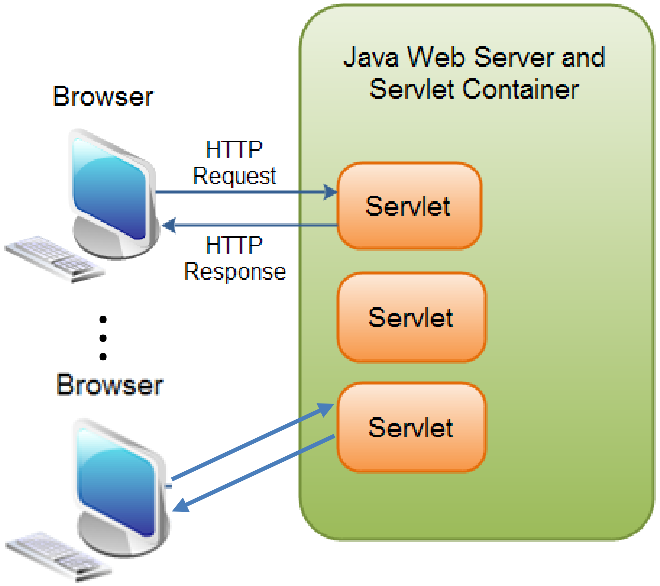
Concurrency

Introduction

- Up to this point, you've been learning about **sequential programming**
  - Everything in a program happens one step at a time
  - A large subset of programming problems can be solved using sequential programming
- For some problems, it becomes convenient or even essential to execute several parts of a program in parallel
  - Web Servlet



The many faces of concurrency

- The problems that you solve with concurrency can be roughly classified as "speed" and "design manageability"
- Faster execution
  - If you want a program to run faster, break it into pieces and run each piece on a separate processor
- Improving code design
  - The design of your program can be greatly simplified
  - Some types of problems, such as simulation, are difficult to solve without support for concurrency
  - Simulations generally involve many interacting elements, each with "a mind of its own"

The many faces of concurrency (Cont.)

- **AIIDE StarCraft AI Competition**
  - ***University of Alberta* Department of Computing Science**

The many faces of concurrency (Cont.)

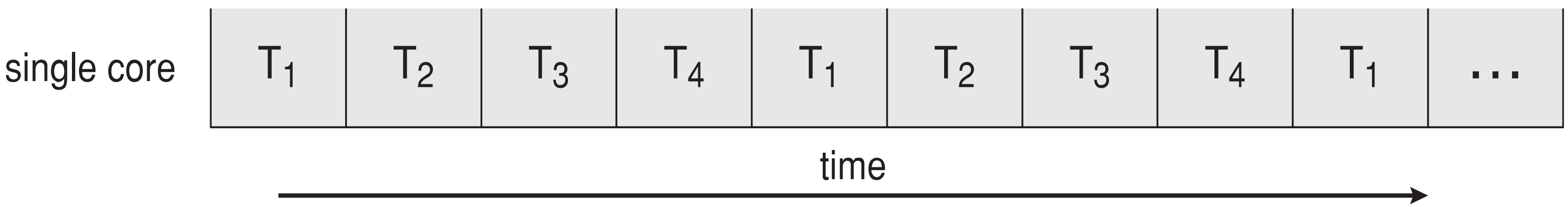
- Stanford Computer Graphics Laboratory

Basic threading

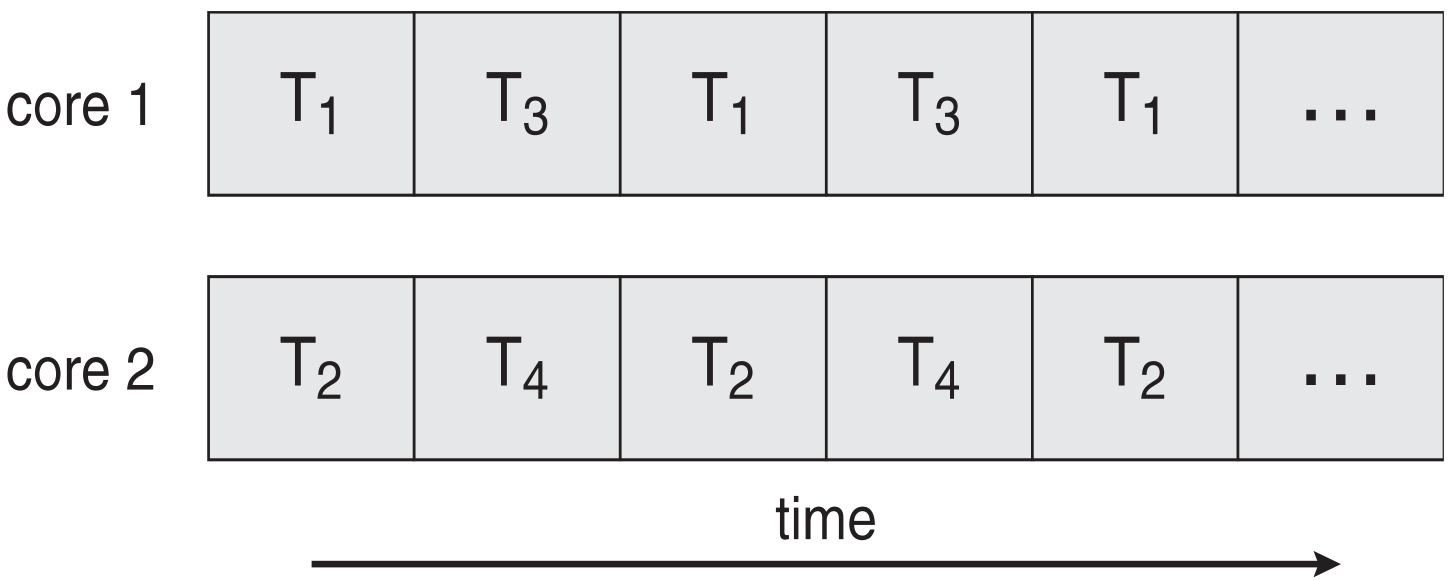
- Concurrent programming allows you to partition a program into separate, independently running tasks
  - Each of these independent tasks (also called subtasks) is driven by a thread of execution
- A thread is a single sequential flow of control within a process
- A single process can thus have multiple concurrently executing tasks
- An underlying mechanism divides up the CPU time for you
  - The CPU will pop around and give each task some of its time

Concurrency

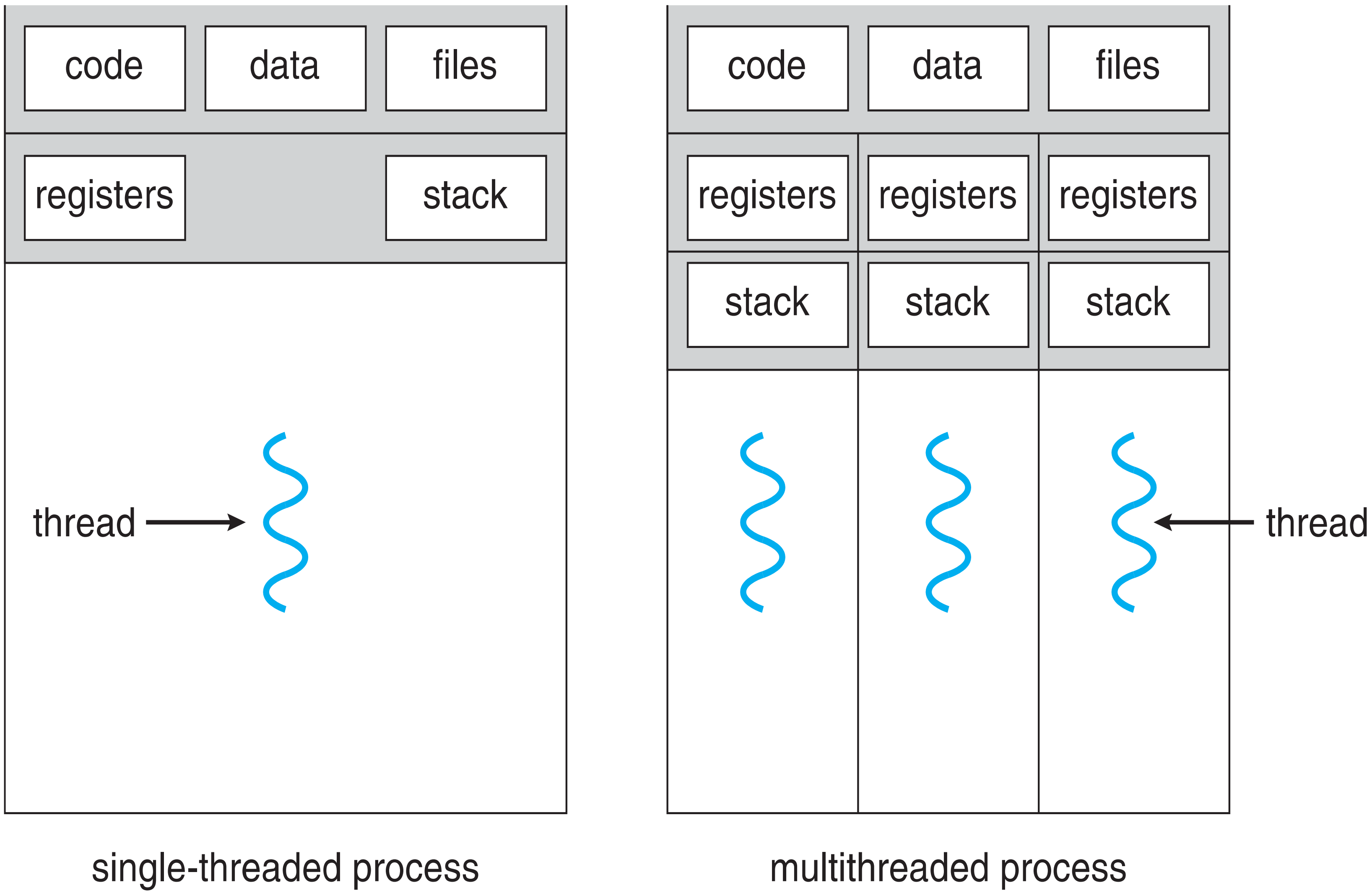
- **Concurrent execution on single-core system:**



- **Concurrent execution on a multi-core system:**



Single and Multithreaded Processes



Defining tasks

- A thread drives a task, so you need a way to describe that task
  - This is provided by the ***Runnable*** interface
- To define a task, simply implement ***Runnable*** and write a ***run( )*** method
- **Thread.yield( )** is to tell the thread scheduler that says, "I've done the important parts of my cycle and this would be a good time to switch to another task for a while."

```java
public class LiftOff implements Runnable {
  protected int countDown = 10; // Default
  private static int taskCount = 0;
  private final int id = taskCount++;
  public LiftOff() {}
  public LiftOff(int countDown) {
    this.countDown = countDown;
  }
  public String status() {
    return "#" + id + "(" +
      (countDown > 0 ? countDown : "Liftoff!") + "), ";
  }
  public void run() {
    while(countDown-- > 0) {
      System.out.print(status());
      Thread.yield();
    }
  }
}
```

```java
public class MainThread {
  public static void main(String[] args) {
    LiftOff launch = new LiftOff();
    launch.run();
  }
}
```

/* Output:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!),
***///:~**

The *Thread* class

- The traditional way to turn a ***Runnable*** object into a working task is to hand it to a ***Thread*** constructor
  - A ***Thread*** constructor **only** needs a ***Runnable*** object
  - Calling a Thread object's ***start( )*** will perform the necessary initialization for the thread and then call that Runnable's ***run( )*** method to start the task in the new thread

```java
public class LiftOff implements Runnable {
  protected int countDown = 10; // Default
  private static int taskCount = 0;
  private final int id = taskCount++;
  public LiftOff() {}
  public LiftOff(int countDown) {
    this.countDown = countDown;
  }
  public String status() {
    return "#" + id + "(" +
      (countDown > 0 ? countDown : "Liftoff!") + "), ";
  }
  public void run() {
    while(countDown-- > 0) {
      System.out.print(status());
      Thread.yield();
    }
  }
}
```

```java
public class BasicThreads {
  public static void main(String[] args) {
    Thread t = new Thread(new LiftOff());
    t.start();
    System.out.println("Waiting for LiftOff");
  }
}
```

/* Output: (90% match)
Waiting for LiftOff
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!),
***///:~**

The *Thread* class (Cont.)

- An anonymous inner class

```
public class BasicThreads {
  public static void main(String[] args) {
    Thread t = new Thread(new Runnable(){
        protected int countDown = 10; // Default
        private int taskCount = 0;
        private final int id = taskCount++;
        public String status() {
            return "#" + id + "(" +
            (countDown > 0 ? countDown : "Liftoff!") + "), ";
        }
        public void run() {
            while(countDown-- > 0) {
            System.out.print(status());
            Thread.yield();
            }
        }
    });
    t.start();
    System.out.println("Waiting for LiftOff");
  }
}
```

The *Thread* class (Cont.)

- You can easily add more threads to drive more tasks
- The output for one run of this program will be different from that of another, because the **thread-scheduling mechanism** is **not deterministic**

```java
public class MoreBasicThreads {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new Thread(new LiftOff()).start();
    System.out.println("Waiting for LiftOff");
  }
}
```

/* Output: (Sample)
Waiting for LiftOff
#0(9), #1(9), #2(9), #3(9), #4(9),
#0(8), #1(8), #2(8), #3(8), #4(8),
#0(7), #1(7), #2(7), #3(7), #4(7),
#0(6), #1(6), #2(6), #3(6), #4(6),
#0(5), #1(5), #2(5), #3(5), #4(5),
#0(4), #1(4), #2(4), #3(4), #4(4),
#0(3), #1(3), #2(3), #3(3), #4(3),
#0(2), #1(2), #2(2), #3(2), #4(2),
#0(1), #1(1), #2(1), #3(1), #4(1),
#0(Liftoff!), #1(Liftoff!), #2(Liftoff!),
#3(Liftoff!), #4(Liftoff!),

Using *Executors*

- *java.util.concurrent* **Executors** **simplify concurrent programming by managing Thread objects for you**
- **Executors allow you to manage the execution of asynchronous tasks without having to explicitly manage the lifecycle of threads**
  - **An *ExecutorService* knows how to build the appropriate context to execute *Runnable* objects**

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

public class CachedThreadPool {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new LiftOff());
    exec.shutdown();
  }
}
```

/* Output: (Sample)
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8), #2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7), #0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5), #2(5), #3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2), #1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2), #4(2), #0(Liftoff!), #1(1), #2(1), #3(1), #4(1), #1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
***///:~**

Using *Executors* (Cont.)

- You can easily replace the *CachedThreadPool* in the previous example with a different type of Executor
- A *FixedThreadPool* uses a limited set of threads to execute the submitted tasks
  - Do expensive thread allocation once, up front
  - Do not Constantly pay for thread creation overhead for every single task

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

public class FixedThreadPool {
  public static void main(String[] args) {
    // Constructor argument is number of threads:
    ExecutorService exec = Executors.newFixedThreadPool(5);
    for(int i = 0; i < 5; i++)
      exec.execute(new LiftOff());
    exec.shutdown();
  }
}
```

```
/* Output: (Sample)
#0(9), #0(8), #1(9), #2(9), #3(9),
#4(9), #0(7), #1(8), #2(8), #3(8),
#4(8), #0(6), #1(7), #2(7), #3(7),
#4(7), #0(5), #1(6), #2(6), #3(6),
#4(6), #0(4), #1(5), #2(5), #3(5),
#4(5), #0(3), #1(4), #2(4), #3(4),
#4(4), #0(2), #1(3), #2(3), #3(3),
#4(3), #0(1), #1(2), #2(2), #3(2),
#4(2), #0(Liftoff!), #1(1), #2(1),
#3(1), #4(1), #1(Liftoff!), #2(Liftoff!),
#3(Liftoff!), #4(Liftoff!),
*///:~
```

Using *Executors* (Cont.)

- A ***SingleThreadExecutor*** is like a ***FixedThreadPool*** with a size of one thread
- If more than one task is submitted to a ***SingleThreadExecutor***, the tasks will be queued and each task will run to completion before the next task is begun, all using the same thread

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

public class SingleThreadExecutor {
  public static void main(String[] args) {
    ExecutorService exec =
      Executors.newSingleThreadExecutor();
    for(int i = 0; i < 5; i++)
      exec.execute(new LiftOff());
    exec.shutdown();
  }
}
```

/* Output:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!), #1(9), #1(8), #1(7), #1(6), #1(5), #1(4), #1(3), #1(2), #1(1), #1(Liftoff!), #2(9), #2(8), #2(7), #2(6), #2(5), #2(4), #2(3), #2(2), #2(1), #2(Liftoff!), #3(9), #3(8), #3(7), #3(6), #3(5), #3(4), #3(3), #3(2), #3(1), #3(Liftoff!), #4(9), #4(8), #4(7), #4(6), #4(5), #4(4), #4(3), #4(2), #4(1), #4(Liftoff!),
***///:~**

Producing return values from tasks

- **If you want the task to produce a value when it's done, you can implement the *Callable* interface rather than the *Runnable* interface**
  - **A *Runnable* is a separate task that performs work, but it doesn't return a value**

- ***Callable* is a generic with a type parameter representing the return value from the method *call( )* (instead of *run( )*)**
  - **Must be invoked using an *ExecutorService submit( )* method**

Producing return values from tasks (Cont.)

- The ***submit( )*** method produces a ***Future*** object, parameterized for the particular type of result returned by the ***Callable***
  - Query the ***Future*** with ***isDone( )*** to see if it has completed
  - ***get( )*** without checking ***isDone( )***, in which case ***get( )*** will **block** until the result is ready

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

class TaskWithResult implements Callable<String> {
  private int id;
  public TaskWithResult(int id) {
    this.id = id;
  }
  public String call() {
    return "result of TaskWithResult " + id;
  }
}

public class CallableDemo {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    ArrayList<Future<String>> results =
      new ArrayList<Future<String>>();
    for(int i = 0; i < 10; i++)
      results.add(exec.submit(new TaskWithResult(i)));
    for(Future<String> fs : results)
      try {
        // get() blocks until completion:
        System.out.println(fs.get());
      } catch(InterruptedException e) {
        System.out.println(e);
        return;
      } catch(ExecutionException e) {
        System.out.println(e);
      } finally {
        exec.shutdown();
      }
  }
}
```

```
/* Output:
result of TaskWithResult 0
result of TaskWithResult 1
result of TaskWithResult 2
result of TaskWithResult 3
result of TaskWithResult 4
result of TaskWithResult 5
result of TaskWithResult 6
result of TaskWithResult 7
result of TaskWithResult 8
result of TaskWithResult 9
*///:~
```

Sleeping

- A simple way to affect the behavior of your tasks is by calling ***sleep( )*** to cease (block) the execution of that task for a given time
- The call to ***sleep( )*** can throw an ***InterruptedException***
  - You can see that this is caught in ***run( )***

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

public class SleepingTask extends LiftOff {
  public void run() {
    try {
      while(countDown-- > 0) {
        System.out.print(status());
        // Old-style:
        // Thread.sleep(100);
        // Java SE5/6-style:
        TimeUnit.MILLISECONDS.sleep(100);
      }
    } catch(InterruptedException e) {
      System.err.println("Interrupted");
    }
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new SleepingTask());
    exec.shutdown();
  }
}
```

/* Output:
#0(9), #1(9), #2(9), #3(9), #4(9),
#0(8), #1(8), #2(8), #3(8), #4(8),
#0(7), #1(7), #2(7), #3(7), #4(7),
#0(6), #1(6), #2(6), #3(6), #4(6),
#0(5), #1(5), #2(5), #3(5), #4(5),
#0(4), #1(4), #2(4), #3(4), #4(4),
#0(3), #1(3), #2(3), #3(3), #4(3),
#0(2), #1(2), #2(2), #3(2), #4(2),
#0(1), #1(1), #2(1), #3(1), #4(1),
#0(Liftoff!), #1(Liftoff!), #2(Liftoff!),
#3(Liftoff!), #4(Liftoff!),
***///:~**

Priority

- The priority of a thread conveys the importance of a thread to the scheduler
  - *Thread.currentThread( )* get a reference to the Thread object
  - the JDK has 10 priority levels
    - MAX_PRIORITY
    - NORM_PRIORITY
    - MIN_PRIORIT

```
/* Output: (70% match)
Thread[pool-1-thread-6,10,main]: 5
Thread[pool-1-thread-6,10,main]: 4
Thread[pool-1-thread-6,10,main]: 3
Thread[pool-1-thread-6,10,main]: 2
Thread[pool-1-thread-6,10,main]: 1
Thread[pool-1-thread-3,1,main]: 5
Thread[pool-1-thread-2,1,main]: 5
Thread[pool-1-thread-1,1,main]: 5
Thread[pool-1-thread-5,1,main]: 5
Thread[pool-1-thread-4,1,main]: 5
...
*///:~
```

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

public class SimplePriorities implements Runnable {
  private int countDown = 5;
  private volatile double d; // No optimization
  private int priority;
  public SimplePriorities(int priority) {
    this.priority = priority;
  }
  public String toString() {
    return Thread.currentThread() + ": " + countDown;
  }
  public void run() {
    Thread.currentThread().setPriority(priority);
    while(true) {
      // An expensive, interruptable operation:
      for(int i = 1; i < 100000; i++) {
        d += (Math.PI + Math.E) / (double)i;
        if(i % 1000 == 0)
          Thread.yield();
      }
      System.out.println(this);
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(
        new SimplePriorities(Thread.MIN_PRIORITY));
    exec.execute(
        new SimplePriorities(Thread.MAX_PRIORITY));
    exec.shutdown();
  }
}
```

Yielding

- Give a hint to the threadscheduling mechanism that you've done and that some other task might as well have the CPU
  - *yield( ) -* you are suggesting that other threads of the same priority might be run

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

public class SimplePriorities implements Runnable {
  private int countDown = 5;
  private volatile double d; // No optimization
  private int priority;
  public SimplePriorities(int priority) {
    this.priority = priority;
  }
  public String toString() {
    return Thread.currentThread() + ": " + countDown;
  }
  public void run() {
    Thread.currentThread().setPriority(priority);
    while(true) {
      // An expensive, interruptable operation:
      for(int i = 1; i < 100000; i++) {
        d += (Math.PI + Math.E) / (double)i;
        if(i % 1000 == 0)
          Thread.yield();
      }
      System.out.println(this);
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(
        new SimplePriorities(Thread.MIN_PRIORITY));
    exec.execute(
        new SimplePriorities(Thread.MAX_PRIORITY));
    exec.shutdown();
  }
}
```

```
/* Output: (70% match)
Thread[pool-1-thread-6,10,main]: 5
Thread[pool-1-thread-6,10,main]: 4
Thread[pool-1-thread-6,10,main]: 3
Thread[pool-1-thread-6,10,main]: 2
Thread[pool-1-thread-6,10,main]: 1
Thread[pool-1-thread-3,1,main]: 5
Thread[pool-1-thread-2,1,main]: 5
Thread[pool-1-thread-1,1,main]: 5
Thread[pool-1-thread-5,1,main]: 5
Thread[pool-1-thread-4,1,main]: 5
...
*///:~
```

Daemon threads

- A "daemon" thread is intended to provide a general service in the background as long as the program is running

```java
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

public class SimpleDaemons implements Runnable {
  public void run() {
    try {
      while(true) {
        TimeUnit.MILLISECONDS.sleep(100);
        print(Thread.currentThread() + " " + this);
      }
    } catch(InterruptedException e) {
      print("sleep() interrupted");
    }
  }
  public static void main(String[] args) throws Exception {
    for(int i = 0; i < 10; i++) {
      Thread daemon = new Thread(new SimpleDaemons());
      daemon.setDaemon(true); // Must call before start()
      daemon.start();
    }
    print("All daemons started");
    TimeUnit.MILLISECONDS.sleep(175);
  }
}
```

```
/* Output: (Sample)
All daemons started
Thread[Thread-0,5,main] SimpleDaemons@530daa
Thread[Thread-1,5,main] SimpleDaemons@a62fc3
Thread[Thread-2,5,main] SimpleDaemons@89ae9e
Thread[Thread-3,5,main] SimpleDaemons@1270b73
Thread[Thread-4,5,main] SimpleDaemons@60aeb0
Thread[Thread-5,5,main] SimpleDaemons@16caf43
Thread[Thread-6,5,main] SimpleDaemons@66848c
Thread[Thread-7,5,main] SimpleDaemons@8813f2
Thread[Thread-8,5,main] SimpleDaemons@1d58aae
Thread[Thread-9,5,main] SimpleDaemons@83cc67
...
*///:~
```

Daemon threads

- It is possible to customize the attributes (daemon, priority, name) of threads created by **Executors** by writing a custom **ThreadFactory**

```java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadFactory implements ThreadFactory {
  public Thread newThread(Runnable r) {
    Thread t = new Thread(r);
    t.setDaemon(true);
    return t;
  }
}
```

```java
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DaemonFromFactory implements Runnable {
  public void run() {
    try {
      while(true) {
        TimeUnit.MILLISECONDS.sleep(100);
        print(Thread.currentThread() + " " + this);
      }
    } catch(InterruptedException e) {
      print("Interrupted");
    }
  }
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool(
      new DaemonThreadFactory());
    for(int i = 0; i < 10; i++)
      exec.execute(new DaemonFromFactory());
    print("All daemons started");
    TimeUnit.MILLISECONDS.sleep(500); // Run for a while
  }
}
```

Daemon threads

- **When all of the non-daemon threads complete, the program is terminated, killing all daemon threads in the process**

- **If a thread is a daemon, then any threads it creates will automatically be daemons**

- **You can find out if a thread is a daemon by calling *isDaemon( )***

Coding variations

- The task classes all implement ***Runnable***
- Use the alternative approach of inheriting directly from ***Thread***

```java
public class SimpleThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  public SimpleThread() {
    // Store the thread name:
    super(Integer.toString(++threadCount));
    start();
  }
  public String toString() {
    return "#" + getName() + "(" + countDown + "), ";
  }
  public void run() {
    while(true) {
      System.out.print(this);
      if(--countDown == 0)
        return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread();
  }
}
```

/* Output:
#1(5), #1(4), #1(3), #1(2), #1(1), #2(5), #2(4), #2(3), #2(2), #2(1), #3(5), #3(4), #3(3), #3(2), #3(1), #4(5), #4(4), #4(3), #4(2), #4(1), #5(5), #5(4), #5(3), #5(2), #5(1),
***///:~**

Joining a thread

- One thread may call ***join( )*** on another thread to wait for the second thread to complete before proceeding
  - The call to join( ) may be aborted by calling ***interrupt( )*** on the calling thread, so a try-catch clause is required

```java
import static net.mindview.util.Print.*;

class Sleeper extends Thread {
  private int duration;
  public Sleeper(String name, int sleepTime) {
    super(name);
    duration = sleepTime;
    start();
  }
  public void run() {
    try {
      sleep(duration);
    } catch(InterruptedException e) {
      print(getName() + " was interrupted. " +
        "isInterrupted(): " + isInterrupted());
      return;
    }
    print(getName() + " has awakened");
  }
}

```

// Output:
Grumpy was interrupted. isInterrupted(): false
Doc join completed
Sleepy has awakened
Dopey join completed

```java
class Joiner extends Thread {
  private Sleeper sleeper;
  public Joiner(String name, Sleeper sleeper) {
    super(name);
    this.sleeper = sleeper;
    start();
  }
  public void run() {
   try {
      sleeper.join();
    } catch(InterruptedException e) {
      print("Interrupted");
    }
    print(getName() + " join completed");
  }
}

public class Joining {
  public static void main(String[] args) {
    Sleeper
      sleepy = new Sleeper("Sleepy", 1500),
      grumpy = new Sleeper("Grumpy", 1500);
    Joiner
      dopey = new Joiner("Dopey", sleepy),
      doc = new Joiner("Doc", grumpy);
    grumpy.interrupt();
  }
}
```

Sharing resources

- You can think of a single-threaded program as one lonely entity moving around through your problem space and doing one thing at a time

- Think about the problem of two entities trying to use the same resource at the same time
  - Two people try to park in the same space
  - walk through a door at the same time
  - Talk at the same time

- With concurrency, things aren't lonely anymore, but you now have the possibility of two or more tasks interfering with each other

Improperly accessing resources

- Consider the following example, where one task generates even numbers and other tasks consume those numbers
- Here, the only job of the consumer tasks is to check the validity of the even numbers

```java
public abstract class IntGenerator {
  private volatile boolean canceled = false;
  public abstract int next();
  // Allow this to be canceled:
  public void cancel() { canceled = true; }
  public boolean isCanceled() { return canceled; }
} ///:~
```

```java
public class EvenGenerator extends IntGenerator {
  private int currentEvenValue = 0;
  public int next() {
    ++currentEvenValue; // Danger point here!
    ++currentEvenValue;
    return currentEvenValue;
  }
  public static void main(String[] args) {
    EvenChecker.test(new EvenGenerator());
  }
}
```

/* Output: (Sample)
Press Control-C to exit
89476993 not even!
89476993 not even!
***///:~**

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

public class EvenChecker implements Runnable {
  private IntGenerator generator;
  private final int id;
  public EvenChecker(IntGenerator g, int ident) {
    generator = g;
    id = ident;
  }
  public void run() {
    while(!generator.isCanceled()) {
      int val = generator.next();
      if(val % 2 != 0) {
        System.out.println(val + " not even!");
        generator.cancel(); // Cancels all EvenCheckers
      }
    }
  }
  // Test any type of IntGenerator:
  public static void test(IntGenerator gp, int count) {
    System.out.println("Press Control-C to exit");
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < count; i++)
      exec.execute(new EvenChecker(gp, i));
    exec.shutdown();
  }
  // Default value for count:
  public static void test(IntGenerator gp) {
    test(gp, 10);
  }
}
```

Resolving shared resource contention

- The previous example shows a fundamental problem when you are using threads
  - You never know when a thread might be run
- Preventing this kind of collision is simply a matter of putting a lock on a resource when one task is using it
- To solve the problem of thread collision, virtually all concurrency schemes serialize access to shared resources
  - Only one task at a time is allowed to access the shared resource
- Mutex
  - To accomplish it, put a clause around a piece of code that only allows one task at a time to pass through that piece of code
  - This clause produces mutual exclusion, a common name for such a mechanism is *mutex*
- Java has built-in support in the form of the *synchronized* keyword
  - All objects automatically contain a single lock

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

Resolving shared resource contention (Cont.)

- **Synchronizing the EvenGenerator**
- **Note**
  - **It's especially important to make fields *private* when working with concurrency**
  - **Otherwise the *synchronized* keyword cannot prevent another task from accessing a field directly, and thus producing collisions**

```java
public class
SynchronizedEvenGenerator extends IntGenerator {
  private int currentEvenValue = 0;
  public synchronized int next() {
    ++currentEvenValue;
    Thread.yield(); // Cause failure faster
    ++currentEvenValue;
    return currentEvenValue;
  }
  public static void main(String[] args) {
    EvenChecker.test(new SynchronizedEvenGenerator());
  }
}
```

Resolving shared resource contention (Cont.)

- **Using explicit *Lock* objects**
  - **The *Lock* object must be explicitly created, locked and unlocked**
  - **It is more *flexible* for solving certain types of problems**

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

public class MutexEvenGenerator extends IntGenerator {
  private int currentEvenValue = 0;
  private Lock lock = new ReentrantLock();
  public int next() {
    lock.lock();
    try {
      ++currentEvenValue;
      Thread.yield(); // Cause failure faster
      ++currentEvenValue;
      return currentEvenValue;
    } finally {
      lock.unlock();
    }
  }
  public static void main(String[] args) {
    EvenChecker.test(new MutexEvenGenerator());
  }
}
```

Resolving shared resource contention (Cont.)

- **Critical sections**
  - **Prevent multiple thread access to part of the code inside a method instead of the entire method**
  - **The section of code you want to isolate this way is called a critical section and is created using the *synchronized* keyword**
  - **This is also called a *synchronized* block; before it can be entered, the lock must be acquired on syncObject**

```
synchronized(syncObject) {
  // This code can be accessed
  // by only one task at a time
}
```

Resolving shared resource contention (Cont.)

Thread local storage

- A second way to prevent tasks from colliding over shared resources is to **eliminate the sharing of variables**
- Thread local storage is a mechanism that automatically **creates different storage for the same variable**

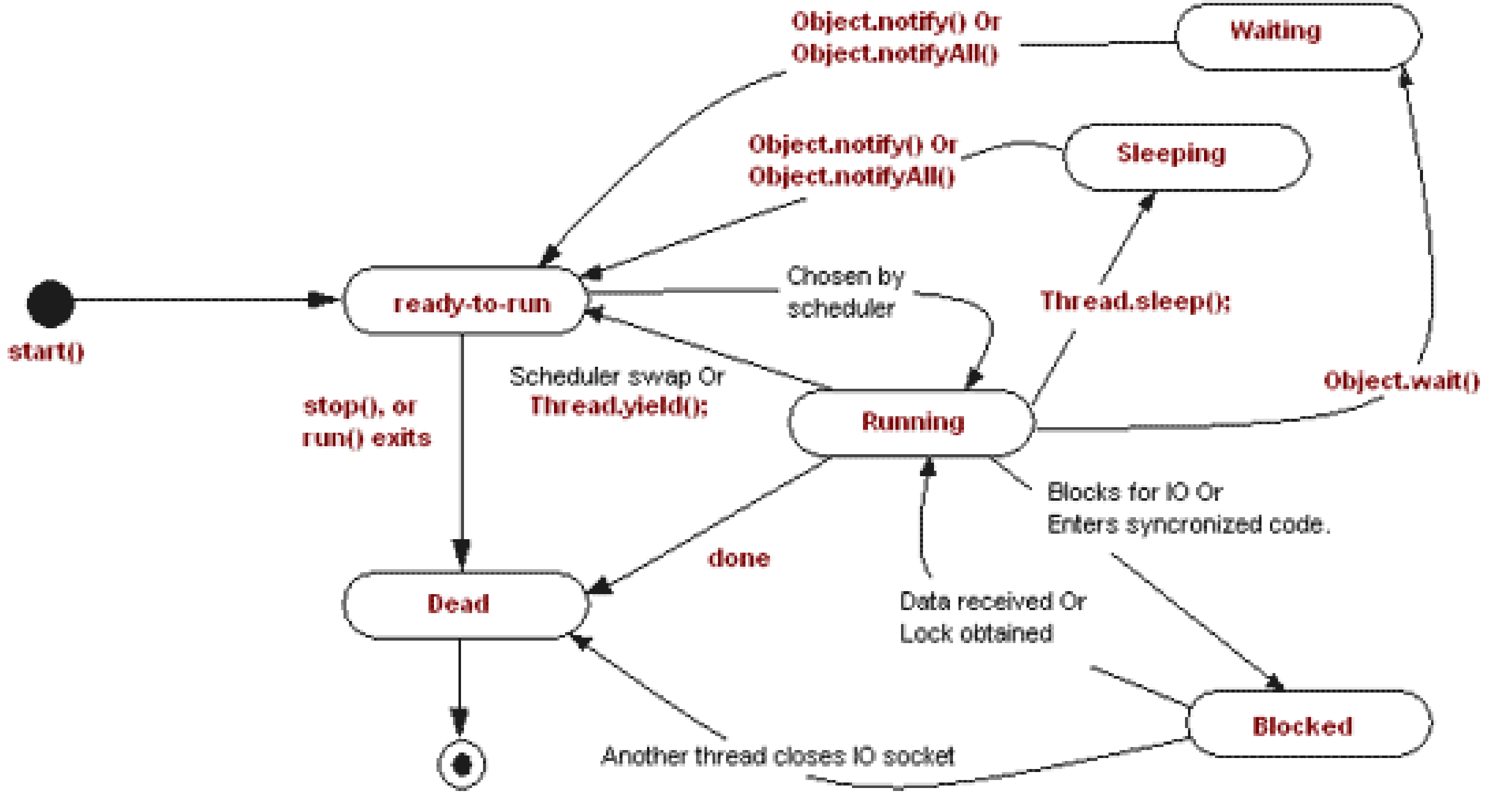
```java
import java.util.concurrent.*;
import java.util.*;

class Accessor implements Runnable {
  private final int id;
  public Accessor(int idn) { id = idn; }
  public void run() {
    while(!Thread.currentThread().isInterrupted()) {
      ThreadLocalVariableHolder.increment();
      System.out.println(this);
      Thread.yield();
    }
  }
  public String toString() {
    return "#" + id + ": " +
      ThreadLocalVariableHolder.get();
  }
}

```

```java
public class ThreadLocalVariableHolder {
  private static ThreadLocal<Integer> value =
    new ThreadLocal<Integer>() {
      private Random rand = new Random(47);
      protected synchronized Integer initialValue() {
        return rand.nextInt(10000);
      }
    };
  public static void increment() {
    value.set(value.get() + 1);
  }
  public static int get() { return value.get(); }
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Accessor(i));
    TimeUnit.SECONDS.sleep(3);  // Run for a while
    exec.shutdownNow();         // All Accessors will quit
  }
}
```

/* Output: (Sample)
#0: 9259 #1: 556 #2: 6694 #3: 1862 #4: 962 #0: 9260
#1: 557 #2: 6695 #3: 1863 #4: 963
...
***///:~**

Terminating tasks

- Life cycle of a thread
  - New, Ready, Running, Blocked, Dead
- Terminating tasks
  - *cancel( )* and *isCanceled( )* methods
  - *Interrupt( )* method



Cooperation between tasks

- Learn how to make tasks **cooperate with each other**, so that multiple tasks can work together to solve a problem
- The key issue when tasks are cooperating is **handshaking between those tasks**
- To accomplish this handshaking, we use the same foundation: the **mutex**
  - Guarantee that only one task can respond to a signal
  - Eliminate any possible race conditions
- On top of the mutex, we add a way for a task to **suspend itself until some external state changes**
  - Use the Object methods ***wait( )*** and ***notifyAll( )***

Cooperation between tasks

- *wait( )* suspends the task while waiting for the world to change
- Only when a *notify( )* or *notifyAll( )* occurs—suggesting that something of interest may have happened—does the task wake up and check for changes
- *wait( )* provides a way to synchronize activities between tasks
- *sleep( )* and *yield( )* does not release the object lock when it is called
- There are two forms of *wait( )*
  - One version takes an argument in milliseconds
  - More commonly used form of *wait( )* takes no arguments. This *wait( )* continues indefinitely until the thread receives a *notify( )* or *notifyAll( )*

Cooperation between tasks

```java
package concurrency.waxomatic;
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Car {
  private boolean waxOn = false;
  public synchronized void waxed() {
    waxOn = true; // Ready to buff
    notifyAll();
  }
  public synchronized void buffed() {
    waxOn = false; // Ready for another coat of wax
    notifyAll();
  }
  public synchronized void waitForWaxing()
  throws InterruptedException {
    while(waxOn == false)
      wait();
  }
  public synchronized void waitForBuffing()
  throws InterruptedException {
    while(waxOn == true)
      wait();
  }
}

```

```java
class WaxOn implements Runnable {
  private Car car;
  public WaxOn(Car c) { car = c; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        printnb("Wax On! ");
        TimeUnit.MILLISECONDS.sleep(200);
        car.waxed();
        car.waitForBuffing();
      }
    } catch(InterruptedException e) {
      print("Exiting via interrupt");
    }
    print("Ending Wax On task");
  }
}

class WaxOff implements Runnable {
  private Car car;
  public WaxOff(Car c) { car = c; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        car.waitForWaxing();
        printnb("Wax Off! ");
        TimeUnit.MILLISECONDS.sleep(200);
        car.buffed();
      }
    } catch(InterruptedException e) {
      print("Exiting via interrupt");
    }
    print("Ending Wax Off task");
  }
}

```

```java
public class WaxOMatic {
  public static void main(String[] args) throws Exception {
    Car car = new Car();
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new WaxOff(car));
    exec.execute(new WaxOn(car));
    TimeUnit.SECONDS.sleep(5); // Run for a while...
    exec.shutdownNow(); // Interrupt all tasks
  }
}
```

Using pipes for I/O between tasks

```java
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Sender implements Runnable {
  private Random rand = new Random(47);
  private PipedWriter out = new PipedWriter();
  public PipedWriter getPipedWriter() { return out; }
  public void run() {
    try {
      while(true)
        for(char c = 'A'; c <= 'z'; c++) {
          out.write(c);
          TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
        }
    } catch(IOException e) {
      print(e + " Sender write exception");
    } catch(InterruptedException e) {
      print(e + " Sender sleep interrupted");
    }
  }
}

class Receiver implements Runnable {
  private PipedReader in;
  public Receiver(Sender sender) throws IOException {
    in = new PipedReader(sender.getPipedWriter());
  }
  public void run() {
    try {
      while(true) {
        // Blocks until characters are there:
        printnb("Read: " + (char)in.read() + ", ");
      }
    } catch(IOException e) {
      print(e + " Receiver read exception");
    }
  }
}

public class PipedIO {
  public static void main(String[] args) throws Exception {
    Sender sender = new Sender();
    Receiver receiver = new Receiver(sender);
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(sender);
    exec.execute(receiver);
    TimeUnit.SECONDS.sleep(4);
    exec.shutdownNow();
  }
}
```

/* Output: (65% match)
Read: A, Read: B, Read: C, Read: D, Read: E, Read: F, Read: G, Read: H, Read: I, Read: J, Read: K, Read: L, Read: M, java.lang.InterruptedException: sleep interrupted Sender sleep interrupted java.io.InterruptedIOException Receiver read exception

Thank you

zhenling@seu.edu.cn