Introduction to Computer Science with Java

Chapter 22 – Parallel Computing in Java

This chapter is a brief introduction to concepts related to parallel computing, and some Java features used for parallel computing. The development and implementation of parallel algorithms is beyond the scope of this course, but the chapter provides a good introduction to the topic in Java.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the concept of an *parallel computing* and the difference between *parallel processing* and *time-sharing multi-tasking*, also known as *preemptive multi-tasking*.
- describe the difference between a *process* and a *thread*.
- view the processes running on a *Microsoft Windows* system and on an *Apple OS X* system.
- describe the difference between *I/O intensive* and *processing intensive* software and how multithreading can speed up both types of processing.
- list and describe the steps in the *Java Thread Life Cycle*.
- describe how thread priorities are used in Java.
- create Java code that creates multiple threads by implementing the *Runnable Interface*.
- create Java code that creates multiple threads by extending the *Thread Class*.
- describe the importance of *thread synchronization* and several things that can go wrong if threads are not synchronized.
- describe what Java's *fork/join framework* is and how it can be used to implement parallel processing.
- describe the concept and appropriate use of a *hybrid algorithm*.

22.1 Parallel Computing and Parallel Programming

Parallel computing, also known as **Concurrent Computing** is the ability of a single computer to do more than one thing at the same time. **Parallel programming** is the development of computer programs that engage in parallel computing – in other words, writing programs that can do more than one thing at the same time. Programming languages, operating systems, and system hardware all need to be designed for parallel computing for parallel programming to work on a specific computing platform.

The terminology and concepts of parallel computing can be quite complex, and take some time to master. We will briefly examine the topic in this section, before discussing some features of the java programming language that can be used for parallel programming.

Computers execute algorithms in the form of instruction sets which are loosely referred to as programs. The terms *process*, *thread*, and *task* all refer in some way to a program being executed.

A **process** is a program being executed by a computer, along with the memory and access to system resources needed to run the program, such as maintaining an instruction pointer for the program, keeping track of variables, providing access to a file system, and so on. The exact definition and nature of a process may vary slightly from system to system.

A **thread** is a single flow of control within a process, which may be executed independently of other threads in the process. Threads share the same memory space as other threads in the same process. They are often referred to as "*lightweight processes*". Every Java program has at least one thread – most often a single thread automatically started when a new process begins. Every Java application runs on the JVM as a thread or set of threads. When a new Java application is started, the JVM begins to run the application's *main()* method as a thread.

Almost all of the programs in this textbook so far have been composed of a single thread. **Multithreaded programming**, discussed later in this chapter, is one form of parallel computing that allows a process to be broken down into threads, some of which can be executed at the same time. Java is a multi-threaded programming language. Multithreaded programming can be used to speed up a program and to better manage the use of system resources within a program.

The term **task** is less well-defined than *process* or *thread*. It loosely refers to a job being executed on a computer, which in most instances has the same meaning as process.

Multitasking vs. Parallel Processing

Parallel computing may be actual **parallel processing**, in which multiple processing units in the computer execute processes simultaneously, or **time-sharing multi-tasking**, in which a single processor rapidly switches back and forth between several tasks, appearing as if it is performing parallel processing. True parallel processing requires multiple processing units or a processing unit with multiple cores that each function as a separate processing unit.

The operating system is critical in parallel computing. A multitasking operating system, which can schedule processing time and resources for different processes or different threads, is required for either multitasking or true parallel processing. An operating system capable of scheduling and coordinating multiple processing units is also required for true parallel processing.

Multitasking

Time-sharing multitasking is also known as **preemptive multi-tasking** because one task (a thread or a process) can interrupt, or preempt, another process to make the best use of system resources or because one task has priority over another.

To better understand how time-sharing multitasking works, let us consider a simple question: *How many games of chess can a person play at one time?*

As of January 2014, the best chess player in the world, according to the World Chess Federation[1], is the Norwegian, Magnus Carlsen. Carlsen performs exhibitions in which he plays more than one game of chess at the same time. Chess champions before him – Bobby Fisher, Garay Kasparov, and so on – did the same thing. Bobby Fisher, for example, would play against a US high school's entire chess team at once, and would almost always win every game.

But if we watch what happens during such as exhibition, we can see that the champion is not really playing all of the opponents at once – he usually makes a move against one opponent on one board, then makes a move against the next opponent on the next board, then the next, and so on, before cycling back to the first opponent.

If we watch the games in operation, we can see that the champion plays one turn at a time, rotating from one player to next. If we watch the entire set of games as a unit, we see them all begin at one time and all are in progress – all are being played -- at the same time.

In computing, this is similar to time-sharing multitasking, in which several tasks appear to be running at the same time, but really a single processor is taking turns working on different tasks, with each turn lasting a fraction of a second. On a larger time scale – over several seconds or several minutes – it seems as if more than one task is running at the same time. Yet, at the level of the processor, the tasks are each being run one at a time, with the processor working on one task, then switching to another task, and so on for each task, before rotating back to start the cycle again with the first task.

Multi-user computer systems that used time-sharing multitasking were common in the days before personal computers became widely used. A school, for example, would have several dumb terminals to a single computer. Each terminal had a keyboard and a screen, but no CPU. Students could sit at each terminal and work on programs at the same time on a central computer. Within the computer, however, the program from the first terminal would be run for a few hundredths of a second, then the program from the next terminal would be run for a few hundredths of a seconds, and so on. If there were eight terminals, then each terminal would be using 1/8 of the processor's time, minus some overhead for the system. The first computer terminals used by bank tellers, school registration clerks, or any similar office with multiple terminals to a single system worked this way.

Today, each student, teller, or clerk in such an environment would most likely have an individual computer, communicating over a network with a centralized database management system.

Time-sharing multitasking is one way in which a computer with a single CPU can process more than one task at a time. A user on a single processor system can be running programs to play music, browse the web, read email, and edit a word processing document all at the same time. At least, it looks to the user as if they are all running at the same time.

[1], known as *FIDE* from its French acronym, *Federation Internationale des Echecs.* See: http://ratings.fide.com/top.phtml

Parallel Processing

Many currently available computers, even relatively inexpensive personal computers, go further. They actually do have more than one processor. Intel Core i3, Core i5, and Core i7 processors, for example can have up to six processors – each called a core – in a single CPU chip. AMD produces similar chips. The inside of each core in such processors has a pipelined architecture that can overlap different steps of the machine execution cycle for each instruction, further multiplying the parallel processing capabilities of the system.

The internal hardware of computer systems is studied in *CSCI 213 – Computer Organization*. The bottom line for us as programmers is this – most modern computers are capable of real parallel processing, not just old-fashioned time-sharing multitasking. In fact, modern computers can handle hundreds, sometimes thousands, of tasks at the same time by using both real parallel processing and time-sharing multitasking.

22.2 Viewing Processor Information and System Processes

The Microsoft Windows operating system has a *Task Manager* that will show us the processes running on a computer. Apple OS X's *Activity Monitor* has a similar feature. The two short exercises in this section show you how see what processes are running on a Windows-based computer and on an a computer running OS X.

Exercise – Viewing Process and System Information on a Windows System

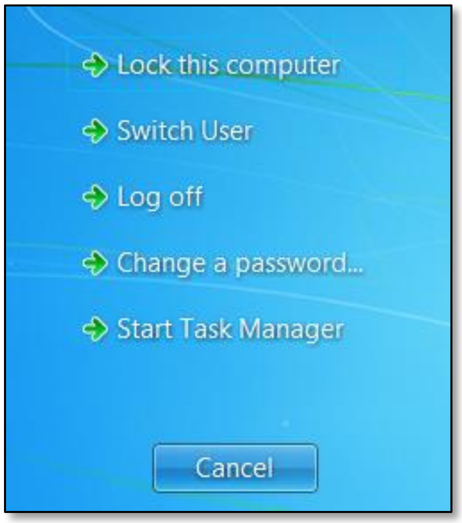
We can view information about the processes running under the Microsoft Windows operating system by using the *Windows Task Manager*, as follows:

STEP 1.

On a computer that is running the Windows operating system, press the **[Ctrl]** **[Alt]** and **[Delete]** keys at the same time. An easy way to do this is by holding down the **[Ctrl]** and **[Alt]** keys and then press the **[Delete]** key.

You can also open Task Manager by right-clicking an empty area on the **taskbar** and selecting **Start Task Manager** from the menu that appears, or by pressing **[Ctrl]** **[Shift]** **and** **[Esc]**.

A system menu similar to the one shown here will appear.



STEP 2.

Select **Start Task Manager** from the menu that appears. When the task manager screen appears, select the **Processes** tab, shown below.

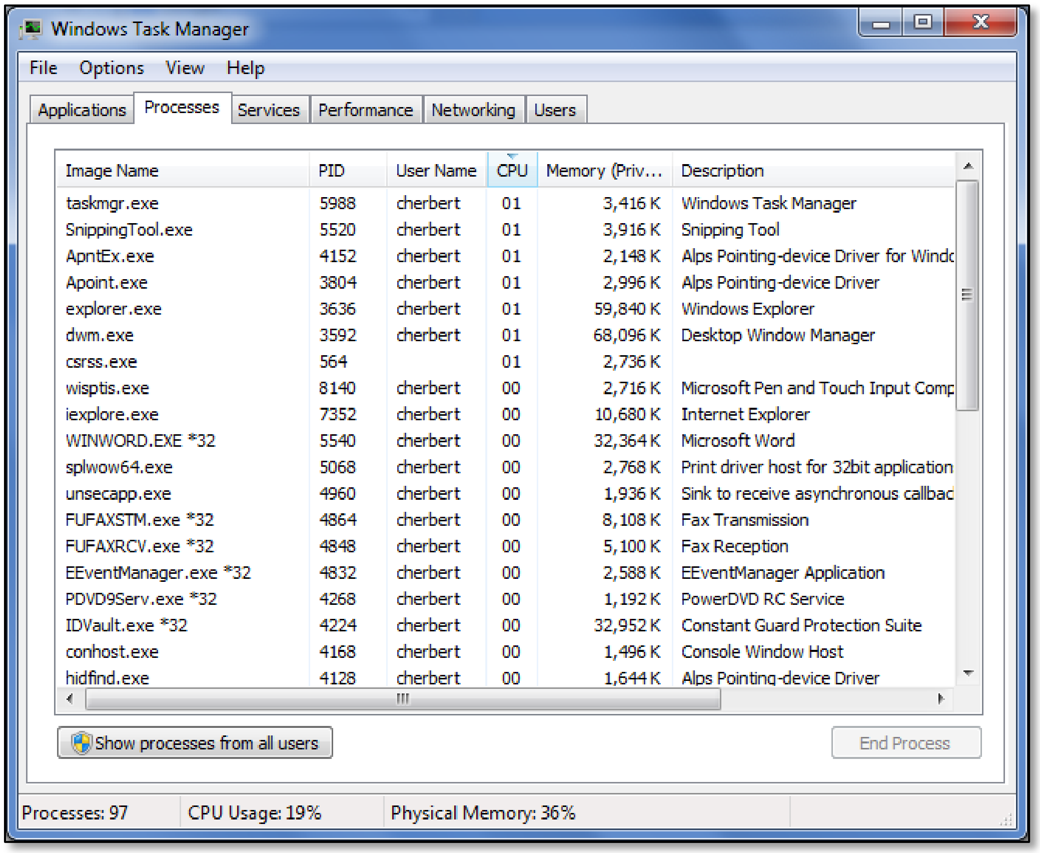

Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Show processes from all users
End Process

Image Name	PID	User Name	CPU	Memory (Priv...	Description
taskmgr.exe	5988	cherbert	01	3,416 K	Windows Task Manager
SnippingTool.exe	5520	cherbert	01	3,916 K	Snipping Tool
ApntEx.exe	4152	cherbert	01	2,148 K	Alps Pointing-device Driver for Windc
Apoint.exe	3804	cherbert	01	2,996 K	Alps Pointing-device Driver
explorer.exe	3636	cherbert	01	59,840 K	Windows Explorer
dwm.exe	3592	cherbert	01	68,096 K	Desktop Window Manager
csrss.exe	564		01	2,736 K	
wisptis.exe	8140	cherbert	00	2,716 K	Microsoft Pen and Touch Input Comp
iexplore.exe	7352	cherbert	00	10,680 K	Internet Explorer
WINWORD.EXE *32	5540	cherbert	00	32,364 K	Microsoft Word
splwow64.exe	5068	cherbert	00	2,768 K	Print driver host for 32bit application
unsecapp.exe	4960	cherbert	00	1,936 K	Sink to receive asynchronous callbac
FUFAXSTM.exe *32	4864	cherbert	00	8,108 K	Fax Transmission
FUFAXRCV.exe *32	4848	cherbert	00	5,100 K	Fax Reception
EEventManager.exe *32	4832	cherbert	00	2,588 K	EEventManager Application
PDVD9Serv.exe *32	4268	cherbert	00	1,192 K	PowerDVD RC Service
IDVault.exe *32	4224	cherbert	00	32,952 K	Constant Guard Protection Suite
conhost.exe	4168	cherbert	00	1,496 K	Console Window Host
hidfind.exe	4128	cherbert	00	1,644 K	Alps Pointing-device Driver

Processes: 97 CPU Usage: 19% Physical Memory: 36%

The processes tab shows the processes currently running on the system. Each process is uniquely identified by a Process ID number, shown in the *PID* column. The *image name* is a string property identifying the name of the process.

We can see some familiar programs running in the task manager window shown above, such as *Microsoft Word* whose image name is *WINWORD*. Other processes, such as *unsecapp*, are less familiar. The *"*32"* after a name indicates the computer, which in this case is a 64-bit system running 64-bit Windows, is running a 32-bit process.

At the bottom of the task manager window we can see that when this image was captured there were 97 processes running, using the CPU to 19% of its capacity and consuming 36% of the available system memory.

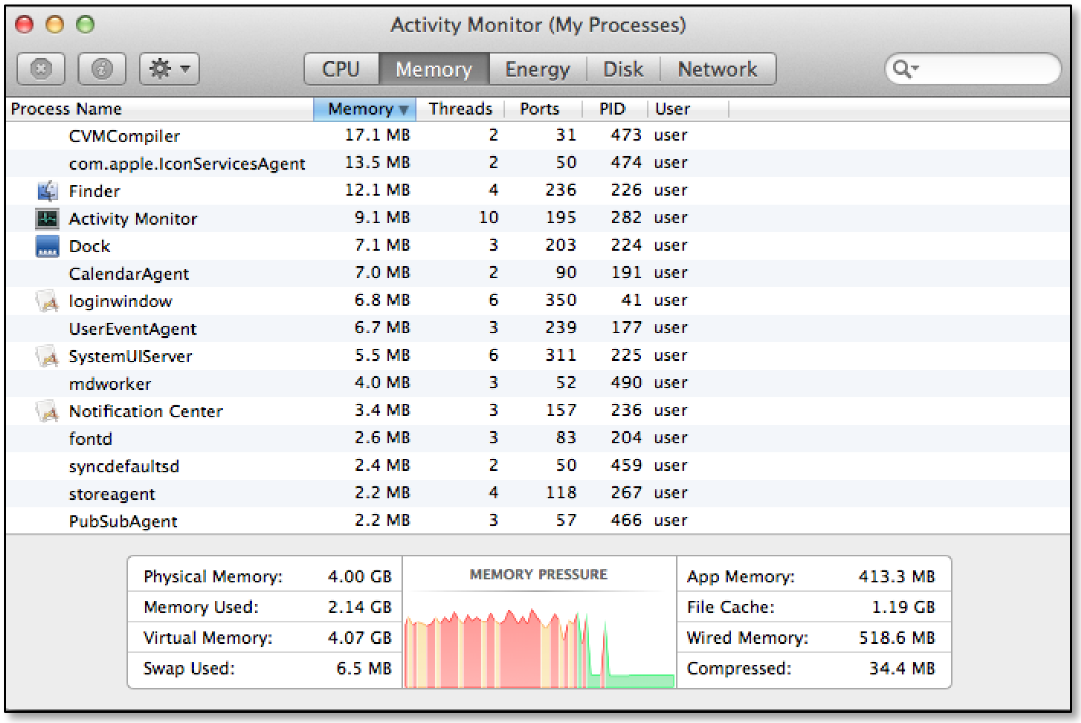
The CPU column shows us that there are at least two CPUs running on this machine, since some processes are assigned to CPU 1 and some are assigned to CPU 0.

Exercise – Viewing Process and System Information on an OS X System

We can view information about the processes running on Apple's OS X operating system by using the OS X Activity Monitor, as follows:

STEP 1.

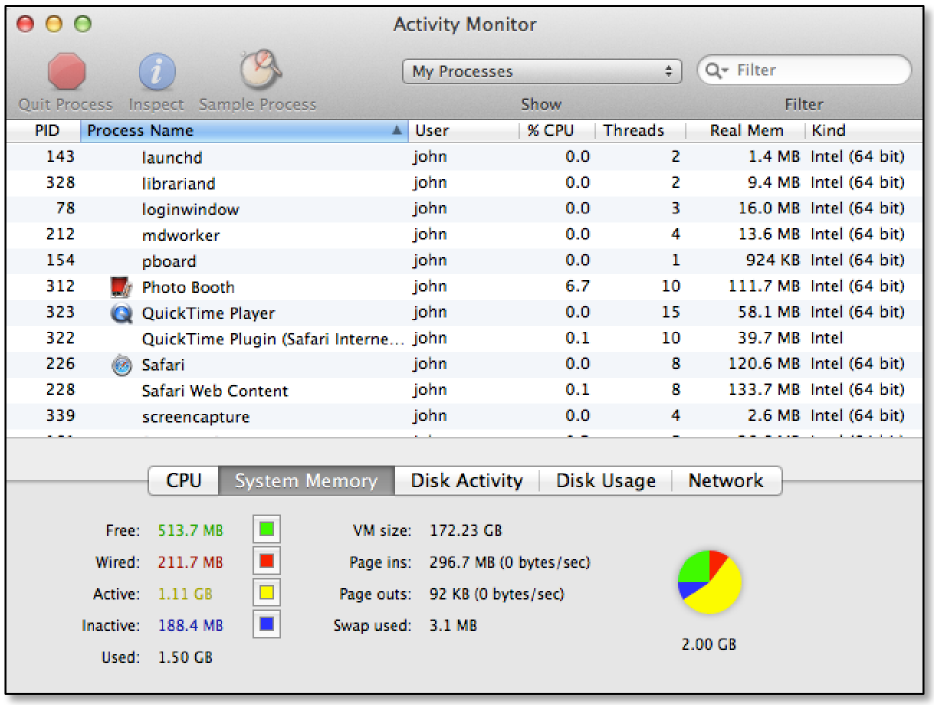
Open the Activity Monitor on a system running OS X by going to ***Applications*** →***Utilities*** → ***Activity Monitor***, or find ***Activity Monitor*** using *Spotlight*.



Process Name	Memory ▾	Threads	Ports	PID	User
CVMCompiler	17.1 MB	2	31	473	user
com.apple.IconServicesAgent	13.5 MB	2	50	474	user
Finder	12.1 MB	4	236	226	user
Activity Monitor	9.1 MB	10	195	282	user
Dock	7.1 MB	3	203	224	user
CalendarAgent	7.0 MB	2	90	191	user
loginwindow	6.8 MB	6	350	41	user
UserEventAgent	6.7 MB	3	239	177	user
SystemUIServer	5.5 MB	6	311	225	user
mdworker	4.0 MB	3	52	490	user
Notification Center	3.4 MB	3	157	236	user
fontd	2.6 MB	3	83	204	user
syncdefaultsd	2.4 MB	2	50	459	user
storeagent	2.2 MB	4	118	267	user
PubSubAgent	2.2 MB	3	57	466	user

Physical Memory:	4.00 GB	App Memory:	413.3 MB
Memory Used:	2.14 GB	File Cache:	1.19 GB
Virtual Memory:	4.07 GB	Wired Memory:	518.6 MB
Swap Used:	6.5 MB	Compressed:	34.4 MB

The activity monitor on older versions of OS X is slightly different:



PID	Process Name	User	% CPU	Threads	Real Mem	Kind
143	launchd	john	0.0	2	1.4 MB	Intel (64 bit)
328	librariand	john	0.0	2	9.4 MB	Intel (64 bit)
78	loginwindow	john	0.0	3	16.0 MB	Intel (64 bit)
212	mdworker	john	0.0	4	13.6 MB	Intel (64 bit)
154	pboard	john	0.0	1	924 KB	Intel (64 bit)
312	Photo Booth	john	6.7	10	111.7 MB	Intel (64 bit)
323	QuickTime Player	john	0.0	15	58.1 MB	Intel (64 bit)
322	QuickTime Plugin (Safari Interne…	john	0.1	10	39.7 MB	Intel
226	Safari	john	0.0	8	120.6 MB	Intel (64 bit)
228	Safari Web Content	john	0.1	8	133.7 MB	Intel (64 bit)
339	screencapture	john	0.0	4	2.6 MB	Intel (64 bit)

Free:	513.7 MB	🟩	VM size:	172.23 GB
Wired:	211.7 MB	🟥	Page ins:	296.7 MB (0 bytes/sec)
Active:	1.11 GB	🟨	Page outs:	92 KB (0 bytes/sec)
Inactive:	188.4 MB	🟦	Swap used:	3.1 MB
Used:	1.50 GB			

2.00 GB

In both cases, there is a menu bar with options for the CPU, Memory, Disk, and so on, although the options are worded slightly differently in older versions of OS X.

STEP 2.

Select the **Memory** or ***System Memory*** tab.

As you can see, the memory tab will show us information about the processes currently running, including the number of threads running in each process. As with Windows, each process has a unique *Process ID* as well as a *Process Name*. The activity monitor will also show us how many threads are running in each process, such as the 15 threads of the *QuickTime Player* running when this image was captured.

22.3 Multithreaded Programming in Java

Each thread in a multithreaded program can run concurrently, handling different task at the same time to optimize the use of system resources, including processor time or the concurrent use of multiple processors. This provides several advantages over programs that only have one thread of execution. Programs are either **I/O intensive**, in which the movement of data between memory and I/O devices or external storage devices dominates the program, or **processing intensive**, in which numerical calculations or similar operations consume much of the program's running time. In either case, multithreaded programming can speed up a program, as described in some of these examples:

- A portion of a program that might have to wait for an external device, such as waiting for user input or waiting for a printer to be free, can be in its own thread. While this thread is in a waiting state the processor can be executing other threads. Multithreaded programs make better use of CPU time than single thread programs, even on systems with just one processor.

- Imagine a program that has two threads doing different things that can be done at the same time, such as reading data into arrays from two different data sources. Both can be done in separate threads at the same time.

- Software data pipelining can be implemented in a multithreaded program. Consider a situation in which a program needs to play a video stored as compressed data. The data for the video needs to be read in from a file, decompressed, then played. Each of these three data processes – reading, decompressing, and playing – can be organized as threads that work on small blocks of data. The first block is read in, then decompressed, and then played. While the first block is being decompressed, a second block can be read in. When the first block is being played, the second block can be decompressed, while a third block is being read in. This pattern can continue for all of the blocks – reading, decompressing and playing consecutive blocks simultaneously.

- Multithreaded programming can be more robust (more resistant to crashing). Imagine a complicated system, such as an airplane's flight control system, that reads data from sensors while also displaying data on the screen and getting input from a pilot. it can be written as compartmentalized threads so that If part of the system in one thread fails, then the rest of the system in other threads can continue. If the display code fails and stop running in one thread, other code in separate threads continues to function.

There is system overhead for multithreaded programming, just as there is with recursion, so multithreaded is not always appropriate. If the work to be done involves very simple processing, such as incrementing each element in an array, that often a single iterative process is faster and uses fewer resources than a multithreaded program. If each of several threads must be executed sequentially, with one thread not being able to start until the previous thread finishes, then it may be better to put the code in a single thread.

The proper design of multithreaded software is beyond the scope of this course. Here our intention is just to introduce the topic. In the following section we will see how the mechanism of multithreaded programming works in java.

The Java Thread Life Cycle

Each thread is an instance of an object. There are six different states in the life cycle of a thread in Java, as described here . The state of a thread is actually an enumerated data type with the following values:[2]

- **NEW** – A thread that has been started but not yet run is in the *new* stage. The start of a thread is sometimes referred to as the *birth* of as thread and a new thread is sometimes called a *born* thread. The *start()* method for a Runnable object instantiates a thread.

- **RUNNABLE**–A runnable thread is a thread being executed by the JVM, although it may be delayed from actually running on a processor at any given moment in time. The *run()* method, whose use is described later in this section, putting a thread in a runnable state.

[2] See Oracle's documentation page for *Thread States*, online at: http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html

- **BLOCKED** – A thread that is waiting for a lock is in this state. The following quote from Oracle's Java documentation explains what this is all about:

  "Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them. A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock." [3]

  Consider threads accessing a data file, for example. One thread will acquire the File object's lock while writing to the file, and any other thread that attempts to read from or write to the file will be blocked. Once the first thread is done, it releases the lock and another thread may acquire the File object's lock and move from the *blocked* to the *runnable* state. Remember that every Java application is a thread running in the JVM. This could happen behind the scenes in any code that accesses data files, even if we don't see it as multi-threaded programming.

- **WAITING** – A thread that is waiting for another thread to perform an operation before it can operate is said to be in a *waiting* state. One thread might make another wait while it performs a critical operation that must be completed before the second thread's operation can begin, such as decompressing data that will be used by the second thread. A running thread can send a signal to run a blocked thread when it is okay for the waiting thread to proceed.

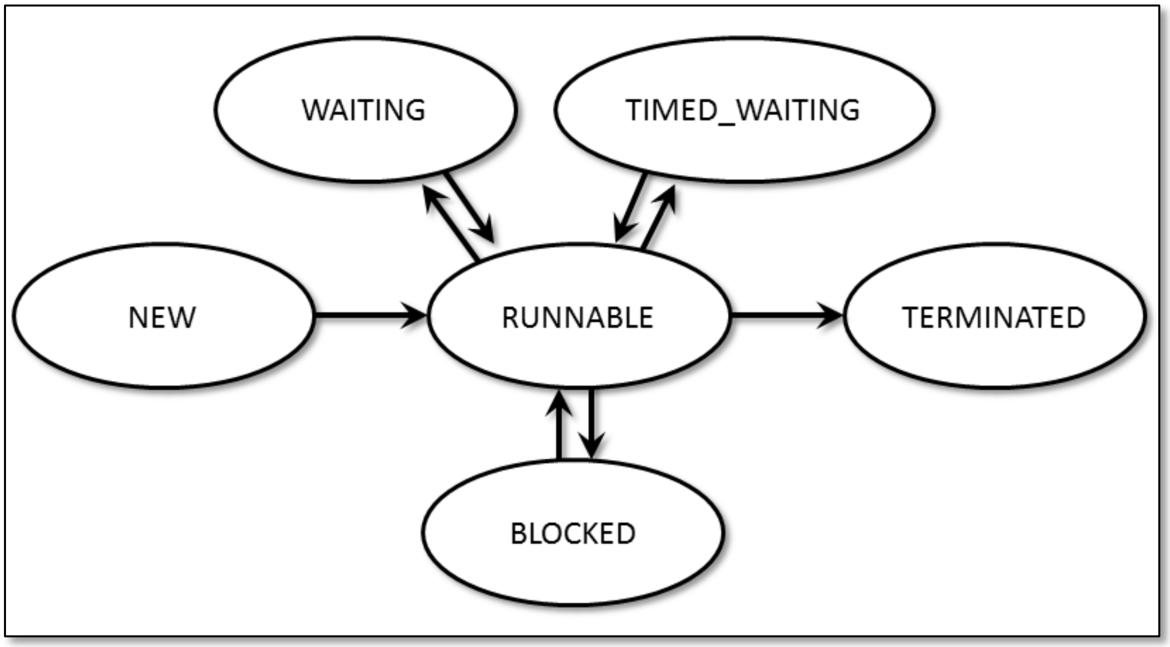
- **TIMED_WAITING** – A thread that is waiting for a specified amount of time is in the timed waiting state, which is sometimes called the *sleep* state.

- **TERMINATED** – A thread that has finished running its instruction set and has no more work to do will enter a terminated state. A thread can also be prematurely terminated as a result of internal exceptions or certain external factors. Terminated threads may not be restarted, and are sometimes referred to as a *dead* threads.

A thread can be in only one state at a given point in time. These states are Java Virtual Machine thread states, not operating system thread states. Operating system thread or process states are different from JVM thread states and may vary from system to system.

The state table below shows to which states threads can move from other states. This is sometimes referred to as the *life cycle* of a thread, moving from instantiation to eventual termination.

[3] See Oracle's Java Tutorial page about *Thread Synchronization and Locks*, online at: http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html



Thread Priorities

Threads in java have a *priority* property, which is used by the JVM and the operating system when scheduling threads to run on available processors. The priority is an *int* ranging from a low priority of 1 to a high priority of 10. Three static int constants exist to help manipulate thread priorities:

- **MAX_PRIORITY**, the maximum priority for a thread, 10 in current implementation of Java.
- **MIN_PRIORITY**, the minimum priority for a thread, 1 in current implementation of Java.
- **NORM_PRIORITY**, the minimum priority for a thread, 5 in current implementation of Java.

Two methods, getPriority() and setPriority are used to manipulate thread priorities. The default priority for new threads is NORM_PRIORITY, except that a thread started by another thread will have the same default priority as the parent thread.

Creating Threads

One thread, the main thread for a process, is created whenever an application runs on a JVM. Other threads must be created as objects of a class that implements the *Runnable Interface* and, in particular, the *run()* method. There are two commonly used techniques for programmers to do this in in a Java program:

- by creating a class for your threads that implements the ***Runnable Interface*** and defining the *run()* method in your new class.

- by creating a class for your threads that extends the ***Thread Class*,** which itself implements the *Runnable Interface,* and overriding its run() method which unless overridden does not do anything when it runs a thread and the thread dies immediately.

In either case, you will need to create a *run()* method, which will contain the code for whatever you want the thread to do. The *run()* method is a void method with no parameters. Parameters can be passed to a thread through its constructor, but not through the *run()* or *start()* methods. The run method can manipulate the properties of other objects to which it has access, but cannot return a value. The class you create for your threads can have several different constructors with different parameters. (Remember polymorphism?)

You can create several different thread classes for threads that do different things, or a single thread class for multiple threads the will perform similar operations. Several threads of the same class running at the same time are an example of symmetric parallel computing. **Symmetric parallel computing** occurs when coordinated multiple processes or threads are all performing the same task at the same time. If multiple processing units are performing the same operations at the same time in a coordinated manner, then this is an example of true *Symmetric parallel processing*. **Asymmetric parallel computing** occurs when processes or threads are performing different tasks at the same time, and true *Asymmetric parallel processing* occurs when multiple processors working in cooperation with one another are performing different operations.

The following two examples, ***ThreeThreadRunnableDemo*** and ***ThreeThreadClassDemo*** show how to implement threads, in the first example by implementing the *Runnable Interface*, them in the second example by extending the *Thread Class*. Many programmers prefer to implement the *Runnable Interface* so that they can write thread classes that inherit properties of a class other than *Thread*.

ThreeThreadRunnableDemo and ***ThreeThreadClassDemo*** are included with the files for this chapter as zipped NetBeans projects.

Each of these examples creates the *TestThread* class, a new class for threads in the program. You will notice that the methods within the *TestThread* class in the two examples are the same. Apart from one technique implementing the *Runnable Interface* and one techniques extending the *Thread Class*, both are really the same.

Programming example – Creating Threads by Implementing the *Runnable Interface*

The following code shows an example of thread creation by creating a class for threads that implements the Runnable Interface. In this simple example, three threads of the TestThread class are created: *a*, *b*, and *c*.

This code is available with the files for this chapter as the NetBeans project ***ThreeThreadRunnableDemo.***

Each thread simply counts from 1 to 3, printing the count, but with a random sleep timer in the counting loop. Each time through the loop, the thread will sleep for up to 100 milliseconds before printing the value of count. The operations of the three threads will overlap each other, as is evidenced by the output, which includes messages showing the state of each thread in addition to messages showing each thread's count.

```
/* ThreeThreadRunnableDemo.java
 * last edited Apr 12, 2014 by C. Herbert
 * This code demonstrates the creation of three threads by implementing the
 * Runnable Interface.  The class TestThread implements the Runnable Interface.
 * Three TestThread objects are instantiated in the application's main method,
 * then started, run, etc. using methods from the TestThread class for each object.
 *
 * The threads each sleep for a random time in a loop that repeats 3 times.
 * The print commands actually take longer than the rest of the processing.
 * The print commands are to demonstrate what the code is doing.
 */
package threethreadrunnabledemo;

public class ThreeThreadRunnableDemo {
    /* The main method instantiates three new threads, passing the name of the thread
     * to the TestThread constructor.
     *
     * Parameters for the thread can be passed to a constructor, to the start method,
     * or through other methods you create for the class.  Passing them to the
     * constructor is probably the most effiecient for values that are available at the
     * time the thread is instatiated.
     *
     * The run class could call other methods, including methods to spin off new threads.
     */
    public static void main(String args[])  {

        // instantiate three thread objects
        TestThread a = new TestThread("Thread a");
        TestThread b = new TestThread("Thread b");
        TestThread c = new TestThread("Thread c");

        // start the threads.
        // The start() method causes the JVM to invoke the run() method for a thread
        a.start();
        b.start();
        c.start();

    }    // end main()
}  // end class ThreeThreadRunnableDemo
//************************************************************************

// the class for the thread objects must implement Runnable
class TestThread implements Runnable {

    private Thread t;             // the thread for this TestThread object
    private String threadName;    // a name for this TestThread object

    // TestThread constructor
    TestThread(String name) {
        threadName = name;
        System.out.println("Instantiating " + threadName);
    }
// TestThread start method
    public void start() {
        System.out.println("Starting " + threadName);

        // instantiate and start a thread
            t = new Thread(this, threadName);
            t.start();

    } // end start()

    /* TestThread run() method
     * The program logic for what the thread should do goes in this method.
     *
     * In this case, a thread will count from 1 to 3, printing its count, but
     * it will sleep for a random time between 1 and 100 milliseconds each time
     * through the loop before printing its count.
     */
    public void run() {
        System.out.println("Running " + threadName);
        try {
            int i;          // a loop counter
            int sleeptime;  // the time a process should sleep in milliseconds

            // count to three, with a built in sleep delay
            for (i = 1; i <= 3; i++) {

                // Let the thread sleep for a random short time 1 to 100 milliseconds
                sleeptime = (int) (Math.random() * 100 + 1);
                Thread.sleep(sleeptime);
                System.out.println(threadName + " count = " + i);

            }// end for loop
        } catch (InterruptedException e) {
            System.out.println(threadName + " has been interrupted.");
        } // end try/catch

        System.out.println(threadName + " is now terminating.");

    } // end run()

} // TestThread class
```

The main() method instantiates and then starts three TestThread objects: a, b, and c.

The *TestThread* class is defined following the *ThreeThreadRunnableDemo* class. Its only constructor takes the TestThread's name as a parameter and prints a message indicating the constructor has been invoked and a new *TestThread* has been created.

TestThread's *start()* method will automatically cause the JVM to invoke the *run()* method for the thread; we don't need to tell it to do so, nor do we need to invoke the *run()* method directly.

The start method prints a message telling us that the thread has started, then it instantiates anew thread for this TestThread class object. It passes a reference to this TestThread class object and the object's name property to the Thread constructor in the JVM. It then tells the JVM to start the thread.

The run() method is where we put the code to make the thread do what we want it to do. In this case two local variables are declared, one a loop counter and one for the sleep timer. A loop then counts from 1 to 3, causing the thread to sleep for a random time between 1 and 100 milliseconds before printing the current count for this thread.

The last instruction in the run() method is the last thing a thread will do before terminating. In this case, it will print a message saying that the thread is about to terminate.

Programming example – Creating Threads by Extending the *Thread Class*

The following code shows the same example of thread creation as above, but this time by creating a class for threads that extends the Thread Class. This code is available with the files for this chapter as the NetBeans project ***ThreeThreadClassDemo.***

The comments above describing how the ***ThreeThreadRunnableDemo*** works apply to this code as well. You will see that the only difference in the two techniques for creating threads is that in this code for the

ThreeThreadClassDemo the *TestThread Class* extends the Thread class, whereas the *TestThread Class* in the ***ThreeThreadRunnableDemo*** implements the *Runnable Interface*. The class headers show the difference:

- in ***ThreeThreadClassDemo*** the *TestThread Class* header is:
  `class TestThread extends Thread`
- in ***ThreeThreadRunnableDemo*** the *TestThread Class* header is:
  `class TestThread implements Runnable`

Everything else in the code for the two Netbeans projects is the same.

```
/* ThreeThreadClassDemo.java
 * last edited Apr 12, 2014 by C. Herbert
 * This code demonstrates the creation of three threads by extending the Thread
 * class.  The class TestThread extends the Thread class.
 * Three TestThread objects are instantiated in the application's main method,
 * then started, run, etc. using methods from the TestThread class for each object.
 *
 * The threads each sleep for a random time in a loop that repeats 3 times.
 * The print commands actually take longer than the rest of the processing.
 * The print commands are to demonstrate what the code is doing.
 */package threethreadrunnabledemo;

public class ThreeThreadRunnableDemo {
    /* The main method instantiates three new threads, passing the name of the thread
     * to the TestThread constructor.
     *
     * Parameters for the thread can be passed to a constructor, to the start method,
     * or through other methods you create for the class.  Passing them to the
     * constructor is probably the most effiecient for values that are available at the
     * time the thread is instatiated.
     *
     * The run class could call other methods, including methods to spin off new threads.
     */
    public static void main(String args[])  {

        // instantiate three thread objects
        TestThread a = new TestThread("Thread a");
        TestThread b = new TestThread("Thread b");
        TestThread c = new TestThread("Thread c");

        // start the threads.
        // The start() method causes the JVM to invoke the run() method for a thread
        a.start();
        b.start();
        c.start();

    }    // end main()
}  // end class ThreeThreadRunnableDemo
//*************************************************************************
```

```
// the class for the thread objects extends Thread, which in turn implements runnable
class TestThread extends Thread {
    private Thread t;             // the thread for this TestThread object
    private String threadName;    // a name for this TestThread object

    // TestThread constructor
    TestThread(String name) {
        threadName = name;
        System.out.println("Instantiating " + threadName);
    }
// TestThread start method
    public void start() {
        System.out.println("Starting " + threadName);

       // instantiate and start a thread
            t = new Thread(this, threadName);
            t.start();
    } // end start()

    /* TestThread run() method
     * The program logic for what the thread should do goes in this method.
     *
     * In this case, a thread will count from 1 to 3, printing its count, but
     * it will sleep for a random time between 1 and 100 milliseconds each time
     * through the loop before printing its count.
     */
    public void run() {
        System.out.println("Running " + threadName);
        try {
            int i;          // a loop counter
            int sleeptime;  // the time a process should sleep in milliseconds

            // count to three, with a built in sleep delay
            for (i = 1; i <= 3; i++) {

                // Let the thread sleep for a random short time 1 to 100 milliseconds
                sleeptime = (int) (Math.random() * 100 + 1);
                Thread.sleep(sleeptime);
                System.out.println(threadName + " count = " + i);

            }// end for loop
        } catch (InterruptedException e) {
            System.out.println(threadName + " has been interrupted.");
        } // end try/catch

        System.out.println(threadName + " is now terminating.");

    } // end run()

} // TestThread class
```

22.4 Working with Multiple Threads

Working with multiple threads to create Java applications in which the threads need to be synchronized and scheduled by the software you create is beyond the scope of this course. Here we discuss the topic briefly and describe some of the methods, in addition to *start()* and *run()*, that can be used to work with threads.

A quote from Oracle about thread synchronization best describes some of the issues involved: [4]

> *Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors. The tool needed to prevent these errors is synchronization.*
>
> *However, synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. Starvation and livelock are forms of thread contention.*

Communication between threads is important if they are working on the same data or related data, or if the need to use the same resources. Memory inconsistency errors and classical deadlock are two examples of the tings that can go wrong in multi-threaded programming.

The term **memory consistency errors** is used to describe data that is not necessarily correct because of the sequence of thread operations. As you saw in the *ThreeThreadClassDemo* and *ThreeThreadRunnableDemo* examples in the last chapter, we can't always be sure which of the order in which threads running at the same time will execute. Data errors can occur if threads read and write data in the wrong order, such as a method to print paychecks accessing a *netPay* property before the method to calculate the netPay finishes its work.

Multiple threads trying to use the same resources can also result in a classical deadlock, in which one thread (or process) is blocking another. A **classical deadlock** in a computer system occurs when two or more processes or threads need to use two or more resources to complete a job and they each block one another. The first thread "*grabs*" the first resource it needs and locks out other threads, but before it can grab the second resource it needs another threads grabs the second resource and locks out the first process. The two threads sit locked, each waiting for the other to finish.

These are just two examples of some of the things that can go wrong with multi-threaded programming. The information in this section is enough to get you started with multi-threaded programming in Java, but a student who wishes to develop multi-threading or multi-processing software needs to study the topic for some time first. Courses in the topic are usually not taken until the junior or senior year, such as Drexel University's *CS 476 - High Performance Computing.*

[4] See the synchronization page in Oracle's java Tutorials, online at: http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

Some Useful Methods for Java Threads

whether your multithreaded software implements the Runnable Interface or extends the Thread class, it has access to methods in Java's Thread Class. Here is a summary of some of the more useful methods. For more detailed information see the Oracle Thread Class Documentation, available online at: http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

return type	Method Name	Description
String	getName()	Returns this thread's name.
int	getPriority()	Returns this thread's priority.
Thread.State	getState()	Returns the state of this thread.
ThreadGroup	getThreadGroup()	Returns the thread group to which this thread belongs.
static boolean	holdsLock(Object obj)	Returns true if and only if the current thread holds the monitor lock on the specified object.
void	interrupt()	Interrupts this thread.
boolean	isAlive()	Tests if this thread is alive.
boolean	isInterrupted()	Tests whether this thread has been interrupted.
void	join()	Waits for this thread to die.
void	notify()	Wakes up a single thread that is waiting on this object's lock.
void	notifyAll()	Wakes up all threads waiting on this object's lock.
void	run()	If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
void	setName(String name)	Changes the name of this thread to be equal to the argument name.
void	setPriority(int Priority)	Changes the priority of this thread.
static void	sleep(long millis)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
static void	sleep(long millis, int nanos)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers.
void	start()	Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
String	toString()	Returns a string representation of this thread, including the thread's name, priority, and thread group.
void	wait()	Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long millis)	Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
static void	yield()	A hint to the scheduler that the current thread is willing to yield its current use of a processor.

22.5 Parallel Processing in Java

In addition to multi-threaded programming, the java programming language has features that will enable a single Java application to run multiple processes, including the Fork/Join Framework. The Java documentation for the Fork/Join Framework tells us that: [5]

> "*the fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.*"

The *ExecutorService Interface* can be used to schedule how threads are managed and scheduled in a Java application. Its use is beyond the scope of this course, but we will see an example of what the fork/join framework can do. It is an extension of the *ExecutorService* that allows for true parallel processing.

The fork/join framework is designed to be used with recursive algorithms that can be implemented as parallel processes, which will be managed by the JVM and the operating system. The processes can run concurrently on a machine that has multiple processors or in a time-sharing multiprocessing mode on a single CPU system. Actually, multiple processor systems very often use both time-sharing multiprocessing and multiple processors for maximum efficiency, and also take advantage of other parallel features of a processor. For this to work, as stated earlier in this chapter, the hardware, the operating system, and the programming language, including the JVM, all need to be able to handle parallel processing. So, this feature of Java works best if the computer running the application has a newer operating system and a newer JVM.

The Fork/Join process includes the *ForkJoinPool Class*, which can be used to manage how the Fork/ Join Framework executes processes, including defining the number of concurrent processes the system attempts to run at one time.

There are two fork/join subclasses that can be extended to implement recursive parallel processing: *RecursiveTask,* which can return a value and *RecursiveAction*, which does not return a value.

The Netbeans project ***Sum*** is included with the files for this chapter. The code is shown below.

It uses *RecursiveTask* is used to calculate the sum of values in an array of integers. This is normally a quick process, even for very large arrays, but in this case, a 10 millisecond delay has been inserted in the code that adds the value of an element in the array to the sum. For an array of 1,000 items, this would require 10 seconds of processing time to iteratively sum the array. By using parallel recursion, the code can be completed in less than 1 second.

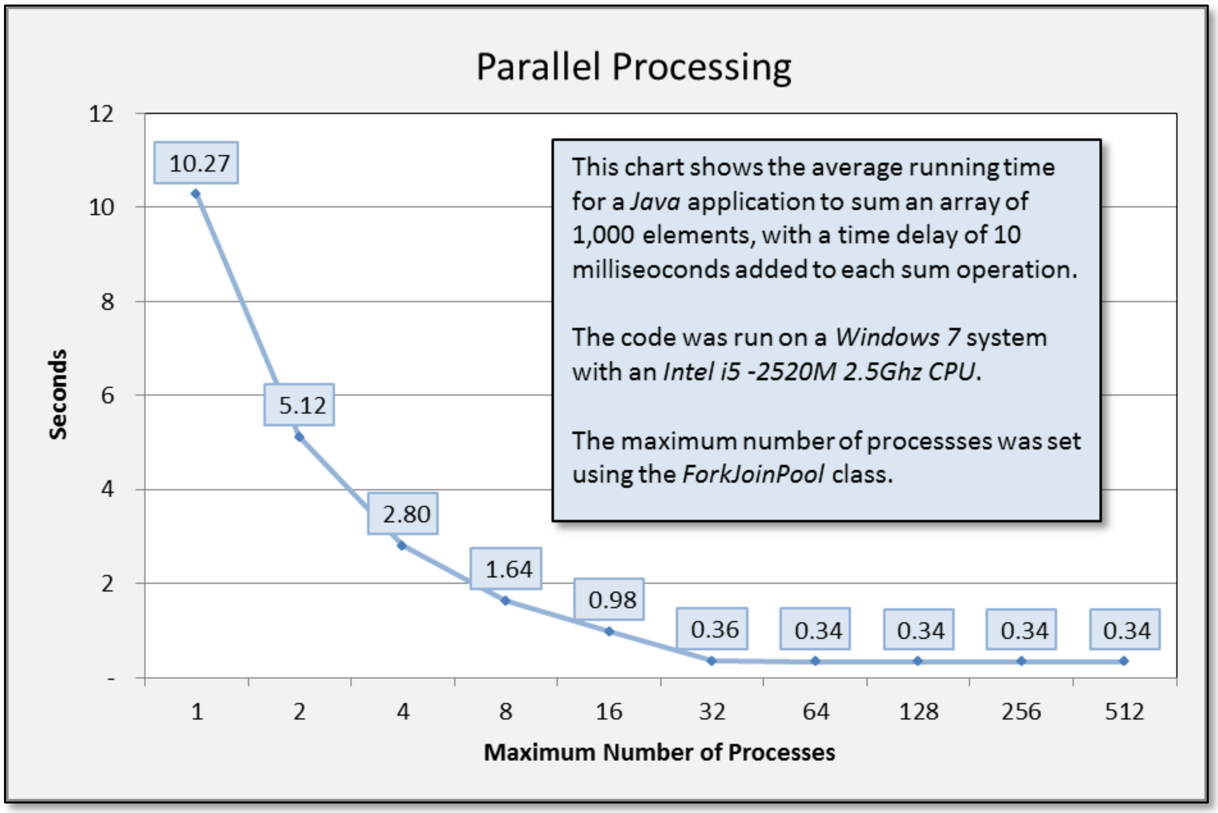
The algorithm for ***Sum*** is a *hybrid* algorithm, which is the proper way to conduct parallel processing. A **hybrid algorithm** uses recursive parallel processing for datasets above a certain size, then switches to an iterative process for data sets below that size. It does so because there is system overhead for parallel

[5] from the Oracle Java Tutorials pages about *Concurrency*, online at: http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

processing, just as there is with recursion and with multithreading. A single iterative process is faster for small datasets, and uses fewer resources than setting up and managing separate parallel processes. In the code below, the threshold for switching to an iterative process is a dataset with 50 elements. The code recursively breaks the algorithm into two parallel processes, until each process has a dataset with 50 or fewer elements, then it lets the process proceed iteratively.

```
/* Sum.Java - last edited 4/11/2014 by C. Herbert
 * This application uses the Java Fork/Join Framework to achieve true parallel
 * processing combined with time-sharing mutitasking. The purpose of the code
 * is to demonstrate parallel processing in Java.
 *
 * The code sums an array of 1,000 randomly generated intergers (1 <= a[i] <= 100)
 * with a 10 millisecond time delay added to each sum operation. The Sum class
 * extends the RecursiveTask class, recursively splits the array into parts of
 * at most 50 elements assigned to separate processes to be summed iteratively.
 *
 * It uses a ForJoinPool object to allow the user to set the maximum number of
 * allowable concurrent processes. Hence, it is a hybrid recursive/iterative algorithm.
 *
 * Without the 10 millisecond time delay, processing times are trivial,
 * even for large arrays (less than 1/100 sec. for 1 million elements).
 */

package sum;

import java.util.concurrent.*;
import java.util.Scanner;

// class for managing ForkJoinPool settings
class Globals {

    static int processes = 1;   // set default number of processes to 1
    static ForkJoinPool fjPool; // ForkJoinPool object variable
} // end class Globals
//*****************************************************************************

 class Sum extends RecursiveTask<Long> {

    // set constant to switch to iterative sequential processes at n = 50
    static final int SEQUENTIAL_THRESHOLD = 50;
    int low;        // low (left) end of dataset
    int high;       // high (right end of dataset
    long[] array;

    // Sum constructor lo and hi establish section of array for this Sum object
    Sum(long[] arr, int lo, int hi) {
        array = arr;
        low = lo;
        high = hi;
    } // end Sum constructor
    //*****************************************************************

    // the compute method is the hybrid summation algorithm
    protected Long compute() {

        // if below threshold, computer iterative sum
        if (high - low < SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            // place add a random value to the array and add it to the sum
            for (int i = low; i < high; ++i) {
                sum = sum + array[i];
                // sleep for 10 milliseconds to delay operation
                try {
                    Thread.sleep(10);
                } catch (InterruptedException ex) {
                    Thread.currentThread().interrupt();
                } // end try catch

            }  //end for
            return sum;
        } // end if

        // else perform recursion
        else {

            // find midpoint
            int mid = low + (high - low) / 2;
            // find sum of left half
            Sum left = new Sum(array, low, mid);
            // find sum of left half
            Sum right = new Sum(array, mid, high);

            //separate into different processes, then join results
            left.fork();
            long rightAns = right.compute();
            long leftAns = left.join();
            return leftAns + rightAns;
        } // end else
    } // end  compute()

    // the sumArray method ivokes processes from the pool of processes
    static long sumArray(long[] array) {
        return Globals.fjPool.invoke(new Sum(array, 0, array.length));
    }  // end sumArray()
    //*********************************************************************************

    /* The main method asks the user to set the maximum number of processes that will be
     * allowed to run concurrently.  It casn exceed the number of processors
     * because of time-sharing mutitasking as well as parallel processing.
     */
    public static void main(String[] args) {

        // variable to hold the sum of the values in the array
        long sum = 0;

        Scanner kb = new Scanner(System.in);

        System.out.println("Enter the maximum number of concurrent processes for this code:");
        Globals.processes = kb.nextInt();

        //set the maximum number of processes;
        Globals.fjPool = new ForkJoinPool(Globals.processes);

        // declare a long array and load it with random values
        long[] myArray = new long[1000];
        for (int i = 0; i < myArray.length; ++i)
                myArray[i] = (long) (Math.random() * 100 + 1);

        // get the start time in nanoseconds
        long startTime = System.nanoTime();

        // sum the array
        sum = sumArray(myArray);

        // get the end time in nanoseconds
        long endTime = System.nanoTime();

        // calculate elapsed time in nanoseconds
        long duration = endTime - startTime;
        // print the sum of the array
        System.out.printf("the sum of the values in the array is: %-,12d%n", sum);

        // print the elapsed time in seconds   (nanaoseconds/ 1 billion)
        System.out.printf("The algorithm took %12.8f seconds.%n", (double) duration /
1.0e+09);

    } // end main
} // end class Sum
```

The chart below shows the running time of the algorithm. This code was run on a Dell Latitude laptop, with an Intel Core i5 processor. It was run from *NetBeans 7.3.1* using Oracle's *HotSpot JVM* version 23.25-b01.



The time it takes to complete the algorithm is largely determined by the 10 millisecond time delay, since the sum operation without the added delay takes only a few machine cycles and can be performed by the processor in less than one millionth of a second.

The maximum number of processes is set by the user. With only a single process allowed, the algorithm takes an average of 10.27 seconds to run, which is expected: 1,000 sum operations at 10 milliseconds each would be 10 seconds. As the number of allowable processes increases, the running time decreases, roughly inversely proportional to the number of processes until it reaches an optimal value at around 40 processes.

At least 30 operations must be happening concurrently to perform 1,000 10-millisecond operations in one-third of a second. How is this possible on a machine that does not have 30 processors (or 30 cores)? A combination of true parallel processing, time sharing multiplexing, and factors linked to the architecture of the processor that are beyond the scope of what is covered in this course – such as pipelined architecture and caching results within the CPU – account for the performance. The bottom line is that Java's Fork/Join Framework allows us to take advantage of the processing power of modern computer systems capable of parallel processing.

For more about parallel computing in Java, see the Concurrency lessons in Oracle's Java Tutorials pages, online at: http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Chapter Review

Section 1 introduced some of the terminology and concepts of parallel computing. keywords included *parallel computing, concurrent Computing, parallel programming, process , thread, multithreaded programming, task , parallel processing, time-sharing multi-tasking,* and *preemptive multi-tasking.*

Section 2 included two short exercises showing how to view the processes running on Microsoft Windows based computers and Apple OS X based computers.

Section 3 discussed multithreaded programming in Java, including the Java thread life cycle and thread priorities. Key terms included: *I/O intensive, processing intensive, symmetric parallel computing,* and *asymmetric parallel computing*.

Section 4 was a short section about some of the difficulties in working with multiple threads in Java. Key terms included *memory consistency errors*, and *classical deadlock*. The section included a table of some of the Java methods that can be used to manage threads.

Section 5 discussed true parallel processing in Java with an example of Java's fork/join framework, which can be used to take advantage of a system's capabilities for parallel computing. The concept of a *recursive/iterative hybrid algorithm* was introduced.

Chapter Questions

1. What is needed for parallel programming to work on a specific computing platform?
2. What is the difference between a *process* and a *thread*? How are these terms related to the concept of a *task*?
3. What is the difference between actual parallel processing and time-sharing multi-tasking?
4. Are currently available computers capable of actual parallel processing?
5. How can the processes running on a Microsoft Windows based computer be viewed?
6. How can the processes running on an Apple OS X based computer be viewed?
7. What is the difference between I/O intensive and processing intensive software? Which can be speeded up by multithreaded programming?
8. When is multi-threaded programming not appropriate?
9. List and describe the stages in the java Thread Life Cycle.

10. What is an *intrinsic lock*? How is it related to blocked threads?
11. How are thread priorities used to manage threads in Java?
12. What is the range of values for Java thread priorities? What is the default value for a new thread?
13. How are new threads created in Java? What interface and what method are related to running a thread? How is a thread's run() method related to its start() method?
14. What is the difference between symmetric and asymmetric parallel computing? Which can be implemented with java threads?
15. What are some examples of the things that can go wrong if multiple threads are not properly synchronized?
16. Which feature of the Java programming language is used to schedule how threads are managed and scheduled in the Java programming language? Which extension of this feature allows software to take advantage of multiple processors?
17. What kind of algorithms is Java's fork/join framework designed to be used with? What subclasses does it have that allow for the implementation of this kind of parallel processing?
18. How is a hybrid algorithm related to recursion and iteration? Why are hybrid algorithms used?
19. Why are most modern computers capable of running more processes than the number of processors or processing they have?
20. Where can we find out more about parallel computing in Java?

Contents