***Chapter 26***

## *Multithreaded Programming*

* A multithreaded program contains two or more parts that can run concurrently.
* Each part of the program is called a *thread*.
* Each thread defines a separate path of execution.

*Thus, multithreading is a specialized form of multitasking.*

There are two distinct forms of multitasking:

* *process based*: A process is a program that is executing. This type of multitasking allows your program to run multiple programs concurrently.
* *thread based*: The thread is a small unit of code that can be executed. A single program can dispatch two or more threads that can be executed concurrently.

*Process based* multitasking deals with the "big picture" where *thread based* multitasking handles the details.

Multitasking threads require less overhead than multitasking processes. Processes require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is costly. Threads are lightweight in comparison. They share the same address space and cooperatively share the same process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.

Multithreading allows you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

For example: In a single threaded machine, the system must wait for individual tasks to complete before moving on to the next task. This can be quite costly when the task involves waiting for user input, or writing to the printer, since the CPU sits idle during most of these processes. Multithreading enables you to gain access to this idle time.

*The Java Thread*

Threading is integrated throughout the Java system enabling the entire environment to be asynchronous. All of the class libraries incorporate multithreading.

The non-Java single-threaded system: uses an approach called an *event loop* with *polling*. Here, a single thread of control runs in an infinite loop, polling a single *event queue* to decide what to do next. If the *queue* returns information indicating that an event has occurred (let's say data is ready to be read), then the appropriate event handler is activated. While the event handler is working, nothing else can happen in the system. If the event handler needs to wait, the system must wait. Also, this kind of process can result in one part of a program dominating the processor, thereby preventing other events from being processed.

The Java Multithreaded system: eliminates the main loop/polling mechanism. One thread can pause without stopping the other parts of your program. With multithreading, when an event handler must wait, it enters into a pause state, relinquishing control of the processor so that other events may be handled. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.

Threads can exist in several states*: running, ready to run, suspended,* and *blocked.*

Java assigns *priorities* to each thread, by assigning an integer value that specifies the relative priority of one thread to another. A thread's priority is used to determine which thread to activate when multiple threads *are ready to run*. The activation of a thread is referred to as *context switching*. The rules that determine when a context switch occurs are simple:

* A thread can voluntarily relinquish control by explicitly yielding, sleeping or blocking on pending I/O. Here, all other threads are examined. The thread with the highest priority and is *ready to run* is activated.
* A thread can be preempted by a higher-priority thread. In this case, a low-priority thread that does not yield the CPU is preempted (or deactivated) no matter what it is doing by a higher-priority thread. Therefore, higher-priority threads are given the CPU when *ready to run*. This is called *preemptive multitasking*.

When threads have the same priority, different schemes are used (depending on the operating system you are running). With Windows 95, threads of equal priority are time-sliced in a round-robin fashion. Some operating systems allow threads of equal priority to voluntarily relinquish control to their peers. If they don't, the other threads don't run. OOPS!

*Synchronization*

If you want two threads to communicate and share data structures, you need some way to ensure that they don't conflict with each other (i.e., you don't want one thread to be writing data while another thread is in the middle of reading it). In Java, the *monitor* (first defined by Hoare) manages this kind of synchronization. You can think of it as a small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until it exits, thereby protecting shared assets from being manipulated by more that one thread simultaneously.

*The Thread Class & Runnable Interface*

The Thread class defines the following methods:

|  |  |
| --- | --- |
| **Method** | **Meaning** |
| getName | Obtain a thread's name |
| getPriority | Obtain a thread's priority |
| isAlive | Determines if a thread is still running |
| join | Wait for a thread to terminate |
| resume | Resume execution of a suspended thread |
| run | Entry point for the thread |
| sleep | Suspend a thread for a period of time |
| start | Start a thread by calling its run method |
| suspend | Suspend a thread |

*Main Thread*

* Executed when your program begins
* The thread from which child threads are spawned
* Must be the last thread to finish execution
* Can be controlled through a Thread object by calling **currentThread()**

static Thread currentThread( )

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("Coach's Thread");

System.out.println("After name change: " + t);

**sleep** might throw exception

if some other thread wanted

to interrupt this sleeping

one. This requires a **catch**.

In a real program, you

would want to handle this

kind of exception differently

try {

for (int n = 5; n > 0; n--) {

System.out.println(n);

Thread.sleep(1000);

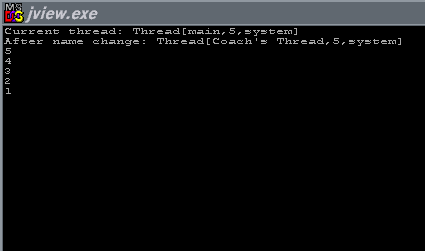
}

} catch (InterruptedException e) {

System.out.println("Main thread interrupted");

}

}

}

Notice that, by default,

* the name of the main thread is **main**,
* the priority is 5, and
* **system** is the name of the *thread group*.

A *thread group* is a data structure that controls the state of a collection of threads as a whole.

*Implementing Runnable*

You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class must implement a single abstract method called **Run( )**:

public abstract void run( )

Remember, the programmer never intends to instantiate an *abstract* class. An *abstract* class is used as a superclass in inheritance situations.

Inside **run( )**, you define the code that constitutes the new thread. **run( )** can call other methods, use other classes and declare variables just like main. The only difference is that **run( )** establishes an entry point for another, concurrent thread of execution within your program. The thread will end when a return from **run** occurs.

After creating a class that implements **Runnable**, an object of type Thread must be instantiated.

Thread(Runnable *threadOb*, String *threadName*)

Next, start the Thread by calling its **start( )** method. In essence, **start( )** executes a call to **run( )**.

The following example creates a new thread and starts it running.

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

}

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.");

Note: the main thread must

be the last thread to finish

running or else the Java

run-times system may "hang."

}

}

class ThreadDemo {

public static void main(String args[ ]) {

new NewThread( );

try {

for (int i = 5; i>0; i--) {

System.out.println("Main Thread: " + i);

Thread.sleep(1000);

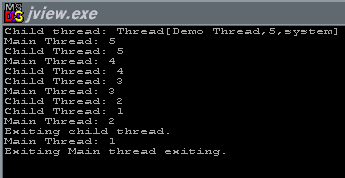
}

} catch (InterruptedException e) {

System.out.println("Main thread interrupted."); }

System.out.println("Exiting Main thread exiting.");

}

}

*Extending Thread*

The 2nd way to create a thread is to create a new class that extends Thread and then to create an instance of that class. The extending class must override the **run( )** method. It must also call **start( )** to begin execution of the new thread.

class NewThread **extends Thread** {

Thread t;

NewThread ( ) {

*t = new Thread(this, "Demo Thread");* previously.

This statement calls the superclass' constructor,

which instantiates a thread named "Demo Thread."

// create a new, second thread

super("Demo Thread");

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

**start( );** // Start the thread

*t.start( );* previously.

Uses the inherited

**start** method

}

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.");

}

}

class ThreadDemo {

public static void main(String args[ ]) {

new NewThread( );

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("Exiting Main thread exiting.");

}

}

*Which Way Do I Do It?*

So, which approach is better? The 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( )**. Many Java programmers feel that classes should only be extended 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 to implement **Runnable**.

*Creating Multiple Threads*

So far, we've seen two threads: the main and one child thread. However, an applet can spawn as many threads as it needs. The following example creates three child threads:

class NewThread implements Runnable {

String name;

Thread t;

NewThread(String threadname) {

name = threadname;

t = new Thread(this, name);

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

t.start( );

}

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.");

}

}

class MultiThreadDemo {

public static void main(String args [ ]) {

new NewThread("One");

new NewThread("Two");

new NewThread("Three");

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

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

}

}

