Multithreading Issues and Parallel Programming

CSE219, Computer Science III

Stony Brook University

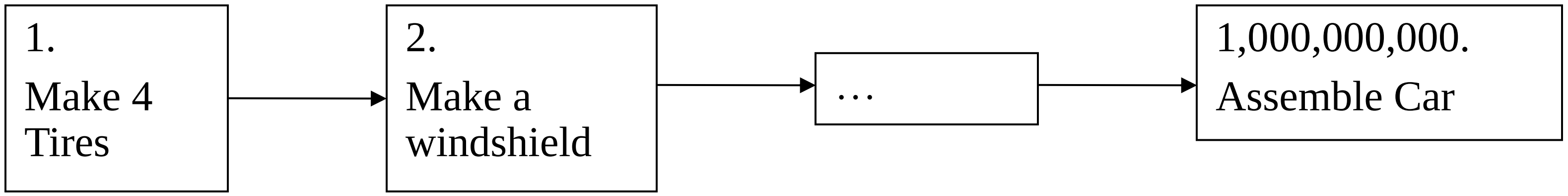
http://www.cs.stonybrook.edu/~cse219

Multi-threaded Applications

- Provide performance advantages:
  - minimize IDLE time,
  - line balancing (think of Diner Dash)

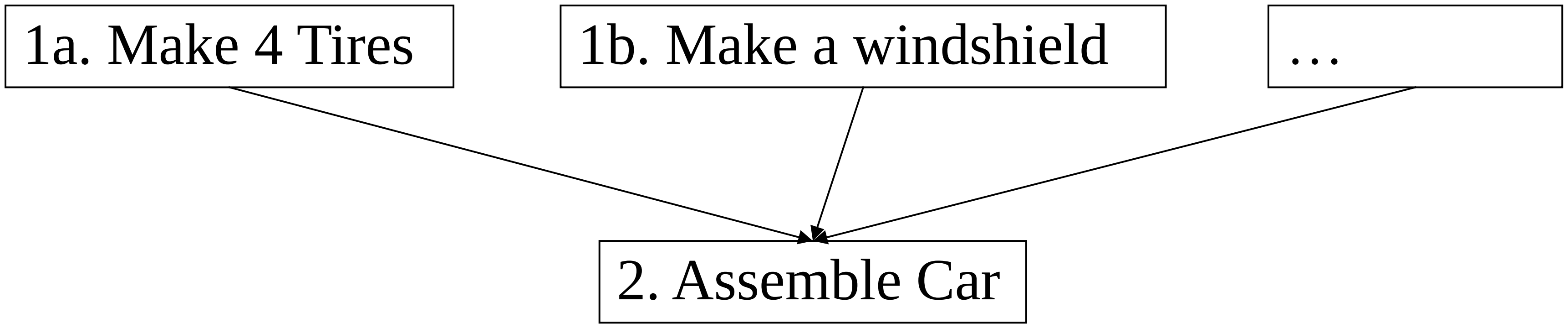
Example: Let's make a CAR

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



Example: Let's make a CAR

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



What could possibly go wrong?

- Lots:
  - **race conditions (shared memory)**
  - **deadlocks**
  - **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 for a data that was not yet produced by another thread that is a Producer

Race Conditions

- When do race conditions happen? What is a race condition?
  - When one thread corrupts another thread's data!
  - When transactions lack *atomicity*!
- Real example: What caused the 2003 Blackout problem?
  - Race Conditions in software:
    http://www.securityfocus.com/news/8412

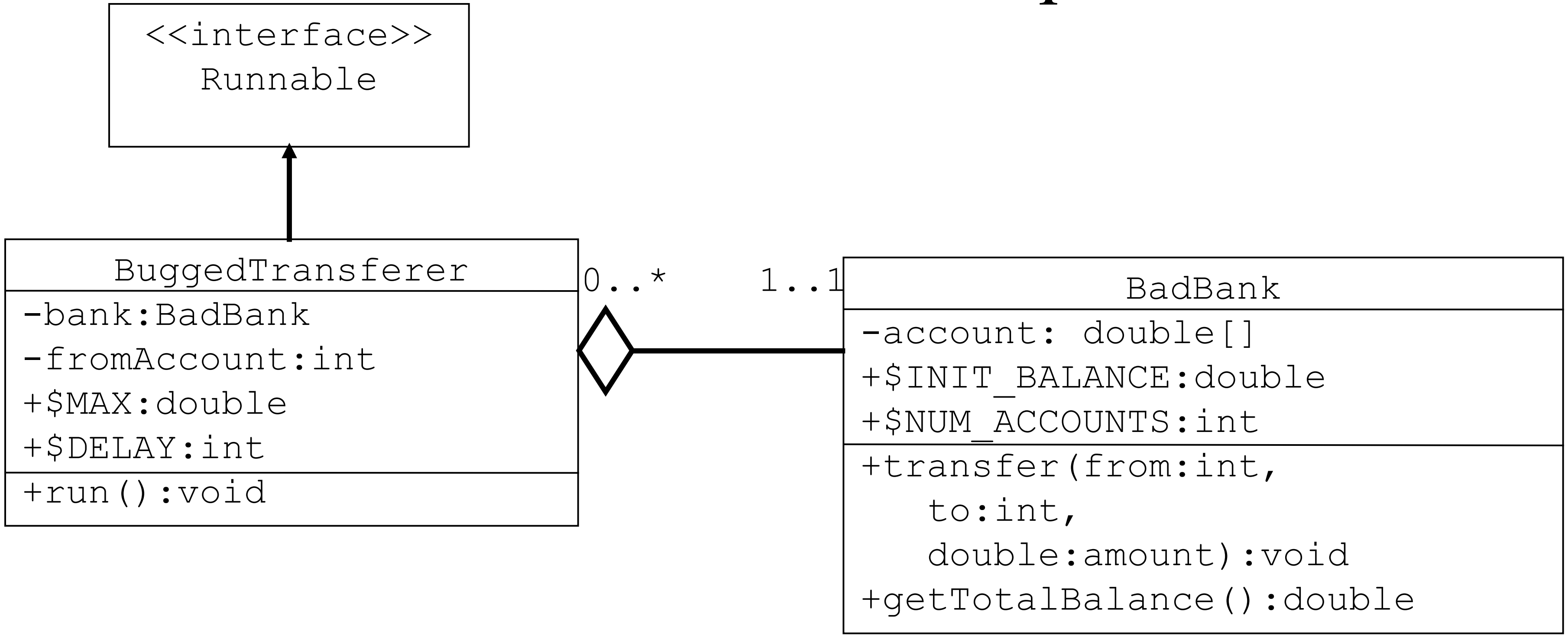
  "About eight weeks after the blackout, the bug was unmasked as a particularly subtle incarnation of a common programming error called a "***race condition***," triggered on August 14th by a perfect storm of events and alarm conditions on the equipment being monitored."

Atomicity

- Atomicity is a property of a transaction
- An atomic transaction runs to completion or not at all
- What's a transaction?
  - execution of a method that changes stored data (databases)
    - A method is a sequence of multiple instructions.
- Ever heard of backing out a transaction?
  - If one cannot execute the entire method, then one should undo all effects of that method = rollback.

Example: A Corruptible Bank

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



```
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

```
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();
        }
    }
}
```

AtomiclessDriver.java

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 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

Is a single statement atomic?

- A single Java statement is compiled into multiple low-level statements.

```
javap –c –v BadBank
```

Note: The ***javap*** command disassembles one or more class files.

- E.g., accounts[from] -= amount; // is compiled into JVM:

```
…
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

Threads 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 $\longrightarrow$

How do we guarantee atomicity?

- By locking methods or code blocks!
- What is a lock?
  - Locks other threads out!
- How do we do it?
  - One Way: `java.util.concurrent.locks.ReentrantLock` class
  - Many other ways.

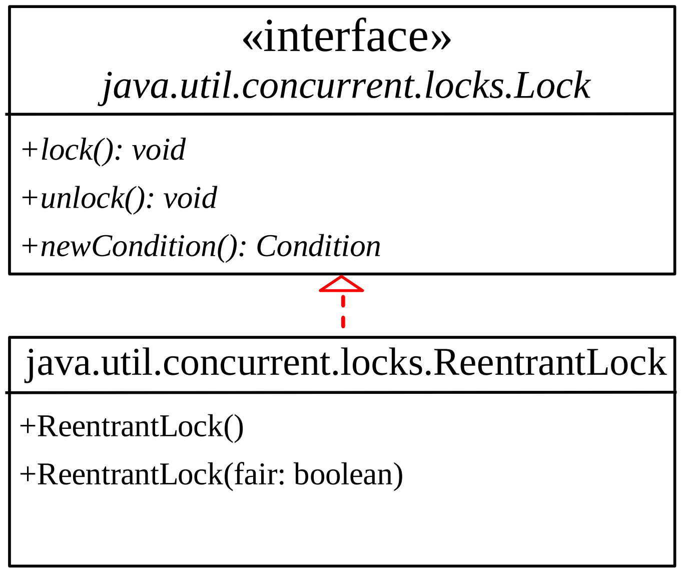
ReentrantLock

- 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.

ReentrantLock



«interface»
java.util.concurrent.locks.Lock
+*lock(): void*
+*unlock(): void*
+*newCondition(): Condition*
java.util.concurrent.locks.ReentrantLock
+ReentrantLock()
+ReentrantLock(fair: boolean)

Acquires the lock.

Releases the lock.

Returns a new Condition instance that is bound to this Lock instance.

Same as ReentrantLock(false).

Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

True fairness policies guarantee the longest-wait thread to obtain the lock first.

False fairness policies grant a lock to a waiting thread without any access order.

Programs using fair locks accessed by many threads may have poor overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Updated transfer method

```
import java.util.concurrent.locks.ReentrantLock;
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(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);
        } 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. */

GoodBank.java

```
import java.util.concurrent.locks.ReentrantLock;
public class GoodBank {
    private ReentrantLock bankLock = new ReentrantLock();
    public static int INIT_BALANCE = 100, NUM_ACCOUNTS = 100;
    private double[] accounts = new double[NUM_ACCOUNTS];
    public GoodBank() {
        for (int i = 0; i < NUM_ACCOUNTS; i++) {
            accounts[i] = INIT_BALANCE;
        }
    }
    public void transfer(int from, int to, double amount) {
        bankLock.lock();
        try {
            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);
        } finally {
            bankLock.unlock();
        }
    }
    public double getTotalBalance() {
        double sum = 0;
        for (double a : accounts) {
            sum += a;
        }
        return sum;
    }
}
```

GoodTransferer.java

```
public class GoodTransferer implements Runnable {
    private GoodBank bank;
    private int fromAccount;
    public static final double MAX = 1000;
    public static final int DELAY = 100;
    public GoodTransferer(GoodBank 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*/

        }
    }
}
```

AtomicDriver.java

```
public class AtomicDriver {

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

}
```

The synchronized keyword

- To avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as the *critical region*.
- The critical region in the bank example is the entire **`transfer`** method.
- You can use the **`synchronized`** keyword to synchronize the method so that only one thread can access the method at a time.

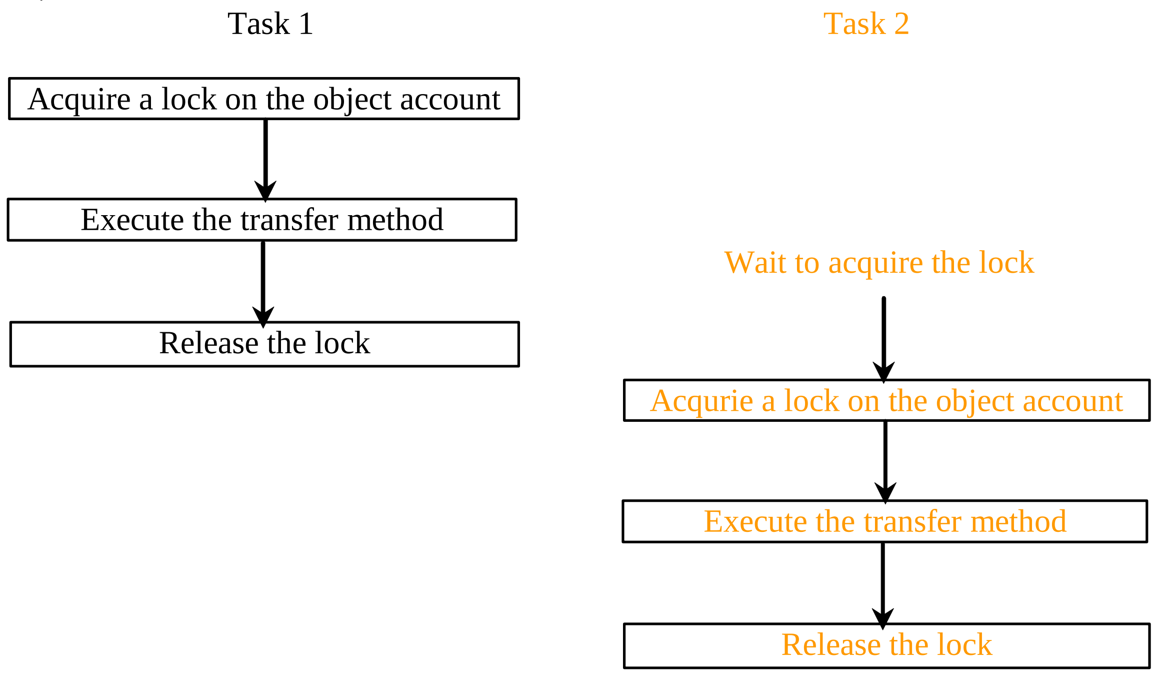
```
public synchronized void transfer(
int from, int to, double amount) { … }
```

Synchronizing Instance Methods and Static Methods

- Internally, a synchronized method acquires a lock before it executes.
  - In the case of an instance method, the lock is on the object for which the method was invoked.
  - In the case of a static method, the lock is on the class.
    - If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released.
    - Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

Synchronizing Instance Methods and Static Methods

- With the transfer method synchronized, the race scenario cannot happen.
  - If Task 2 starts to enter the method, and Task 1 is already in the method, Task 2 is blocked until Task 1 finishes the method.


Task 1
Acquire a lock on the object account
Execute the transfer method
Release the lock
Task 2
Wait to acquire the lock
Acqurie a lock on the object account
Execute the transfer method
Release the lock

Synchronizing Statements

- A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method:

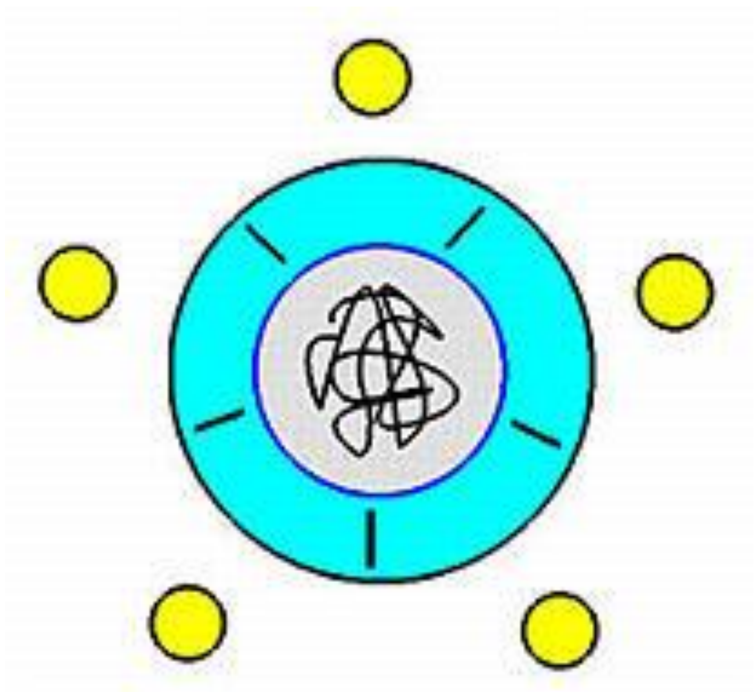
```
synchronized (expr) {
   statements;
}
```

- The expression `expr` must evaluate to an object reference.
  - If the object is already locked by another thread, the thread is blocked until the lock is released.
  - When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

What’s the worst kind of race condition?

- A **deadlock** is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.
- Devastating race conditions that rarely occur
  - even during extensive testing
- Can be hard to simulate
  - or deliberately produce
- Note: We don't control the thread scheduler!
- Moral: don’t rely on thread scheduler, but make sure your program is thread-safe.
  - should be proven logically, before testing

Dining Philosopher's Problem



- 5 philosophers
- 5 forks
- 1 plate of spaghetti

Five silent philosophers sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat.

A philosopher can only eat spaghetti when he has both left and right forks.

A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them.

The possibility of a deadlock: if all five philosophers pick up the left fork at the same time and wait until the right fork is available, then no progress is possible again (starvation).

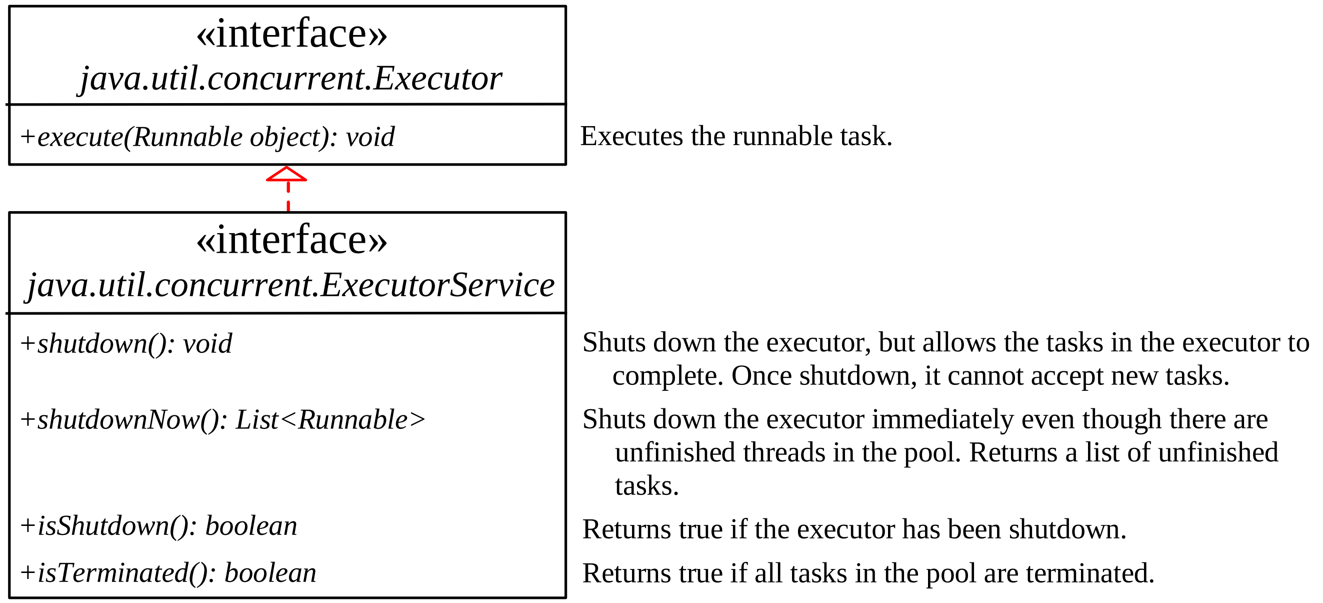
Deadlocks

- Deadlock:
  - a thread T1 holds a lock on L1 and wants lock L2

    AND
  - a thread T2 holds a lock on L2 and wants lock L1.
- How do we resolve this?
  - Deadlock Resolution
    - 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).

Thread Pools

- Starting a new thread for each task could limit throughput and cause poor performance.
- A thread pool is ideal to manage the number of tasks executing concurrently.
  - The Executor interface executes tasks in a thread pool.
  - The ExecutorService interface manages and controls tasks.



«interface»
java.util.concurrent.Executor

+execute(Runnable object): void — Executes the runnable task.

«interface»
java.util.concurrent.ExecutorService

+shutdown(): void — Shuts down the executor, but allows the tasks in the executor to complete. Once shutdown, it cannot accept new tasks.

+shutdownNow(): List<Runnable> — Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

+isShutdown(): boolean — Returns true if the executor has been shutdown.

+isTerminated(): boolean — Returns true if all tasks in the pool are terminated.

Thread Pools

- To create an Executor object, use the static methods in the Executors class.

java.util.concurrent.Executors	
+newFixedThreadPool(numberOfThreads: int): ExecutorService	Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.
+newCachedThreadPool(): ExecutorService	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

ExecutorDemo.java

```
import java.util.concurrent.*;

public class ExecutorDemo {

    public static void main(String[] args) {
        // Create a fixed thread pool with maximum three threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit runnable tasks to the executor
        executor.execute(new PrintChar('a', 100));
        executor.execute(new PrintChar('b', 100));
        executor.execute(new PrintNum(100));

        // Shut down the executor
        executor.shutdown();
    }
}
```

DiningPhilosophersDeadlock.java

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class DiningPhilosophersDeadlock {
    static ReentrantLock[] forkLock = {
        new ReentrantLock(),
        new ReentrantLock(),
        new ReentrantLock(),
        new ReentrantLock(),
        new ReentrantLock()
    };

    public static void main(String[] args) {
        // Create a thread pool with two threads
        ExecutorService executor = Executors.newFixedThreadPool(5);
        executor.execute(new PhilosopherTask(0));
        executor.execute(new PhilosopherTask(1));
        executor.execute(new PhilosopherTask(2));
        executor.execute(new PhilosopherTask(3));
        executor.execute(new PhilosopherTask(4));
        executor.shutdown();
    }

    public static class PhilosopherTask implements Runnable {
        int philosopher, leftFork, rightFork;
        PhilosopherTask(int n) {
            philosopher = n;
            leftFork = n;
            if (philosopher > 0) {
                rightFork = philosopher - 1;
            } else {
                rightFork = 4;
            }
        }
```

DiningPhilosophersDeadlock.java

```
        public void run() {
            try {
                while (true) {
                    // take the left fork
                    forkLock[leftFork].lock();
                    System.out.println("The philosopher " + philosopher
                             + " took his left fork " + leftFork);
                    Thread.sleep(1000);
                    // take the right fork
                    System.out.println("The philosopher " + philosopher
                             + " tries to take his right fork " + rightFork);
                    forkLock[rightFork].lock();
                    System.out.println("The philosopher " + philosopher
                             + " took his right fork " + rightFork);
                    System.out.println("The philosopher " + philosopher + " eats.");
                    System.out.println("The philosopher " + philosopher
                             + " releases his right fork " + rightFork);
                    forkLock[rightFork].unlock();
                    System.out.println("The philosopher " + philosopher
                             + " releases his left fork " + leftFork);
                    forkLock[leftFork].unlock();
                }
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

```
javac DiningPhilosophersDeadlock.java
Java DiningPhilosophersDeadlock

run:
The philosopher 0 took the left fork 0
The philosopher 1 took the left fork 1
The philosopher 2 took the left fork 2
The philosopher 3 took the left fork 3
The philosopher 4 took the left fork 4
The philosopher 0 tries to take the right fork 4
The philosopher 2 tries to take the right fork 1
The philosopher 4 tries to take the right fork 3
The philosopher 3 tries to take the right fork 2
The philosopher 1 tries to take the right fork 0
    NOTHING ELSE
    DEADLOCK
```

DiningPhilosophersResolution.java

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
public class DiningPhilosophersResolution {
    static ReentrantLock[] forkLock = {
        new ReentrantLock(),
        new ReentrantLock(),
        new ReentrantLock(),
        new ReentrantLock(),
        new ReentrantLock()
    };

    public static void main(String[] args) {
        // Create a thread pool with two threads
        ExecutorService executor = Executors.newFixedThreadPool(5);
        executor.execute(new PhilosopherTask(0));
        executor.execute(new PhilosopherTask(1));
        executor.execute(new PhilosopherTask(2));
        executor.execute(new PhilosopherTask(3));
        executor.execute(new PhilosopherTask(4));
        executor.shutdown();
    }

    public static class PhilosopherTask implements Runnable {

        int philosopher, leftFork, rightFork;

        PhilosopherTask(int n) {
            philosopher = n;
            leftFork = n;
            if (philosopher > 0) {
                rightFork = philosopher - 1;
            } else {
                rightFork = 4;
            }
        }
```

DiningPhilosophersResolution.java

```
    public void run() {
        try {
            while (true) {
                // take the left fork
                forkLock[leftFork].lock();
                System.out.println("The philosopher " + philosopher
                        + " took his left fork " + leftFork);
                Thread.sleep(1000);
                // take the right fork
                System.out.println("The philosopher " + philosopher
                        + " tries to take his right fork " + rightFork);
                if (forkLock[rightFork].isLocked()) {
                    // release the left spoon and wait a random time
                    System.out.println("The philosopher " + philosopher
                        + " cannot take the right fork " + rightFork
                        + ", so he releases the left spoon " + leftFork
                        + " and waits a random time");
                    forkLock[leftFork].unlock();
                    Thread.sleep((int) (1000 * Math.random()));
                } else {
                    forkLock[rightFork].lock();
                    System.out.println("The philosopher " + philosopher
                        + " took his right fork " + rightFork);
                    System.out.println("The philosopher " + philosopher + " eats.");
                    System.out.println("The philosopher " + philosopher
                        + " releases his right fork " + rightFork);
                    forkLock[rightFork].unlock();
                    System.out.println("The philosopher " + philosopher
                        + " releases his left fork " + leftFork);
                    forkLock[leftFork].unlock();
                }
            }
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    } }}
```

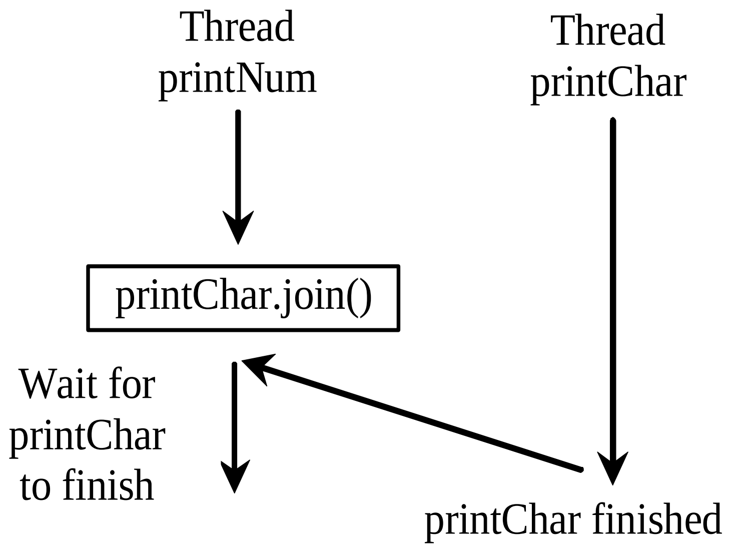
```
javac DiningPhilosophersResolution.java
Java DiningPhilosophersResolution

run:
The philosopher 0 took his left fork 0
The philosopher 2 took his left fork 2
The philosopher 1 took his left fork 1
The philosopher 3 took his left fork 3
The philosopher 4 took his left fork 4
The philosopher 2 tries to take his right fork 1
The philosopher 3 tries to take his right fork 2
The philosopher 0 tries to take his right fork 4
The philosopher 0 cannot take the right fork 4, so he releases the left spoon 0
and waits a random time
The philosopher 1 tries to take his right fork 0
The philosopher 1 took his right fork 0
The philosopher 4 tries to take his right fork 3
The philosopher 1 eats.
The philosopher 3 cannot take the right fork 2, so he releases the left spoon 3
and waits a random time
The philosopher 2 cannot take the right fork 1, so he releases the left spoon 2
and waits a random time
The philosopher 1 releases his right fork 0
The philosopher 1 releases his left fork 1
...
```

Communication between threads

- The `Thread join()`
  - forces one thread to wait for another thread to finish

```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {   }
}
```



TaskThreadDemo.java

```
// The task class for printing number from 1 to n for a given n
class PrintNum implements Runnable {
    private int lastNum;
    public PrintNum(int n) {
        lastNum = n;
    }
    public void run() {
      Thread thread4 = new Thread(new PrintChar('c', 40));
      thread4.start();
      try {
            for (int i = 1; i <= lastNum; i++) {
                  System.out.print(" " + i);
                  if (i == 50) thread4.join();
            }
      } catch (InterruptedException ex) {}
    }
}
```

TaskThreadDemo.java

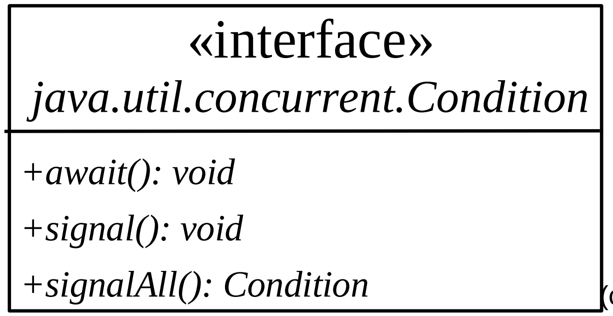
```
public class TaskThreadDemo {
    public static void main(String[] args) {
        // Create tasks
        Runnable printA = new PrintChar('a', 100);
        Runnable printB = new PrintChar('b', 100);
        Runnable print100 = new PrintNum(100);
        // Create threads
        Thread thread1 = new Thread(printA);
        Thread thread2 = new Thread(printB);
        Thread thread3 = new Thread(print100);
        // Start threads
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

TaskThreadDemo.java

```
// The task for printing a specified character in specified times
class PrintChar implements Runnable {
    private char charToPrint; // The character to print
    private int times; // The times to repeat
    /**
     * Construct a task with specified character and number of times
     * to print the character
     */
    public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
    }
    /**
     * Override the run() method to tell the system what the task to perform
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            System.out.print(charToPrint);
        }
    }
}
```

Cooperation Among Threads

- The conditions can be used to facilitate communications among threads: a thread can specify what to do under a certain condition.
  - Conditions are objects created by invoking the new Condition() method on a Lock object.
  - Once a condition is created, you can use its await(), signal(), and signalAll() methods for thread communications.
    - The await() method causes the current thread to wait until the condition is signaled.
    - The signal() method wakes up one waiting thread.
    - The signalAll() method wakes all waiting threads.

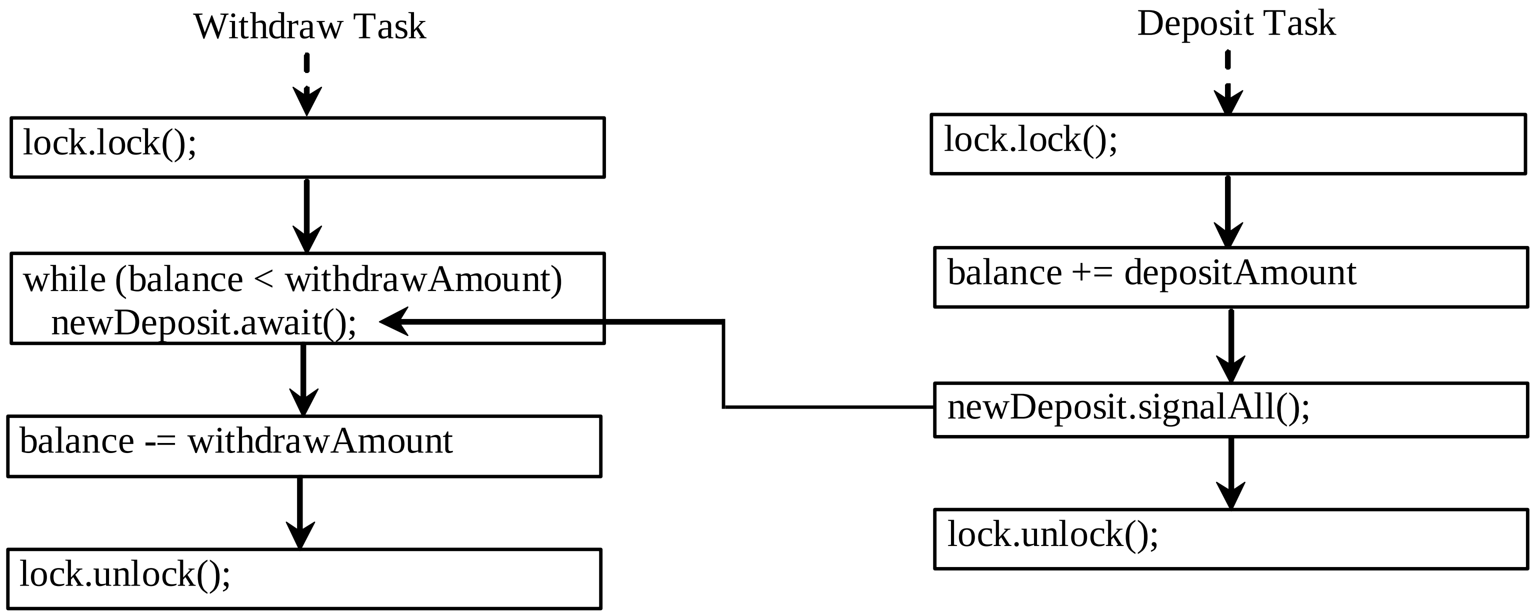


«interface»
java.util.concurrent.Condition

+*await(): void*	Causes the current thread to wait until the condition is signaled.
+*signal(): void*	Wakes up one waiting thread.
+*signalAll(): Condition*	Wakes up all waiting threads.

Cooperation Among Threads

- Example: to synchronize two operations, deposit and withdraw, use a lock with a condition: if the balance is less than the amount to be withdrawn, the withdraw task will wait for the condition newDeposit
  - when the deposit task adds money to the account, the task signals the waiting withdraw task to try again.



ThreadCooperation.java

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
public class ThreadCooperation {
    private static Account account = new Account();
    public static void main(String[] args) {
        // Create a thread pool with two threads
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(new DepositTask());
        executor.execute(new WithdrawTask());
        executor.shutdown();
        System.out.println("Thread 1\t\tThread 2\t\tBalance");
    }
    private static class Account {
        // Create a new lock
        private static Lock lock = new ReentrantLock();
        // Create a condition
        private static Condition newDeposit = lock.newCondition();
        private int balance = 0;

        public int getBalance() {
            return balance;
        }
```

ThreadCooperation.java

```
    public void withdraw(int amount) {
        lock.lock(); // Acquire the lock
        try {
            while (balance < amount) {
                System.out.println("\t\t\tWait for a deposit");
                newDeposit.await();
            }
            balance -= amount;
            System.out.println("\t\t\tWithdraw " + amount
                    + "\t\t" + getBalance());
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        } finally {
            lock.unlock(); // Release the lock
        }
    }
    public void deposit(int amount) {
        lock.lock(); // Acquire the lock
        try {
            balance += amount;
            System.out.println("Deposit " + amount
                    + "\t\t\t\t\t" + getBalance());
            // Signal thread waiting on the condition
            newDeposit.signalAll();
        } finally {
            lock.unlock(); // Release the lock
        }
    }
}
```

ThreadCooperation.java

```
    // A task for adding an amount to the account
    public static class DepositTask implements Runnable {
        public void run() {
            try {
                // Purposely delay it to let the withdraw method proceed
                while (true) {
                    account.deposit((int) (Math.random() * 10) + 1);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }

    // A task for subtracting an amount from the account
    public static class WithdrawTask implements Runnable {
        public void run() {
            while (true) {
                account.withdraw((int) (Math.random() * 10) + 1);
            }
        }
    }
}
```

```
javac ThreadCooperation.java
java ThreadCooperation


Thread 1                 Thread 2                 Balance
Deposit 3                                         3
                         Wait for a deposit
Deposit 4                                         7
                         Wait for a deposit
Deposit 3                                         10
                         Withdraw 9               1
                         Wait for a deposit
Deposit 10                                        11
                         Withdraw 3               8
                         Withdraw 6               2
                         Wait for a deposit
Deposit 6                                         8
                         Withdraw 5               3
                         Wait for a deposit
Deposit 6                                         9
                         Withdraw 4               5
                         Withdraw 4               1
                         Wait for a deposit
Deposit 5                                         6
                         Wait for a deposit
Deposit 8                                         14
                         Withdraw 8               6
                         Wait for a deposit
```

GUIs and Multithreading

- We need to run the code in the GUI *event dispatcher* thread to avoid possible deadlocks.
  - For for quick and simple operations, we can use the static method runLater() in the Platform class to run the code in the event dispatcher thread, i.e., any modifications of the scene graph occur on the FX Application Thread.
  - Example: a background thread which just counts from 0 to 10000 and update progress bar in UI:

```
final ProgressBar bar = new ProgressBar();
new Thread(new Runnable() {
    @Override public void run() {
        for (int i=1; i<=10000; i++) {
            final int counter = i;
            Platform.runLater(new Runnable() {
                @Override public void run() {
                    bar.setProgress(counter/1000000.0);
                }});}}
}).start();
```

GUIs and Multithreading

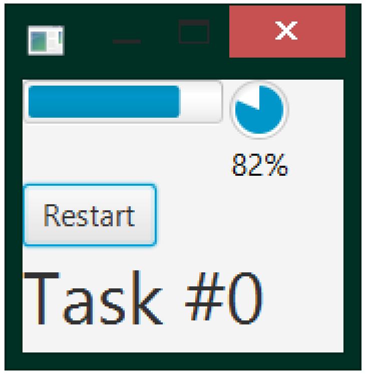
- A better solution is to code using javafx.concurrent.Task<V>:

```
Task<Integer> task = new Task<Integer>() {
       @Override
       protected Integer call() throws Exception {
                int iterations;
                for (iterations = 0; iterations < 10000000;
                       iterations++) {
                    updateProgress(iterations, 10000000);
                }
                return iterations;
            }
        };
ProgressBar bar = new ProgressBar();
bar.progressProperty().bind(task.progressProperty());
new Thread(task).start();
```

- Task implements the Worker interface which is used when you need to run a long task outside the GUI thread (to avoid freezing your application) but still need to interact with the GUI at some stage.

GUIs and Multithreading

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.application.Platform;
import javafx.concurrent.Task;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressBar;
import javafx.scene.control.ProgressIndicator;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
public class ProgressTest extends Application {
    ProgressBar bar;
    ProgressIndicator indicator;
    Button button;
    Label processLabel;
    int numTasks = 0;
    @Override
    public void start(Stage primaryStage) throws Exception {
        VBox box = new VBox();
        HBox toolbar = new HBox();
        bar = new ProgressBar(0);
```

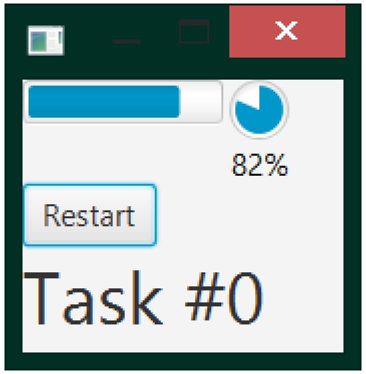


```
indicator = new ProgressIndicator(0);
indicator.setStyle("font-size: 36pt");
toolbar.getChildren().add(bar);
toolbar.getChildren().add(indicator);
button = new Button("Restart");
processLabel = new Label();
processLabel.setFont(new Font("Serif", 36));
box.getChildren().add(toolbar);
box.getChildren().add(button);
box.getChildren().add(processLabel);
Scene scene = new Scene(box);
primaryStage.setScene(scene);
button.setOnAction(e -> {
        Task<Void> task = new Task<Void>() {
            int task = numTasks++;
            double max = 200;
            double perc;
            @Override
            protected Void call() throws Exception {
                for (int i = 0; i < 200; i++) {
                    System.out.println(i);
                    perc = i/max;
                    Platform.runLater(new Runnable() {
                        @Override
                        public void run() {
                            bar.setProgress(perc);
                            indicator.setProgress(perc);
```

```
                                   processLabel.setText("Task #" + task);
                                    }
                                });
                                // SLEEP EACH FRAME
                                try {
                                    Thread.sleep(10);
                                } catch (InterruptedException ie) {
                                    ie.printStackTrace();
                                }
                            }
                            return null;
                        }
                    };
                    // THIS GETS THE THREAD ROLLING
                    Thread thread = new Thread(task);
                    thread.start();
            });
            primaryStage.show();
        }
        public static void main(String[] args) {
            launch(args);
        }
}
// IF YOU CLICK MANY TIMES ON THE BUTTON, THEN ALL THE THREADS WILL
MODIFY THE PROGRESS AT THE SAME TIME - NOT GOOD
```

```
import java.util.concurrent.locks.ReentrantLock;
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import static javafx.application.Application.launch;
import javafx.application.Platform;
import javafx.concurrent.Task;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressBar;
import javafx.scene.control.ProgressIndicator;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
public class BetterProgressTest extends Application {
    ProgressBar bar;
    ProgressIndicator indicator;
    Button button;
    Label processLabel;
    int numTasks = 0;
    ReentrantLock progressLock;
    @Override
    public void start(Stage primaryStage) throws Exception {
        progressLock = new ReentrantLock();
        VBox box = new VBox();
        HBox toolbar = new HBox();
        bar = new ProgressBar(0);
        indicator = new ProgressIndicator(0);
        toolbar.getChildren().add(bar);
        toolbar.getChildren().add(indicator);
        button = new Button("Restart");
        processLabel = new Label();
        processLabel.setFont(new Font("Serif", 36));
        box.getChildren().add(toolbar);
        box.getChildren().add(button);
        box.getChildren().add(processLabel);
        Scene scene = new Scene(box);
        primaryStage.setScene(scene);
        button.setOnAction(e -> {
                Task<Void> task = new Task<Void>() {
                    int task = numTasks++;
                    double max = 200;
                    double perc;
                    @Override
                    protected Void call() throws Exception {
                        try {
                            progressLock.lock();
                        for (int i = 0; i < 200; i++) {
                            System.out.println(i);
                            perc = i/max;
                            Platform.runLater(new Runnable() {
                                @Override
                                public void run() {
```

```
                                bar.setProgress(perc);
                                indicator.setProgress(perc);
                                processLabel.setText("Task #" + task);
                              }
                            });
                            Thread.sleep(10);
                        }}
                        finally {
                                progressLock.unlock();
                                }
                        return null;
                    }
                };
                // THIS GETS THE THREAD ROLLING
                Thread thread = new Thread(task);
                thread.start();
        });
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```



Task #0