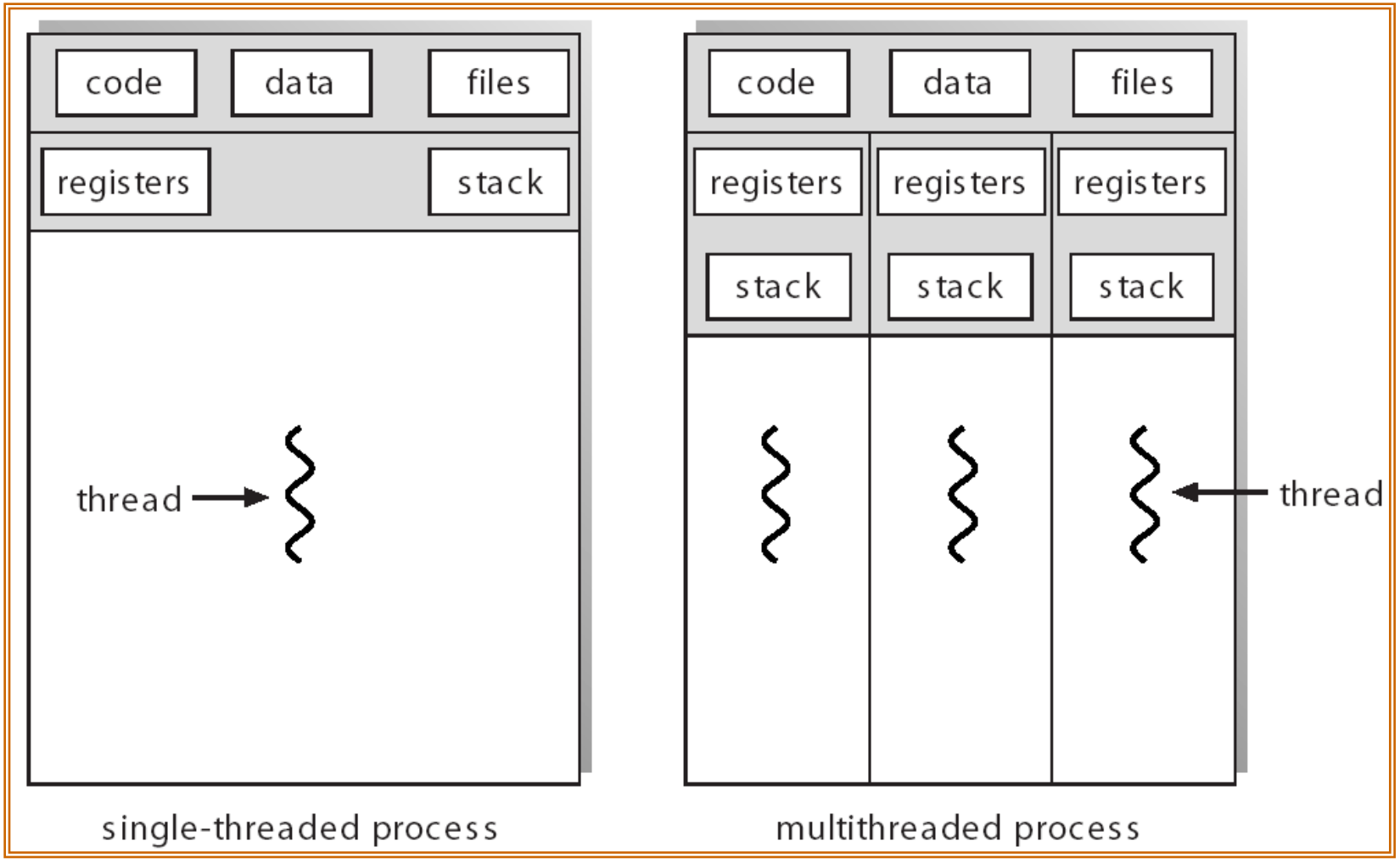
Chapter 11

Thread

Topics

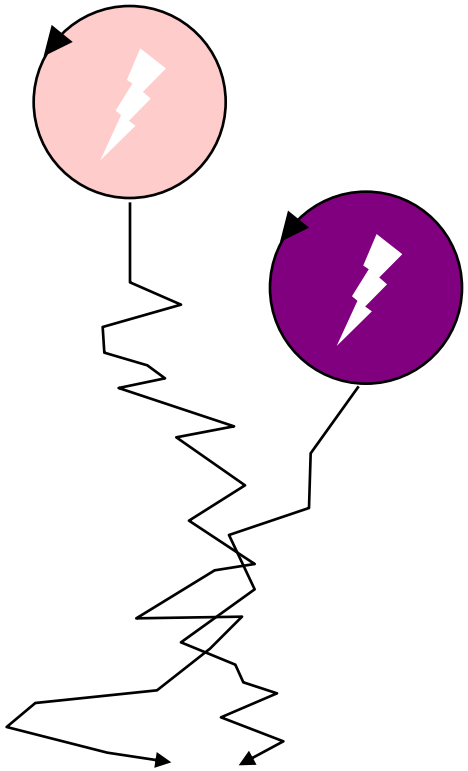
- Overview
- Multithreading Models
- Threading Issues

Single and Multithreaded Processes

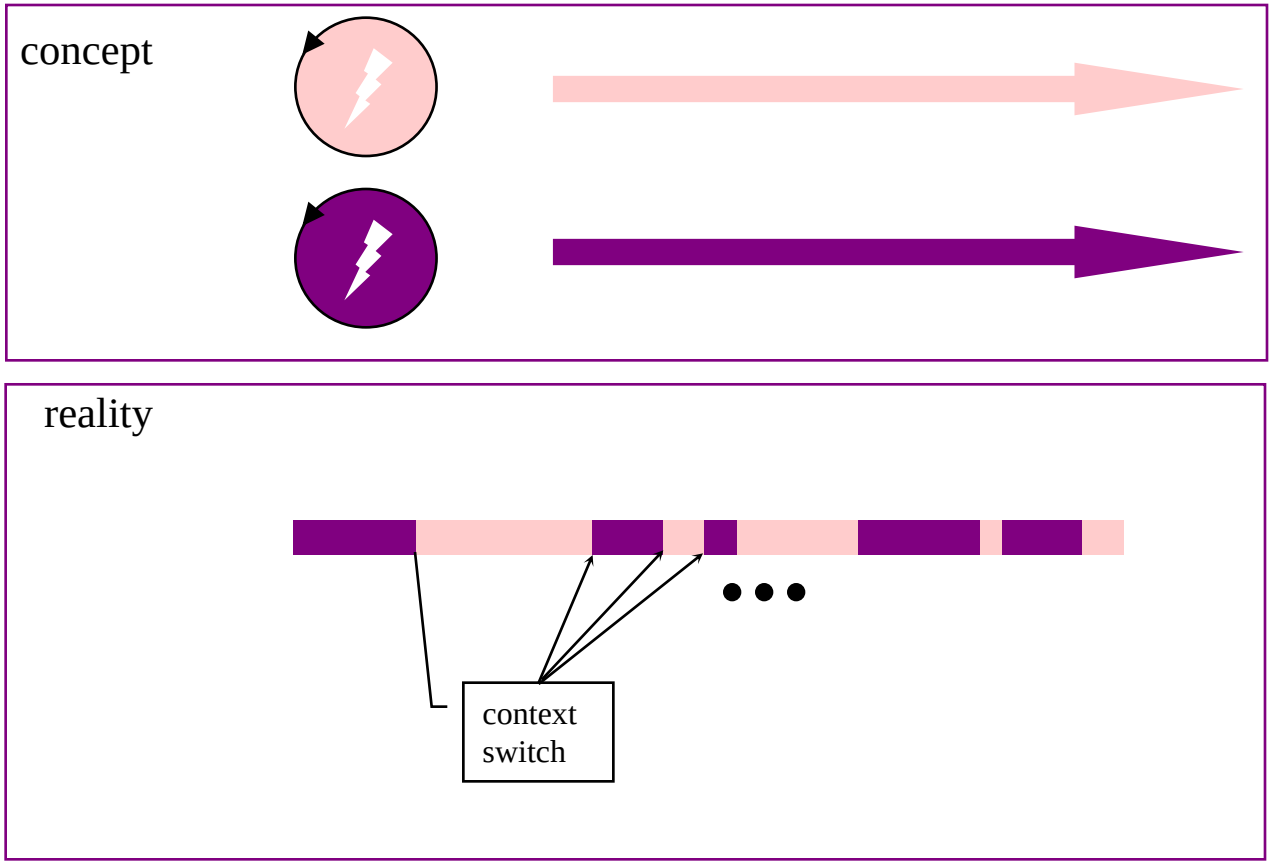


Threads

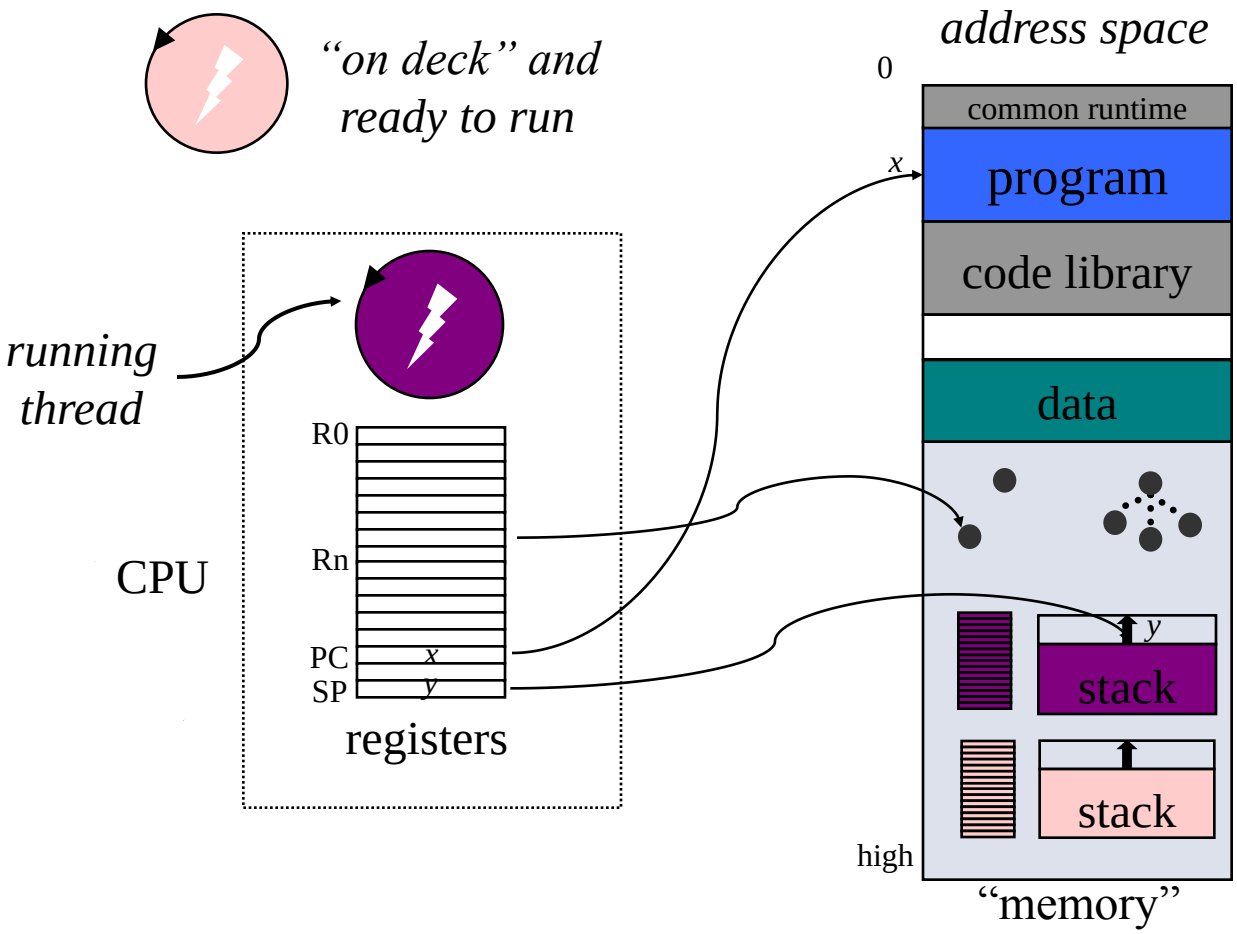
- A *thread* is a schedulable stream of control.
  - defined by CPU register values (PC, SP)
  - *suspend*: save register values in memory
  - *resume*: restore registers from memory
- Multiple threads can execute independently:
  - They can run in parallel on multiple CPUs...
    - ***physical concurrency***
  - …or arbitrarily interleaved on a single CPU.
    - ***logical concurrency***
  - Each thread must have its own stack.



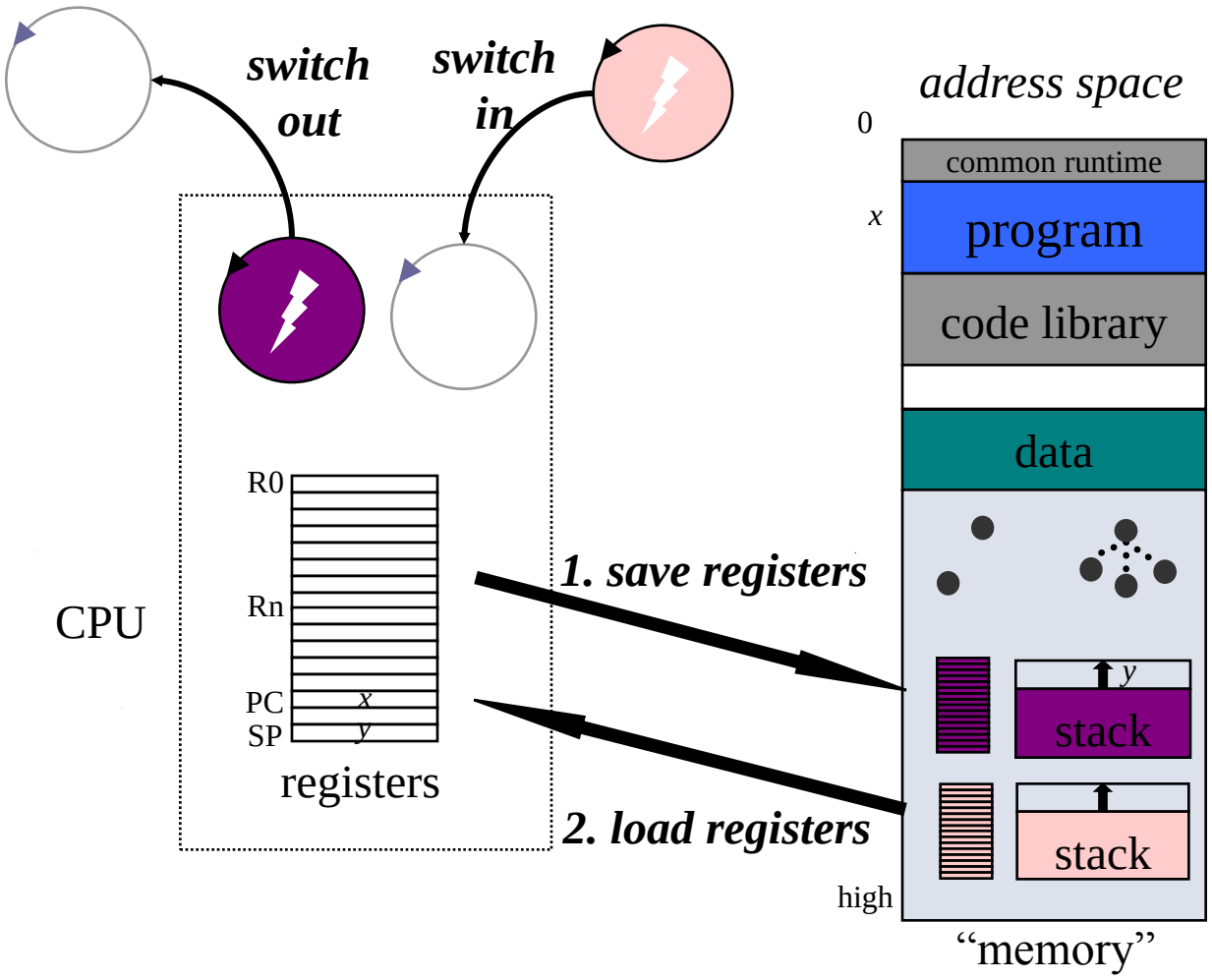
Two Threads Sharing a CPU



A Program With Two Threads



Thread Context Switch

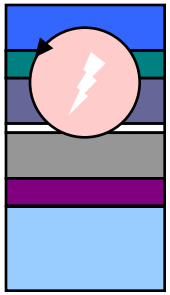


Benefits

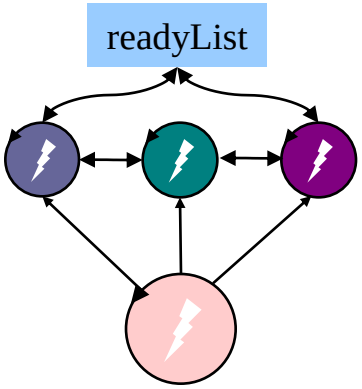
- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

User-level Threads

- No special support needed from the kernel (use any Unix)
- Thread creation and context switch are fast (no system call)
- Defines its own thread model and scheduling policies
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

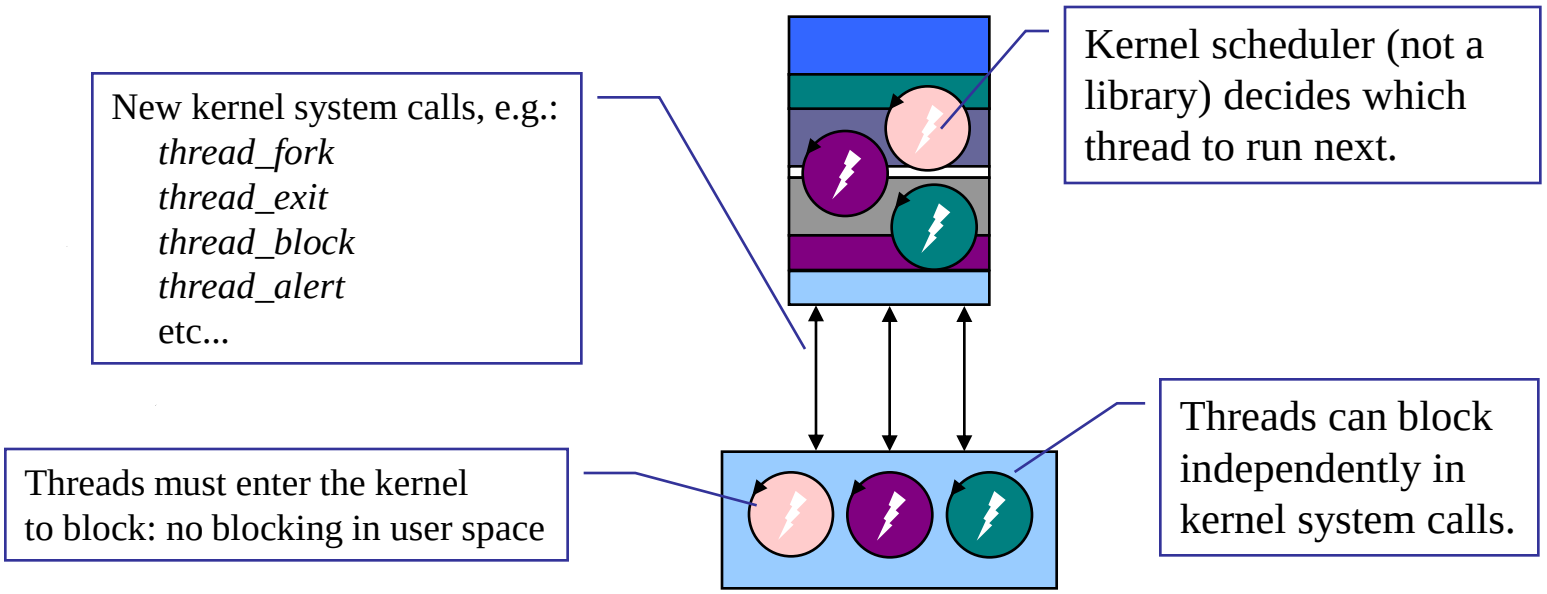


```
while(1) {
      t = get next ready thread;
      scheduler->Run(t);
}
```



Kernel-Supported Threads

- Most newer OS kernels have *kernel-supported threads*.
  - Thread model and scheduling defined by OS
    - Solaris, Linux, Tru64 UNIX, Mac OS X



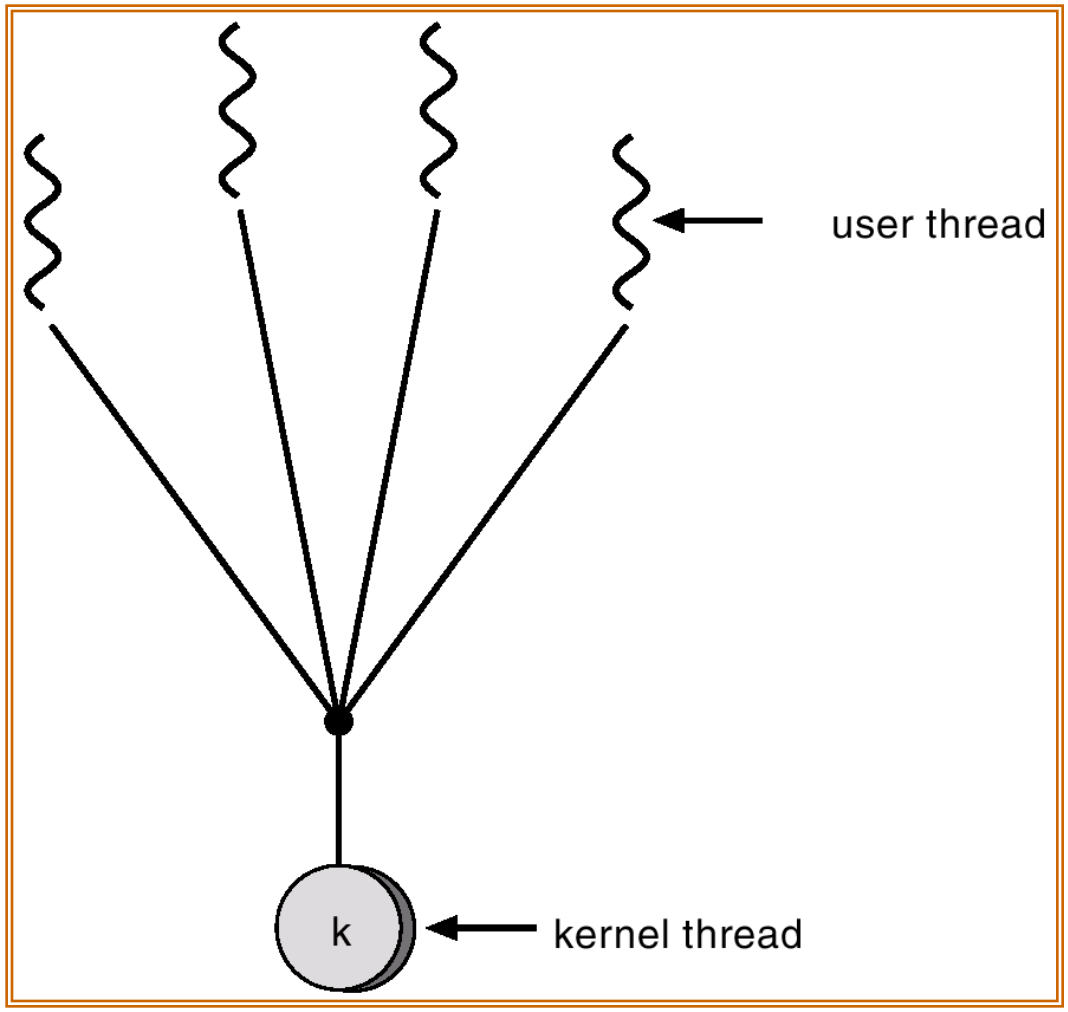
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples
  - Solaris Green Threads
  - GNU Portable Threads

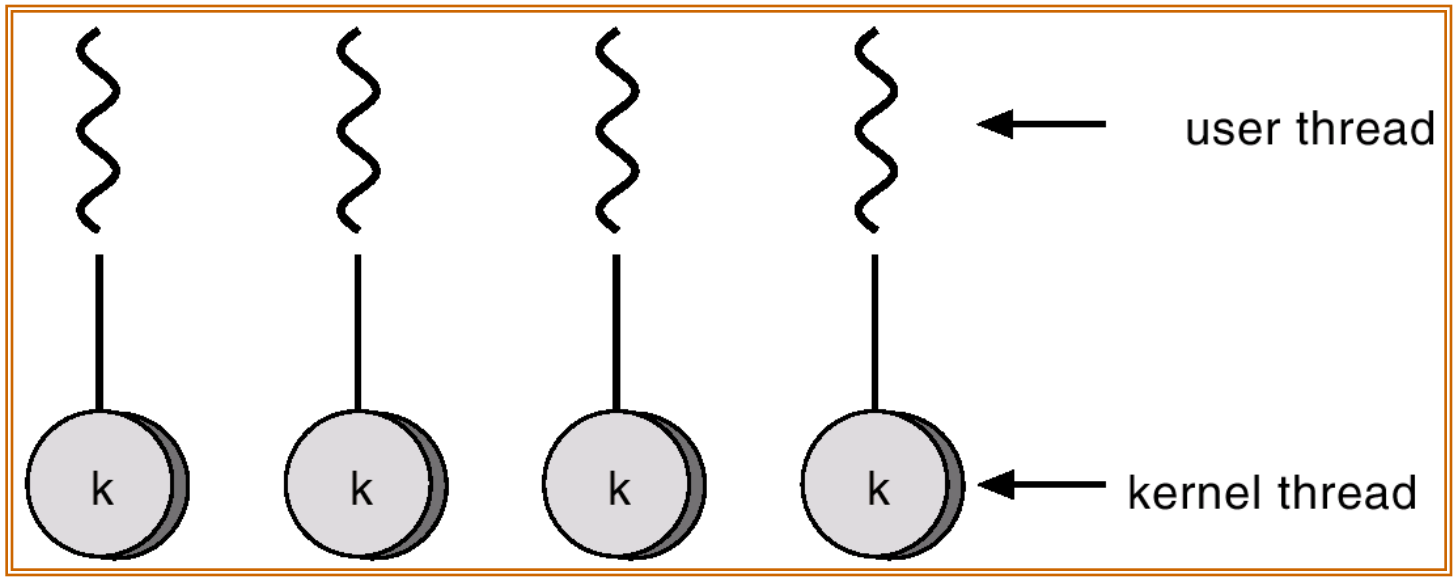
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

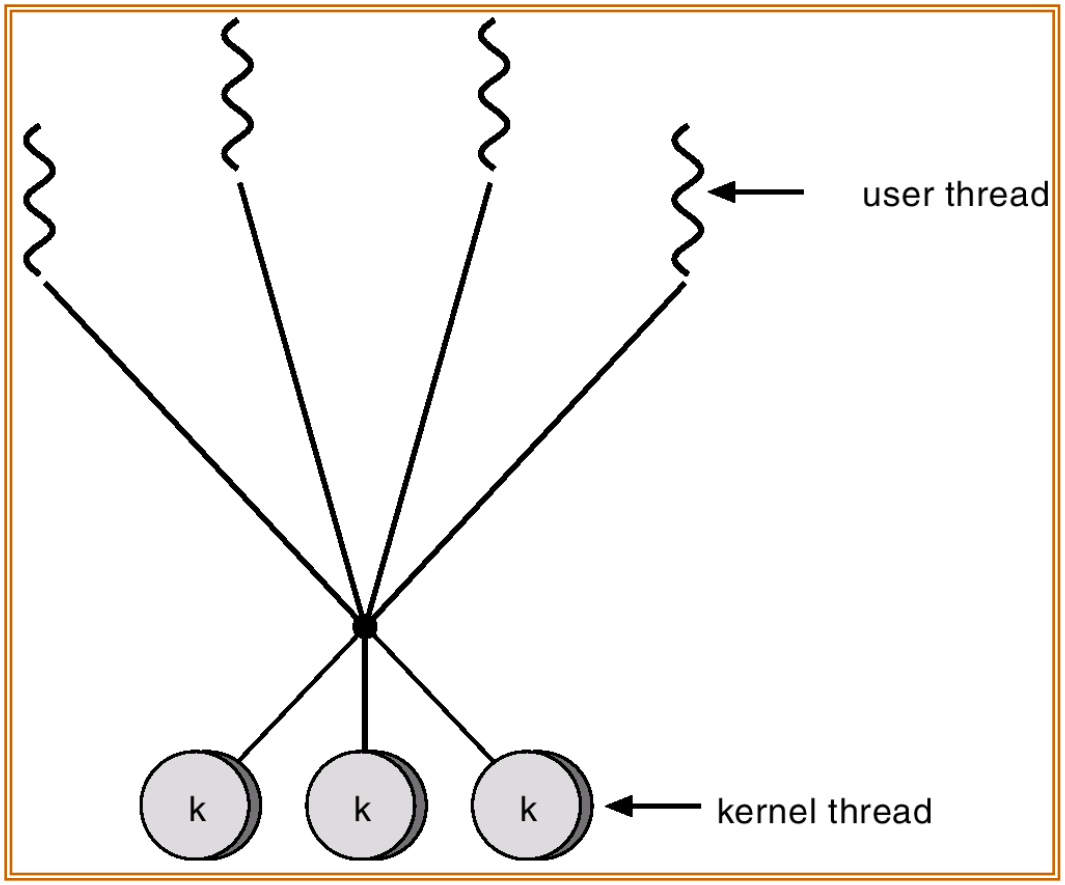
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

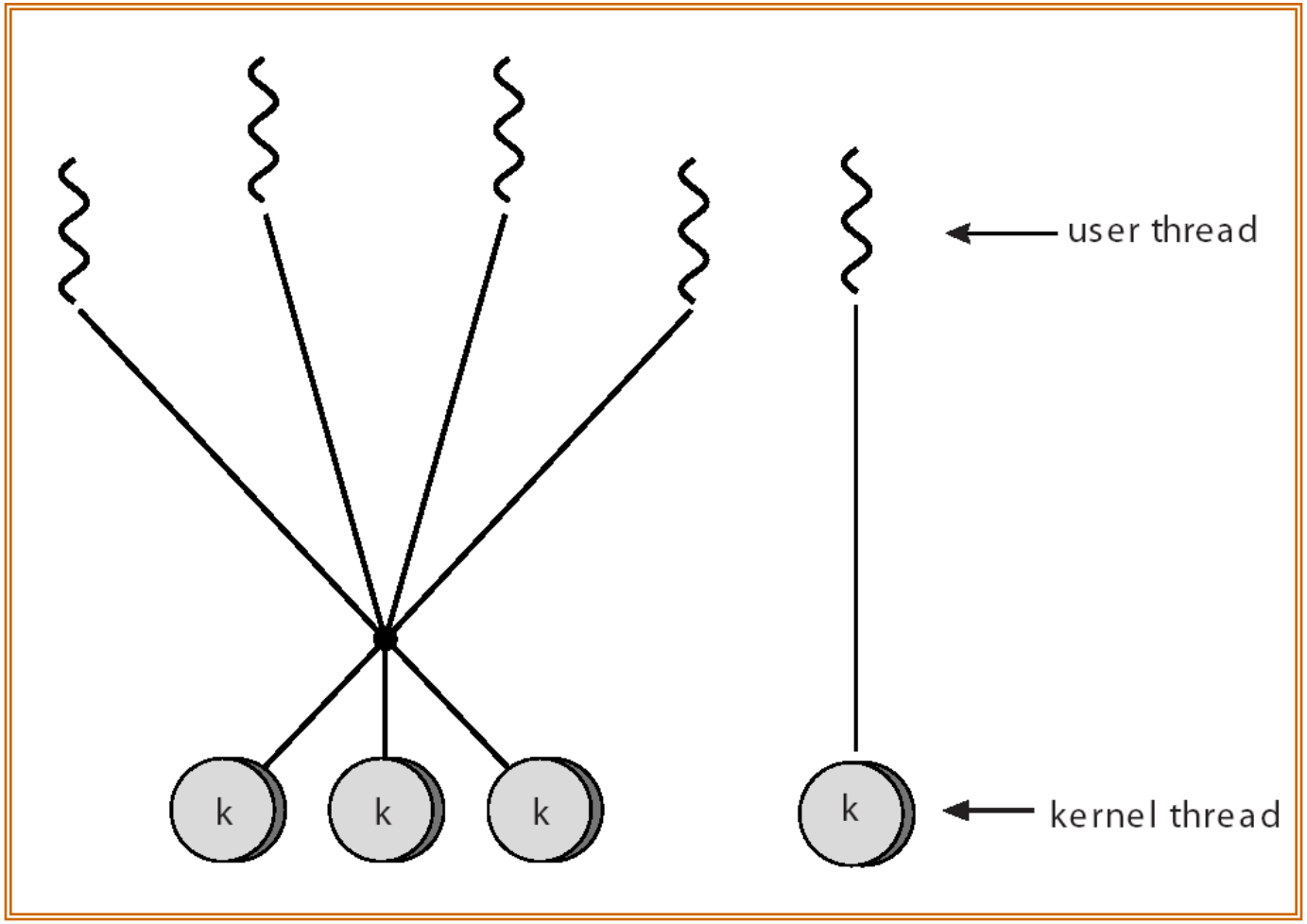
Many-to-Many Model



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-level Model



Multithreaded Programming

- Unlike most other computer languages, Java provides built-in support for *multithreaded programming.*
- Multithreading is a specialized form of multitasking.
- There are two distinct types of multitasking:
  - process-based.
  - thread-based.
- Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is.
- Process-based multitasking is typically managed by the OS, and it involves the concurrent execution of multiple processes.
- In Java, when you run separate Java applications (e.g., running two different Java programs simultaneously), you are effectively launching separate processes. Java doesn't directly control process-based multitasking, as it's primarily the responsibility of the underlying OS.

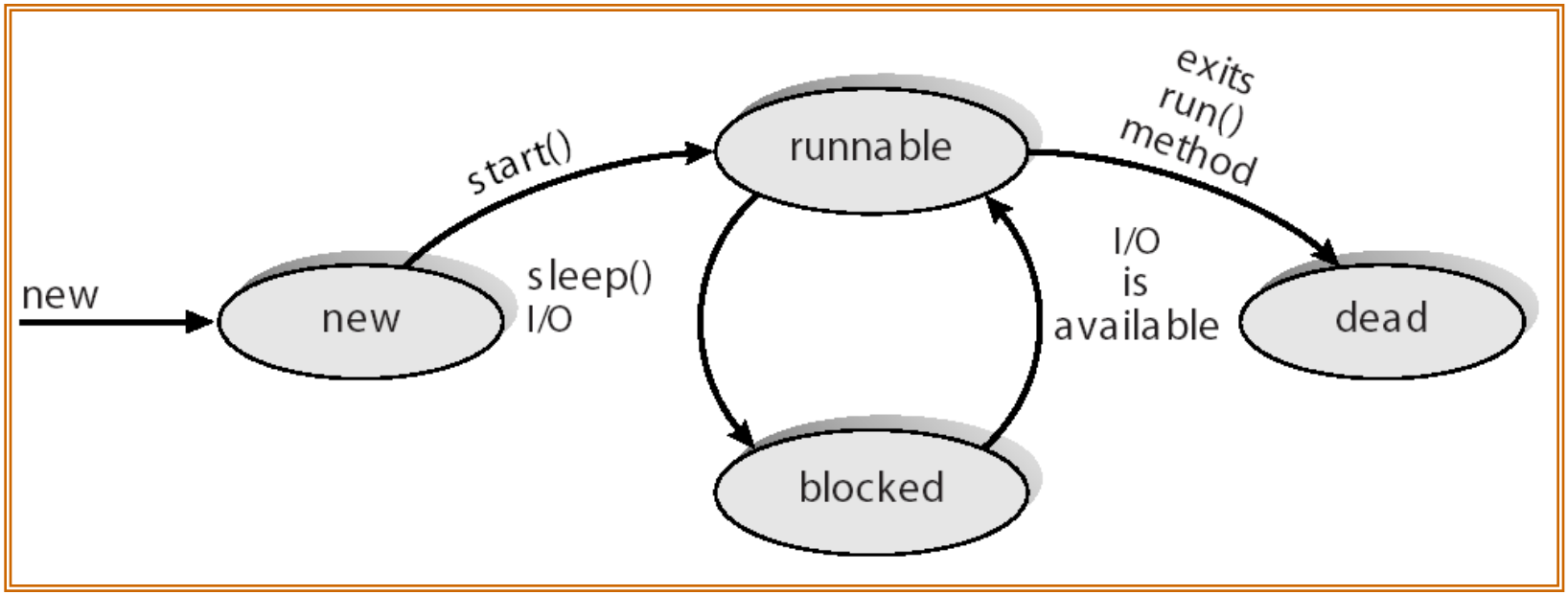
The java thread model

- The value of a multithreaded environment is best understood in contrast to its counterpart.
- Single-threaded systems use an approach called an *event loop* with *polling*.
- In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
- Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler.
- Until this event handler returns, nothing else can happen in the system. This wastes CPU time.
- In a singled-threaded environment, when a thread *blocks* (that is, suspends - execution) because it is waiting for some resource, the entire program stops running.

The java thread model

- In Java's multithreading the main loop/polling mechanism is eliminated.
- One thread can pause without stopping other parts of your program.
- For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
- When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Java Thread States



Thread Priorities

- Thread priorities are integers that specify the relative priority of one thread to another.
- As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the tally thread running. Instead, a thread's priority is used to decide when to switch from a running thread to the next. This is called a *context switch*. The rules that determine when a context switch takes place are simple:
  - *A thread can voluntarily relinquish control.* This is done by explicitly yielding, sleeping, or blocking on pending I/0. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
  - *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking.*

Synchronization

- Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it.
- Java implements interprocess synchronization by monitor.
- You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.
- There is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built in to the language.

Messaging

- When programming with most other languages, you must depend on the operating system to establish communication between threads. This adds overhead.
- By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have.
- Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the Thread class, its methods, and its companion interface - Runnable.
- To create a new thread, your program will either extend Thread or implement the Runnable interface.
- The Thread class defines the following methods.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread

- Thus far, all the examples have used a single thread of execution. The remainder of this presentation explains how to use Thread and Runnable to create and manage threads, beginning with the one thread that all Java programs have: the main thread.
- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins.
- The main thread is important for two reasons:
  - It is the thread from which other "child" threads will be spawned.
  - Often it must be the last thread to finish execution because it performs various shutdown actions.

The Main Thread

- Although the main thread is created automatically when your program is started, it can be controlled through a Thread object.

```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);
```

The Main Thread

```
try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

The Main Thread

- Output

Current thread: Thread[main,5,main]

After name change: Thread[My Thread,5,main]

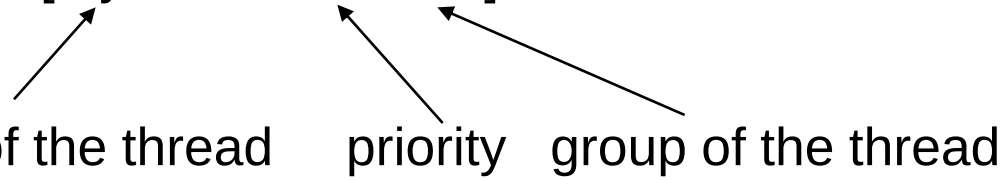
5

4

3

2

1



Creating a Thread

- Java defines two ways
  - You can implement the Runnable interface.
  - You can extend the Thread class, itself.

Implementing Runnable

```
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }
```

Implementing Runnable

```
// This is the entry point for the second thread.
 public void run() {
   try {
     for(int i = 5; i > 0; i--) {
       System.out.println("Child Thread: " + i);
       Thread.sleep(500);
     }
   } catch (InterruptedException e) {
     System.out.println("Child interrupted.");
   }
   System.out.println("Exiting child thread.");
 }
}
```

Implementing Runnable

```
class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // create a new thread
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Implementing Runnable

- Output (May be)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Extending Thread

```
// Create a second thread by extending Thread
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }
```

Extending Thread

```
// This is the entry point for the second thread.
 public void run() {
   try {
     for(int i = 5; i > 0; i--) {
       System.out.println("Child Thread: " + i);
       Thread.sleep(500);
     }
   } catch (InterruptedException e) {
     System.out.println("Child interrupted.");
   }
   System.out.println("Exiting child thread.");
 }
}
```

Extending Thread

```
class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Choosing an Approach

- Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is run( ).
- Of course, the same method required when you implement Runnable.
- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable.
- This is up to you, of course.

Creating Multiple Threads

```
// Create multiple threads.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }
```

Creating Multiple Threads

```
// This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + "Interrupted");
    }
    System.out.println(name + " exiting.");
  }
}
```

Creating Multiple Threads

```
class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");
    try {
      // wait for other threads to end
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Creating Multiple Threads

- Output (May be)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Using isAlive( ) and join( )

- The isAlive() method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.
- join() method waits until the thread on which it is called terminates.

Using isAlive( ) and join( )

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }
```

Using isAlive( ) and join( )

```
// This is the entry point for thread.
 public void run() {
   try {
     for(int i = 5; i > 0; i--) {
       System.out.println(name + ": " + i);
       Thread.sleep(1000);
     }
   } catch (InterruptedException e) {
     System.out.println(name + " interrupted.");
   }
   System.out.println(name + " exiting.");
 }
}
```

Using isAlive( ) and join( )

```
class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");

    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());
```

Using isAlive( ) and join( )

```
// wait for threads to finish
   try {
     System.out.println("Waiting for threads to finish.");
     ob1.t.join();
     ob2.t.join();
     ob3.t.join();
   } catch (InterruptedException e) {
     System.out.println("Main thread Interrupted");
   }
```

Using isAlive( ) and join( )

```
System.out.println("Thread One is alive: "
                   + ob1.t.isAlive());
   System.out.println("Thread Two is alive: "
                   + ob2.t.isAlive());
   System.out.println("Thread Three is alive: "
                   + ob3.t.isAlive());

   System.out.println("Main thread exiting.");
  }
}
```

Using isAlive( ) and join( )

- Output

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
```

Using isAlive( ) and join( )

```
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
```

Using isAlive( ) and join( )

```
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
  - **MIN_PRIORITY is 1**
  - **MAX_PRIORITY is 10**
  - **NORM_PRIORITY is 5**

Thread Priorities

```
// Demonstrate thread priorities.
class clicker implements Runnable {
  int click = 0;
  Thread t;
  private volatile boolean running = true;
  public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
  }
  public void run() {
    while (running) {
      click++;
    }
  }
```

Thread Priorities

```
 public void stop() {
    running = false;
  }

  public void start() {
    t.start();
  }
}
```

Thread Priorities

```
class HiLoPri {
  public static void main(String args[]) {
    Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
    clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
    clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

    lo.start();
    hi.start();
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
    }
```

Thread Priorities

```
    lo.stop();
    hi.stop();
    // Wait for child threads to terminate.
    try {
      hi.t.join();
      lo.t.join();
    } catch (InterruptedException e) {
      System.out.println("InterruptedException caught");
    }
    System.out.println("Low-priority thread: " + lo.click);
    System.out.println("High-priority thread: " + hi.click);
  }
}
```

Thread Priorities

- Output

```
Low-priority thread: 4408112
High-priority thread: 589626904
```

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.
- Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time.
- Languages like C and C++ do not, themselves, support synchronization. Instead, to synchronize threads, your programs need to utilize operating system primitives. Fortunately, because Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated.

Synchronization

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.

```
// This program is not synchronized.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}
```

Synchronization

```
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    target.call(msg);
  }
}
```

Synchronization

```
class Synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");
    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

Synchronization

- Output

```
 [Hello[Synchronized[World]
]
]
```

- To avoid this problem you have to modify the program

class Callme {

synchronized void call(String msg)

- Output

```
[Hello]
[Synchronized]
[World]
```

Synchronization

- The earlier method will not work in all cases.
- When a class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized?
- Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

synchronized(object) (

/ / statements to be synchronized

}

Synchronization

```
// This program uses a synchronized block.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}
```

Synchronization

```
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }
  // synchronize calls to call()
  public void run() {
    synchronized(target) { // synchronized block
      target.call(msg);
    }
  }
}
```

Synchronization

```
class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");
    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

Interthread Communication

- As you know, Polling is usually implemented by a loop that is used to check some condition repeatedly.
- This wastes CPU time.
- To avoid polling, Java includes an elegant interprocess communication mechanism via the wait(), notify(), and notifyAll() methods.
- All classes have them.
- All three methods can be called only from within a synchronized method.

Interthread Communication

- wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
- notify() wakes up the first thread that called wait() on the same object.
- notifyAll() wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

Interthread Communication

```
// An incorrect implementation of a producer and consumer.
class Q {
  int n;
  synchronized int get() {
    System.out.println("Got: " + n);
    return n;
  }
  synchronized void put(int n) {
    this.n = n;
    System.out.println("Put: " + n);
  }
}
```

Interthread Communication

```
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }
  public void run() {
    int i = 0;
    while(true) {
      q.put(i++);
    }
  }
}
```

Interthread Communication

```
class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}
```

Interthread Communication

```
class PC {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

Interthread Communication

- Output

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

Interthread Communication

```
// A correct implementation of a producer and consumer.
class Q {
  int n;
  boolean valueSet = false;
  synchronized int get() {
    if(!valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }
      System.out.println("Got: " + n);
      valueSet = false;
      notify();
      return n;
  }
```

Interthread Communication

```
synchronized void put(int n) {
    if(valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }

      this.n = n;
      valueSet = true;
      System.out.println("Put: " + n);
      notify();
  }
}
```

Interthread Communication

```
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }
  public void run() {
    int i = 0;
    while(true) {
      q.put(i++);
    }
  }
}
```

Interthread Communication

```
class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}
```

Interthread Communication

```
class PCFixed {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

Interthread Communication

- Output

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
Put: 6
Got: 6
Put: 7
Got: 7
```

Deadlock

```
// An example of deadlock.
class A {
  synchronized void foo(B b) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " entered A.foo");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("A Interrupted");
    }
    System.out.println(name + " trying to call B.last()");
    b.last();
  }
  synchronized void last() {
    System.out.println("Inside A.last");
  }
}
```

Deadlock

```
class B {
  synchronized void bar(A a) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " entered B.bar");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("B Interrupted");
    }
    System.out.println(name + " trying to call A.last()");
    a.last();
  }
  synchronized void last() {
    System.out.println("Inside A.last");
  }
}
```

Deadlock

```
class Deadlock implements Runnable {
  A a = new A();
  B b = new B();
  Deadlock() {
    Thread.currentThread().setName("MainThread");
    Thread t = new Thread(this, "RacingThread");
    t.start();
    a.foo(b); // get lock on a in this thread.
    System.out.println("Back in main thread");
  }
  public void run() {
    b.bar(a); // get lock on b in other thread.
    System.out.println("Back in other thread");
  }
  public static void main(String args[]) {
    new Deadlock();
  }
```

Deadlock

- Output

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

End of Chapter 11

Questions?