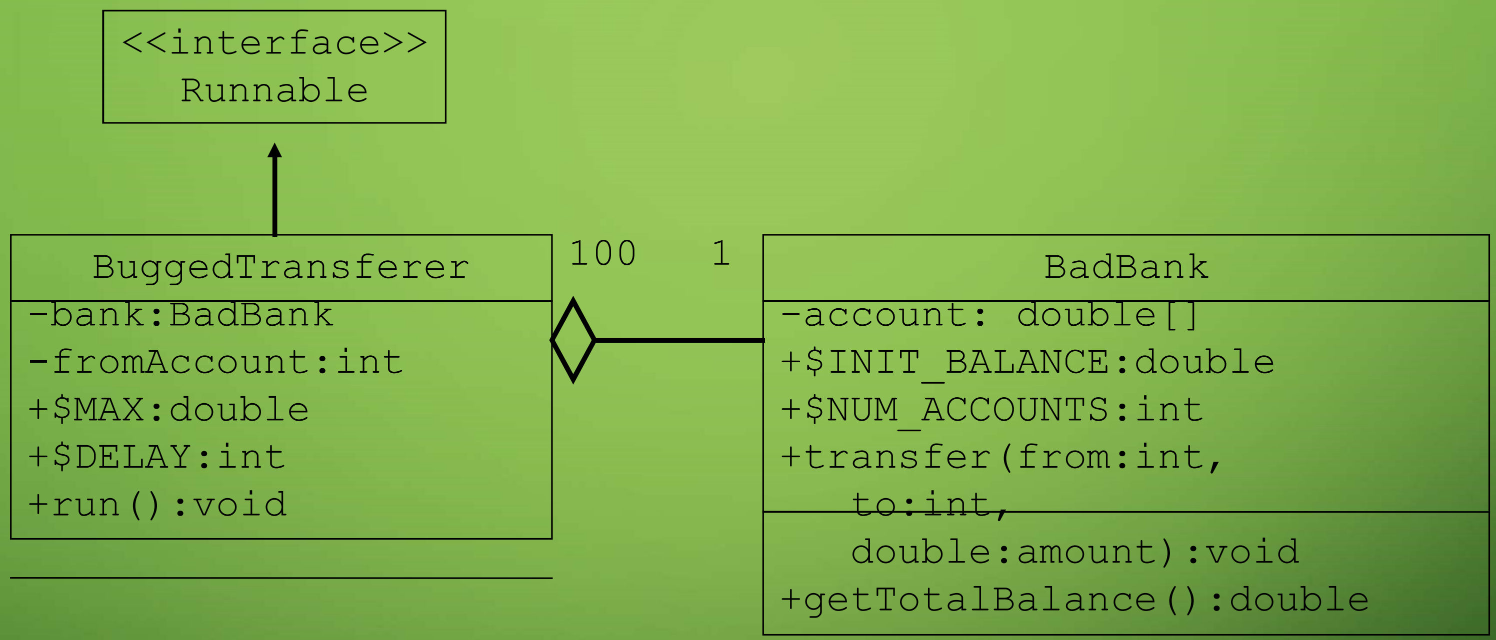
- When one thread corrupts another thread's data
- When do race conditions happen?
  – when transactions lack *atomicity* (Atomic operations in concurrent programming are program operations that run completely independently of any other processes.)
- 2003 Blackout problem? Race Conditions in software:
  – http://www.securityfocus.com/news/8412

Atomicity



- A property of a transaction
- An atomic transaction runs to completion or not at all
- What's a transaction?
  - code execution (ex: method) that changes stored data
- Ever heard of backing out a transaction?
- You are using a TPS this semester. What is it?

Example: A Corruptible Bank



- Assumptions:
  - We will create a single BadBank and make random transfers, each in separate threads

```java
public class BadBank {
 public static int INIT_BALANCE = 1000, NUM_ACCOUNTS = 100;
 private double[] accounts = new double[NUM_ACCOUNTS];

 public BadBank() {
   for (int i = 0; i < NUM_ACCOUNTS; i++)
       accounts[i] = INIT_BALANCE;
 }

 public void transfer(int from, int to, double amount) {
   if (accounts[from] < amount) return;
   accounts[from] -= amount;
   System.out.print(Thread.currentThread());
   System.out.printf("%10.2f from %d to %d",amount,from,to);
   accounts[to] += amount;
   double total = getTotalBalance();
   System.out.printf(" Total Balance: %10.2f%n", total);
 }

 public double getTotalBalance() {
   double sum = 0;
   for (double a : accounts) sum += a;
   return sum;
 }
}
```

BadBank.java

```java
public class BuggedTransferer implements Runnable {
   private BadBank bank;
   private int fromAccount;
   public static final double MAX = 1000;
   public static final int DELAY = 100;

   public BuggedTransferer(BadBank b, int from) {
    bank = b;
    fromAccount = from;
   }

   public void run() {
    try {
      while(true) {
        int toAccount = (int)(bank.NUM_ACCOUNTS * Math.random());
        double amount = MAX * Math.random();
        bank.transfer(fromAccount, toAccount, amount);
        Thread.sleep((int)(DELAY*Math.random()));
      }
    } catch(InterruptedException e) {/*SQUELCH*/}
   }
}
```

BuggedTransferer.java

AtomiclessDriver.java

```
public class AtomiclessDriver {
  public static void main(String[] args) {
    BadBank b = new BadBank();
    for (int i = 0; i < BadBank.NUM_ACCOUNTS; i++) {
      BuggedTransferer bT = new BuggedTransferer(b,i);
      Thread t = new Thread(bT);
      t.start();
    }
  }
}
```

What results might we get?

```
…Total Balance:  100000.00
…Total Balance:  100000.00
…Total Balance:  100000.00
…Total Balance:  100000.00
…Total Balance:  100000.00
…Total Balance:  100000.00
…Total Balance:  100000.00
…Total Balance:  100000.00
…Total Balance:  99431.55
…Total Balance:  99367.34
…
```

- Why might we get invalid balance totals?
  - race conditions
  - operations on shared data may lack *atomicity*

- Bottom line:
  - a method or even a single statement is not an *atomic* operation
  - this means that the statement can be interrupted during its operation

A single statement?

- Compiled into multiple low-level statements
- To see:

javap –c –v BadBank

- Ex: accounts[from] -= amount;

```
…
 21 aload_0
 22 getfield #3 <Field double accounts[]>
 25 iload_1
 26 dup2
 27 daload
 28 dload_3
 29 dsub
 30 dastore
…
```

Race Condition Example

- Thread 1 & 2 are in `transfer` at the same time.

What's the problem?

Thread 1	Thread 2
aload_0	
getfield #3	
iload_1	
dup2	
daload	
dload_3	
	aload_0
	getfield #3
	iload_1
	dup2
	daload
	dload_3
	dsub
	dastore
dsub	
dastore	

This might store corrupted data

How do we guarantee atomicity?

- By locking methods or code blocks

- What is a lock?
  – Locks other threads out

- How do we do it?
  – ReentrantLock - ReentrantLock is mutual exclusive lock, similar to implicit locking provided by synchronized keyword in Java, with extended feature like fairness, which can be used to provide lock to longest waiting thread. Lock is acquired by lock() method and held by Thread until a call to unlock() method.

ReentrantLock

- Basic structure to lock critical code:

```
ReentrantLock myLock = new ReentrantLock();
…
myLock.lock();
try {
// CRITICAL AREA TO BE ATOMICIZED
}
finally {
   myLock.unLock();
}
```

- When a thread enters this code:
  - if no other lock exists, it will execute the critical code
  - otherwise, it will wait until previous locks are unlocked

Updated transfer method

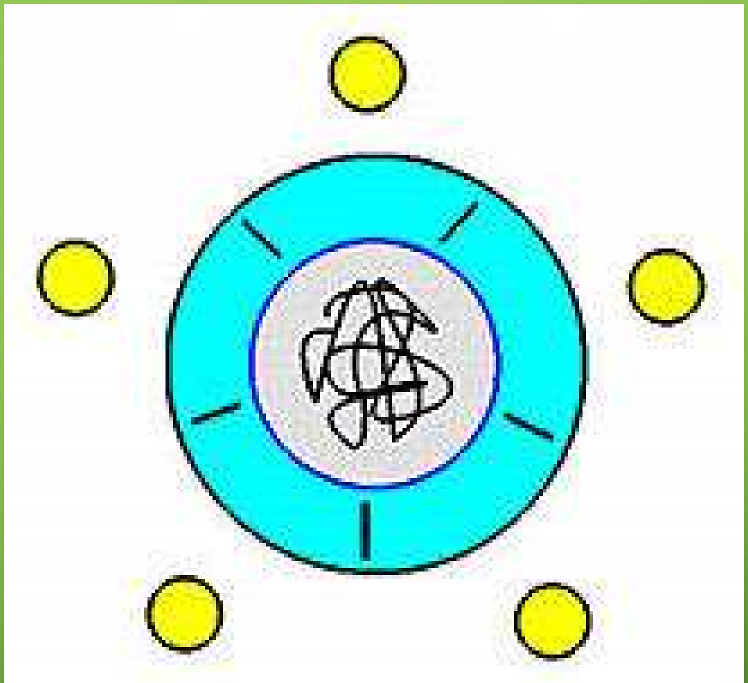
```
public class GoodBank {
 private ReentrantLock bankLock = new ReentrantLock();
 private double[] accounts = new double[NUM_ACCOUNTS];
…
 public void transfer(int from, int to, double amount) {
   bankLock.lock();
   try {
     if (accounts[from] < amount) return;
      accounts[from] -= amount;
      System.out.print(currentThread());
      System.out.printf("%10.2f from %d to%d",amount,from,to);
      accounts[to] += amount;
      double total = getTotalBalance();
      System.out.printf(" Total Balance: %10.2f%n", total);
   } finally{
       bankLock.unlock();
   }
  }
}
```

- NOTE: This works because `transfer` is the only mutator method for `accounts`
  - What if there were more than one?
    - then we'd have to lock the `accounts` object

What’s the worst kind of race condition?

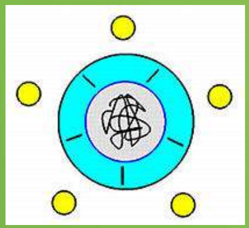
- The devastating ones that rarely occur
  - even during extensive testing
- Can be hard to simulate
  - or deliberately produce
- We don't control the thread scheduler
- Moral: make sure your program is *thread-safe*
  - should be proven logically, before testing

Dining Philosopher's Problem



- 5 philosophers
- 5 chopsticks
- 1 plate of spaghetti

Dining Philosopher's Problem



- Tony Hoare's problem statement is about five philosophers who must alternatively eat and think. All five are sited in a round table with a plate of spaghetti and chopsticks adjacently placed between philosophers.
- A chopstick can only be used by one philosopher at a time. However in order to eat, two chopsticks are required. A philosopher can take an available chopstick but is not allowed to eat unless the philosopher has both of his chopsticks.
- It should be noted that eating is not limited by the possible amount of spaghetti left or stomach space. It is assumed that there is an infinite supply of spaghetti and demand.

Deadlocks

- Deadlock:
  - a thread T_1 holds a lock on L_1 and wants lock L_2 AND
  - a thread T_2 holds a lock on L_2 and wants lock L_1.

- How do we resolve this?

Deadlock Resolution

- Many approaches

- One technique:
  - don’t let waiting threads lock other data
    - release all their locks before making them wait
    - there are all sorts of proper algorithms for thread lock ordering (you’ll see if you take CSE 306)