Multithreading Issues and Parallel Programming

CSE260, Computer Science B: Honors
Stony Brook University
http://www.cs.stonybrook.edu/~cse260
Multi-threaded Applications

- Provide performance advantages:
  - minimize IDLE time
  - line balancing (think of the game 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

It would take 5 years to make a car sequentially!
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

1a. Make 4 Tires
1b. Make a windshield
...

2. Assemble Car
What could possibly go wrong with parallelism?

- Lots:
  - race conditions (in shared memory)
  - deadlocks (worse type of race conditions)
- Why?
  - threads can interfere with one another
  - threads require complex logic to avoid errors
First: 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!

- Real example: What caused the 2003 Blackout problem?
  - Race Conditions in software:

  “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
- What’s a *transaction*?
  - execution of a method that *changes* stored data
    - In this case, one method is a sequence of *multiple* instructions.
- An *atomic* transaction runs to completion or not at all
- Ever heard of backing out a transaction?
  - If one cannot execute the entire transaction, then one should undo all effects of that transaction, that is, the effects of the data should *roll back*
public class SimpleRace {
    static int n = 10;

    public static void main(String[] args) throws InterruptedException {
        RandomThread thread1 = new RandomThread();
        thread1.start();
        RandomThread thread2 = new RandomThread();
        thread2.start();
    }
}

class RandomThread extends Thread {
    public void run() {
        while (true) {
            int num = (int) (Math.random() * 10);
            Tester.n += num;
            int t = (int) (Math.random() * 10);
            try {
                Thread.sleep(t);
            } catch (InterruptedException e) {}  
            Tester.n -= num;
            System.out.println(Tester.n);
        }
    }
}

Output:
10
3
7
14
...
Example: A Corruptible Bank

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

```java
<<interface>>
Runnable

BuggedTransferer
- bank: BadBank
- fromAccount: int
+ $MAX: double
+ $DELAY: int
+ run(): void

0..* 1..1

BadBank
- account: double[]
+ $INIT_BALANCE: double
+ $NUM_ACCOUNTS: int
+ transfer(from: int, to: int, double: amount): void
+ getTotalBalance(): double
```
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;
        }
        double fromVal = accounts[from];
        double toVal = accounts[to];
        System.out.print(Thread.currentThread());
        System.out.printf("%10.2f from %d to %d", amount, from, to);
        accounts[from] = fromVal - amount;
        accounts[to] = toVal + 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;
    }
}
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*/
        }
    }
}
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?
What results might we get?

- 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?

- No, a single Java statement is compiled into multiple low-level statements — we can see that with:

  javap -c -v BadBank

  Example:
  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

Note: The javap command disassembles one or more class files.
# Race Condition Example

Threads 1 & 2 are in **transfer** at the same time.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>aload_0</td>
<td></td>
</tr>
<tr>
<td>getfield #3</td>
<td></td>
</tr>
<tr>
<td>iload_1</td>
<td></td>
</tr>
<tr>
<td>dup2</td>
<td></td>
</tr>
<tr>
<td>daload</td>
<td></td>
</tr>
<tr>
<td>dload_3</td>
<td></td>
</tr>
<tr>
<td>dsub</td>
<td></td>
</tr>
<tr>
<td>dastore</td>
<td></td>
</tr>
<tr>
<td>dsub</td>
<td></td>
</tr>
<tr>
<td>dastore</td>
<td></td>
</tr>
</tbody>
</table>

What’s the problem?

This might store corrupted data
How do we guarantee atomicity?

- By **locking** methods or code blocks!
  - What is a **lock**?
    - Locks other threads out!
  - To avoid race conditions, more than one thread must be prevented from simultaneously entering a certain part of the program, known as the **critical region**.
    - The critical region in the bank example is the entire **transfer** method.
- How do we do it?
  - One Way is the class
    
    `java.util.concurrent.locks.ReentrantLock`

(c) Paul Fodor & Pearson Inc.
ReentrantLock

- Basic structure to lock *critical code*:
  
  ```java
  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

<table>
<thead>
<tr>
<th>«interface»</th>
<th>java.util.concurrent.locks.Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>+lock(): void</td>
<td>Acquires the lock.</td>
</tr>
<tr>
<td>+unlock(): void</td>
<td>Releases the lock.</td>
</tr>
<tr>
<td>+newCondition(): Condition</td>
<td>Returns a new Condition instance that is bound to this Lock instance.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>java.util.concurrent.locks.ReentrantLock</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ReentrantLock()</td>
</tr>
<tr>
<td>+ReentrantLock(fair: boolean)</td>
</tr>
</tbody>
</table>

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 true 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.
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.println(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. */
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;
    }
}
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*/
        }
    }
}
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();
        }
    }
}

...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: 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: 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: 100000.00
...Total Balance: 100000.00
...Total Balance: 100000.00
...Total Balance: 100000.00
...Total Balance: 100000.00
...Total Balance: 100000.00
...
The synchronized keyword

- You can use the `synchronized` keyword to synchronize the method so that only one thread can access the method at a time:

```java
public synchronized void transfer(int from, int to, double amount) { ...
```
Synchronizing Instance Methods

• Internally, a **synchronized** method acquires a lock before it executes
  • In the case of an instance method, the lock is on the **entire object for which the method was invoked**.
    • If one thread invokes a **synchronized** instance method on an object, the lock of that object is acquired first, then the method is executed, and finally the lock is released.
    • Another thread invoking any **synchronized** method of that object is blocked until the lock is released.
public class Tester2 {
    static int n = 10;
    public static void main(String[] args) throws InterruptedException {
        Tester2 t = new Tester2();
        RandomThread1 thread1 = new RandomThread1(t);
        thread1.start();
        RandomThread1 thread2 = new RandomThread1(t);
        thread2.start();
    }
    public synchronized void access() {
        int num = (int) (Math.random() * 10);
        Tester2.n += num;
        int t = (int) (Math.random() * 10);
        try {
            Thread.sleep(t);
        } catch (InterruptedException e) {}
        Tester2.n -= num;
        System.out.println(Tester2.n);
    }
    public synchronized void access2() {
        int num = (int) (Math.random() * 10);
        Tester2.n += num;
        int t = (int) (Math.random() * 10);
        try {
            Thread.sleep(t);
        } catch (InterruptedException e) {}
        Tester2.n -= num;
        System.out.println(Tester2.n);
    }
}
class RandomThread1 extends Thread {
    Tester2 t;
    public RandomThread1(Tester2 o) {
        t = o;
    }
    public void run() {
        while (true) {
            t.access();
            t.access2();
        }
    }
}

Output:
10
10
10
10
10
10
...

Synchronizing Static Methods

• Internally, a *synchronized* method acquires a lock before it executes

  • In the case of a *static* method, the lock is on all the *static synchronized* methods of the class

  • If one thread invokes a *static synchronized* method, the lock of that class is acquired first, then the method is executed, and finally the lock is released.

  • Another thread invoking a *static synchronized* method of that class is blocked until the lock is released.

  • All methods lock the same monitor. Therefore, you can't simultaneously execute them on the same object from different threads (one of the two methods will block until the other is finished).
public class Tester2 {
    static int n = 10;
    public static void main(String[] args) throws InterruptedException {
        RandomThread2 thread1 = new RandomThread2();
        thread1.start();
        RandomThread2 thread2 = new RandomThread2();
        thread2.start();
    }
}

class RandomThread2 extends Thread {
    public void run() {
        while (true) {
            access();
        }
    }

    public static synchronized void access() {
        int num = (int) (Math.random() * 10);
        Tester2.n += num;
        int t = (int) (Math.random() * 10);
        try {
            Thread.sleep(t);
        } catch (InterruptedException e) {} 
        Tester2.n -= num;
        System.out.println(Tester2.n);
    }
}

Output:
10
10
10
10
...
public class Tester2 {
    static int n = 10;
    public static void main(String[] args) throws InterruptedException {
        RandomThread2 thread1 = new RandomThread2();
        thread1.start();
        RandomThread2 thread2 = new RandomThread2();
        thread2.start();
    }
}

class RandomThread2 extends Thread {
    public void run() {
        while (true) {
            access();
            access2();
        }
    }

    public static synchronized void access() {
        int num = (int) (Math.random() * 10);
        Tester2.n += num;
        int t = (int) (Math.random() * 10);
        try {
            Thread.sleep(t);
        } catch (InterruptedException e) {} 
        Tester2.n -= num;
        System.out.println(Tester2.n);
    }
}
public static synchronized void access2() {
    int num = (int) (Math.random() * 10);
    Tester2.n += num;
    int t = (int) (Math.random() * 10);
    try {
        Thread.sleep(t);
    } catch (InterruptedException e) {} 
    Tester2.n -= num;
    System.out.println(Tester2.n);
}

Output:
10
10
10
10
10
...

(c) Paul Fodor & Pearson Inc.
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.
Synchronizing Statements

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

  ```java
  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.

- Such devastating race conditions are hard to detect
  - 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.
    - Elimination of deadlocks should be proven logically, before testing
Dining Philosopher’s Problem

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).

- 5 philosophers
- 5 forks
- 1 plate of spaghetti
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
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
+isTerminated(): boolean

Shuts down the executor, but allows the tasks in the executor to complete. Once shutdown, it cannot accept new tasks.
Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.
Returns true if the executor has been shutdown.
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.

<table>
<thead>
<tr>
<th>java.util.concurrent.Executors</th>
<th>+newFixedThreadPool(numberOfThreads: int): ExecutorService</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.</td>
</tr>
</tbody>
</table>
import java.util.concurrent.*;

public class ExecutorDemo {
    public static void main(String[] args) {
        // Create a fixed thread pool with maximum 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // 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();
    }
}
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;
            }
        }
    }
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 Dining Philosophers Deadlock.java
Java Dining Philosophers Deadlock

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 FOREVER
BECAUSE IT IS A DEADLOCK (ALL WAIT to INFINITE)
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;
            }
        }
    }
}
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

```java
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) { }
}
```

- **Thread printNum**
- **Thread printChar**
- printChar.join() → Wait for printChar to finish → printChar finished
// 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();
            }
        }
    }
}


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();
    }
}
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.
```
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.

```java
while (balance < withdrawAmount)
   newDeposit.await();

balance -= withdrawAmount
lock.unlock();
```

```java
lock.lock();

balance += depositAmount
newDeposit.signalAll();
lock.unlock();
```
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\tt\tThread 2\tt\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;
        }
    }
}
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" + 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" + getBalance());
        // Signal thread waiting on the condition
        newDeposit.signalAll();
    } finally {
        lock.unlock(); // Release the lock
    }
}
// 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);
        }
    }
}
```markdown
<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deposit 3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Wait for a deposit</td>
<td></td>
</tr>
<tr>
<td>Deposit 4</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Wait for a deposit</td>
<td></td>
</tr>
<tr>
<td>Deposit 3</td>
<td>Withdraw 9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Wait for a deposit</td>
<td></td>
</tr>
<tr>
<td>Deposit 10</td>
<td>Withdraw 3</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Withdraw 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait for a deposit</td>
<td></td>
</tr>
<tr>
<td>Deposit 6</td>
<td>Withdraw 5</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Wait for a deposit</td>
<td></td>
</tr>
<tr>
<td>Deposit 6</td>
<td>Withdraw 4</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Withdraw 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait for a deposit</td>
<td></td>
</tr>
<tr>
<td>Deposit 5</td>
<td>Wait for a deposit</td>
<td>6</td>
</tr>
<tr>
<td>Deposit 8</td>
<td>Wait for a deposit</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>Withdraw 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait for a deposit</td>
<td></td>
</tr>
</tbody>
</table>
```
We need to run the code in the GUI event dispatcher thread to avoid possible deadlocks.

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:

```java
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();
```
Another solution is to code using `javafx.concurrent.Task<V>`:

```java
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.
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;

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() {
                            // Code...
                        }
                    });
                }
            } finally {
                progressLock.unlock();
            }
        }
    }
    task.call();
});
bar.setProgress(perc);
indicator.setProgress(perc);
processLabel.setText("Task #" + task);
}
}
} 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);
}
}
Java’s Built-in Monitors

- Locks and conditions are new in Java 5.
- Prior to Java 5, thread communications are programmed using object’s built-in monitors.
- If you work with legacy Java code, you may encounter the Java’s built-in monitor.
- A monitor is an object with mutual exclusion and synchronization capabilities.
- Only one thread can execute a method at a time in the monitor.
- A thread enters the monitor by acquiring a lock on the monitor and exits by releasing the lock.
- Any object can be a monitor.
- An object becomes a monitor once a thread locks it.
- Locking is implemented using the synchronized keyword on a method or a block.
- A thread must acquire a lock before executing a synchronized method or block.
- A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor.
wait(), notify(), and notifyAll()

Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.

The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.

The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.
The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an IllegalMonitorStateException will occur.

When wait() is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.

The wait(), notify(), and notifyAll() methods on an object are analogous to the await(), signal(), and signalAll() methods on a condition.
Case Study: Producer/Consumer (Optional)

Consider the classic Consumer/Producer example. Suppose you use a buffer to store integers. The buffer size is limited. The buffer provides the method `write(int)` to add an int value to the buffer and the method `read()` to read and delete an int value from the buffer. To synchronize the operations, use a lock with two conditions: `notEmpty` (i.e., buffer is not empty) and `notFull` (i.e., buffer is not full). When a task adds an int to the buffer, if the buffer is full, the task will wait for the `notFull` condition. When a task deletes an int from the buffer, if the buffer is empty, the task will wait for the `notEmpty` condition.
Listing 30.8 presents the complete program. The program contains the Buffer class (lines 43-89) and two tasks for repeatedly producing and consuming numbers to and from the buffer (lines 15-41). The write(int) method (line 58) adds an integer to the buffer. The read() method (line 75) deletes and returns an integer from the buffer.

For simplicity, the buffer is implemented using a linked list (lines 48-49). Two conditions notEmpty and notFull on the lock are created in lines 55-56. The conditions are bound to a lock. A lock must be acquired before a condition can be applied. If you use the wait() and notify() methods to rewrite this example, you have to designate two objects as monitors.
§22.8 introduced queues and priority queues. A *blocking queue* causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue.

```
<interface>
java.util.Collection<E>
</interface>

<interface>
java.util.Queue<E>
</interface>

<interface>
java.util.concurrent.BlockingQueue<E>
</interface>
```

- `+put(element: E): void` - Inserts an element to the tail of the queue. Waits if the queue is full.
- `+take(): E` - Retrieves and removes the head of this queue. Waits if the queue is empty.
Concrete Blocking Queues

Three concrete blocking queues ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue are supported in JDK 1.5. All are in the java.util.concurrent package. ArrayBlockingQueue implements a blocking queue using an array. You have to specify a capacity or an optional fairness to construct an ArrayBlockingQueue. LinkedBlockingQueue implements a blocking queue using a linked list. You may create an unbounded or bounded LinkedBlockingQueue. PriorityBlockingQueue is a priority queue. You may create an unbounded or bounded priority queue.
The program gives an example of using an ArrayBlockingQueue for the Consumer/Producer problem.
Semaphores (Optional)

Semaphores can be used to restrict the number of threads that access a shared resource. Before accessing the resource, a thread must acquire a permit from the semaphore. After finishing with the resource, the thread must return the permit back to the semaphore.

```
A thread accessing a shared resource.

Acquire a permit from a semaphore.
Wait if the permit is not available.

semaphore.acquire();

Access the resource

Release the permit to the semaphore.

semaphore.release();
```
Creating Semaphores

To create a semaphore, you have to specify the number of permits with an optional fairness policy. A task acquires a permit by invoking the semaphore’s acquire() method and releases the permit by invoking the semaphore’s release() method. Once a permit is acquired, the total number of available permits in a semaphore is reduced by 1. Once a permit is released, the total number of available permits in a semaphore is increased by 1.

```java
java.util.concurrent.Semaphore

+ Semaphore(numberOfPermits: int)  
  Creates a semaphore with the specified number of permits. The fairness policy is false.

+ Semaphore(numberOfPermits: int, fair: boolean)  
  Creates a semaphore with the specified number of permits and the fairness policy.

+ acquire(): void  
  Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.

+ release(): void  
  Releases a permit back to the semaphore.
```
Deadlock

Sometimes two or more threads need to acquire the locks on several shared objects. This could cause *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object. Consider the scenario with two threads and two objects. Thread 1 acquired a lock on object1 and Thread 2 acquired a lock on object2. Now Thread 1 is waiting for the lock on object2 and Thread 2 for the lock on object1. The two threads wait for each other to release the in order to get the lock, and neither can continue to run.

```
Thread 1
synchronized (object1) {
    // do something here
    synchronized (object2) {
        // do something here
    }
}

Thread 2
synchronized (object2) {
    // do something here
    synchronized (object1) {
        // do something here
    }
}
```

Wait for Thread 2 to release the lock on object2

Wait for Thread 1 to release the lock on object1
Preventing Deadlock

Deadlock can be easily avoided by using a simple technique known as resource ordering. With this technique, you assign an order on all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order. For the example, suppose the objects are ordered as object1 and object2. Using the resource ordering technique, Thread 2 must acquire a lock on object1 first, then on object2. Once Thread 1 acquired a lock on object1, Thread 2 has to wait for a lock on object1. So Thread 1 will be able to acquire a lock on object2 and no deadlock would occur.
Thread States

A thread can be in one of five states: New, Ready, Running, Blocked, or Finished.
Synchronized Collections

The classes in the Java Collections Framework are not thread-safe, i.e., the contents may be corrupted if they are accessed and updated concurrently by multiple threads. You can protect the data in a collection by locking the collection or using synchronized collections.

The Collections class provides six static methods for wrapping a collection into a synchronized version. The collections created using these methods are called *synchronization wrappers*.

```java
java.util.Collections
+synchronizedCollection(c: Collection): Collection
+synchronizedList(list: List): List
+synchronizedMap(m: Map): Map
+synchronizedSet(s: Set): Set
+synchronizedSortedMap(s: SortedMap): SortedMap
+synchronizedSortedSet(s: SortedSet): SortedSet
```

- Returns a synchronized collection.
- Returns a synchronized list from the specified list.
- Returns a synchronized map from the specified map.
- Returns a synchronized set from the specified set.
- Returns a synchronized sorted map from the specified sorted map.
- Returns a synchronized sorted set.
The widespread use of multicore systems has created a revolution in software. In order to benefit from multiple processors, software needs to run in parallel.

JDK 7 introduces the new Fork/Join Framework for parallel programming, which utilizes the multicore processors.
The Fork/Join Framework is used for parallel programming in Java.

In JDK 7’s Fork/Join Framework, a *fork* can be viewed as an independent task that runs on a thread.
ForkJoinTask and ForkJoinPool

The framework defines a task using the `ForkJoinTask` class, and executes a task in an instance of `ForkJoinPool`.

```java
// interface
java.util.concurrent.ExecutorService

// class
java.util.concurrent.ForkJoinPool

+ForkJoinPool()
+ForkJoinPool(parallelism: int)
+invoke(ForkJoinTask<T>): T

See Figure 30.7

Creates a `ForkJoinPool` with all available processors.
Creates a `ForkJoinPool` with the specified number of processors.
Performs the task and returns its result upon completion.
ForkJoinTask

```java
interface java.util.concurrent.Future<V>
+
cancel(interrupt: boolean): boolean
+get(): V
+isDone(): boolean
```

Attempts to cancel this task.
Waits if needed for the computation to complete and returns the result.
Returns true if this task is completed.

```java
class java.util.concurrent.ForkJoinTask<V>
+
adapt(Runnable task): ForkJoinTask<V>
+fork(): ForkJoinTask<V>
+join(): V
+invoke(): V
+invokeAll(tasks ForkJoinTask<?...): void
```

Returns a ForkJoinTask from a runnable task.
Arranges asynchronous execution of the task.
Returns the result of computations when it is done.
Performs the task and awaits for its completion, and returns its result.
Forks the given tasks and returns when all tasks are completed.

```java
class java.util.concurrent.RecursiveAction<V>
#compute(): void
```

Defines how task is performed.

```java
class java.util.concurrent.RecursiveTask<V>
#compute(): V
```

Defines how task is performed. Return the value after the task is completed.
Examples
Invoking synchronizedCollection(Collection c) returns a new Collection object, in which all the methods that access and update the original collection c are synchronized. These methods are implemented using the synchronized keyword. For example, the add method is implemented like this:

```java
public boolean add(E o) {
    synchronized (this) { return c.add(o); }
}
```

The synchronized collections can be safely accessed and modified by multiple threads concurrently.

The methods in java.util.Vector, java.util.Stack, and Hashtable are already synchronized. These are old classes introduced in JDK 1.0. In JDK 1.5, you should use java.util.ArrayList to replace Vector, java.util.LinkedList to replace Stack, and java.util.Map to replace Hashtable. If synchronization is needed, use a synchronization wrapper.
The synchronization wrapper classes are thread-safe, but the iterator is *fail-fast*. This means that if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator will immediately fail by throwing java.util.ConcurrentModificationException, which is a subclass of RuntimeException. To avoid this error, you need to create a synchronized collection object and acquire a lock on the object when traversing it. For example, suppose you want to traverse a set, you have to write the code like this:

```java
Set hashSet = Collections.synchronizedSet(new HashSet());
synchronized (hashSet) { // Must synchronize it
    Iterator iterator = hashSet.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

Failure to do so may result in nondeterministic behavior, such as ConcurrentModificationException.