JDK 1.1 for Solaris Developer's Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 806-3461–10
February 2000

Contents

Preface

The JDK 1.1 for Solaris Developer's Guide gives Java™ developers information about using Java in the Solaris™ 2.6, Solaris 7, and Solaris 8 environments. This information includes overviews and descriptions of the important components of Java on Solaris software, their benefits for developers, and how to use Java on Solaris software to achieve the best application performance. In addition, this document covers compatibility issues.

Who Should Use This Book

This book is intended primarily for these audiences:

- Developers who are new to Java on Solaris software
- Developers new to Java. Information for this audience is starred(*).

How This Book Is Organized

Chapter 1 is an overview of subjects covered in this book.

Chapter 2 discusses the basics of multithreading, and the benefits of using the native-threaded Java Virtual Machine (JVM) on Solaris.

Chapter 3 describes this environment with information specific to using Java on multithreaded Solaris.

Chapter 4 lists those methods that have been deprecated as of Java Development Kit (JDK™ 1.1).

Chapter 5 describes ways in which Java developers can increase their applications' performance.

Related Documentation

For up-to-date information about Java on Solaris software, refer to `http://www.sun.com/solaris/java`.

For information about Java coding style, see `http://dp-websvr.eng.sun.com/products/jpt/`.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com[SM] Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

What Typographic Conventions Mean

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls –a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Introduction to the Java Programming Environment

This developer's guide describes features of and information about application development using Java in the Solaris 2.6, Solaris 7, and Solaris 8 environments.

Note - For important information about this release of Java on Solaris software, refer to `www.sun.com/solaris/java/`.

Java Programming Environment and the Java Runtime Environment (JRE)

This section describes basic information about Java and the JRE.

What is the Java Programming Environment?

Java is a recently developed, concurrent, class-based, object-oriented programming and runtime environment, consisting of:

- A programming language
- An API specification
- A virtual machine specification

Java has the following characteristics:

- *Object oriented* – Java provides the basic object technology of C++ with some enhancements and some deletions.
- *Architecture neutral* – Java source code is compiled into architecture-independent object code. The object code is interpreted by a Java Virtual Machine (JVM) on the target architecture.
- *Portable* – Java implements additional portability standards. For example, `int`s are always 32-bit, 2's-complemented integers. User interfaces are built through an abstract window system that is readily implemented in Solaris and other operating environments.
- *Distributed* – Java contains extensive TCP/IP networking facilities. Library routines support protocols such as HyperText Transfer Protocol (HTTP) and file transfer protocol (FTP).
- *Robust* – Both the Java compiler and the Java interpreter provide extensive error checking. Java manages all dynamic memory, checks array bounds, and other exceptions.
- *Secure* – Features of C and C++ that often result in illegal memory accesses are not in the Java language. The interpreter also applies several tests to the compiled code to check for illegal code. After these tests, the compiled code causes no operand stack over- or underflows, performs no illegal data conversions, performs only legal object field accesses, and all opcode parameter types are verified as legal.
- *High performance* – Compilation of programs to an architecture independent machine-like language, results in a small efficient interpreter of Java programs. The Java environment also compiles the Java bytecode into native machine code at runtime.
- *Multithreaded* – Multithreading is built into the Java language. It can improve interactive performance by allowing operations, such as loading an image, to be performed while continuing to process user actions.
- *Dynamic* – Java does not link invoked modules until runtime.
- *Simple* – Java is similar to C++, but with most of the more complex features of C and C++ removed.

  Java does not provide:

  - Programmer-controlled dynamic memory
  - Pointer arithmetic
  - `struct`
  - `typedef`s
  - `#define`

JRE Components

The JRE is the software environment in which programs compiled for a typical JVM implementation can run. The runtime system includes:

- Code necessary to run Java programs, dynamically link native methods, manage memory, and handle exceptions
- Implementation of the JVM

The following figure shows the JRE and its components, including a typical JVM implementation's various modules and its functional position with respect to the JRE and class libraries.

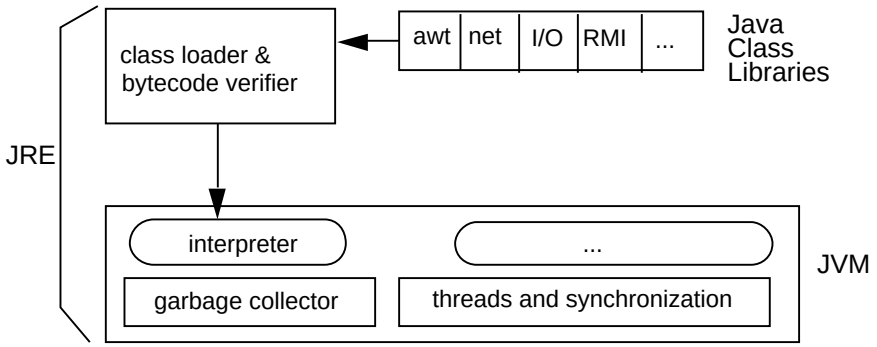

Figure 1–1 Typical JVM's Implementation: Functional Relationship to JRE and Class Libraries

JVM

The JVM is an abstract computing machine, having an instruction set that uses memory. Virtual machines are often used to implement a programming language. The JVM is the cornerstone of the Java programming language. It is responsible for Java's cross-platform portability and the small size of its compiled code.

The Solaris JVM is used to execute Java applications. The Java compiler, `javac`, outputs bytecodes and puts them into a `.class` file. The JVM then interprets these bytecodes, which can then be executed by any JVM implementation, thus providing Java's cross-platform portability. The next two figures illustrate the traditional compile-time environment and the new portable Java compile-time environment.

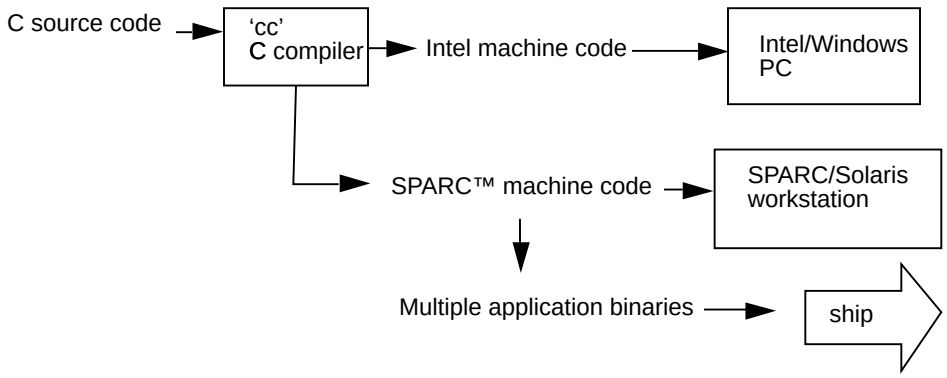

Figure 1–2 Traditional Compile-Time Environment

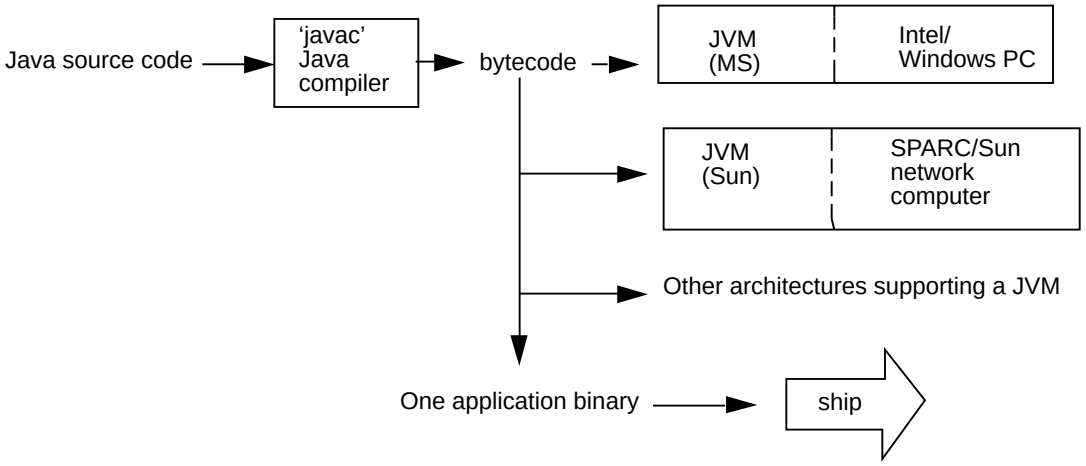

Figure 1–3 New Portable Java Compile-Time Environment

Multithreading JVM

The Java programming language requires that multithreading (MT) programs be supported (see Chapter 2). All Java interpreters provide an MT programming environment. However, many of these interpreters support only uniprocessor multithreading, so Java program threads are executed one at a time.

The Solaris JVM interpreter takes full advantage of multiprocessor systems by using the intrinsic Solaris multithread facilities. These allow multiple threads of a single process to be scheduled simultaneously onto multiple CPUs. An MT Java program run under the Solaris JVM will have a substantial increase in concurrency over the same program run on other platforms.

Sun Just-In-Time (JIT) Compiler

The Sun Java JIT compiler, an integral part of the Solaris JVM, can accelerate execution performance many times over previous levels. Long-running, compute-intensive programs show the best performance improvement.

JIT Compile Process

When the JIT compiler environment variable is on (the default), the JVM reads the `.class` file for interpretation and passes it to the JIT compiler. The JIT compiler then compiles the bytecodes into native code for the platform on which it is running. The next figure illustrates the JIT compile process.

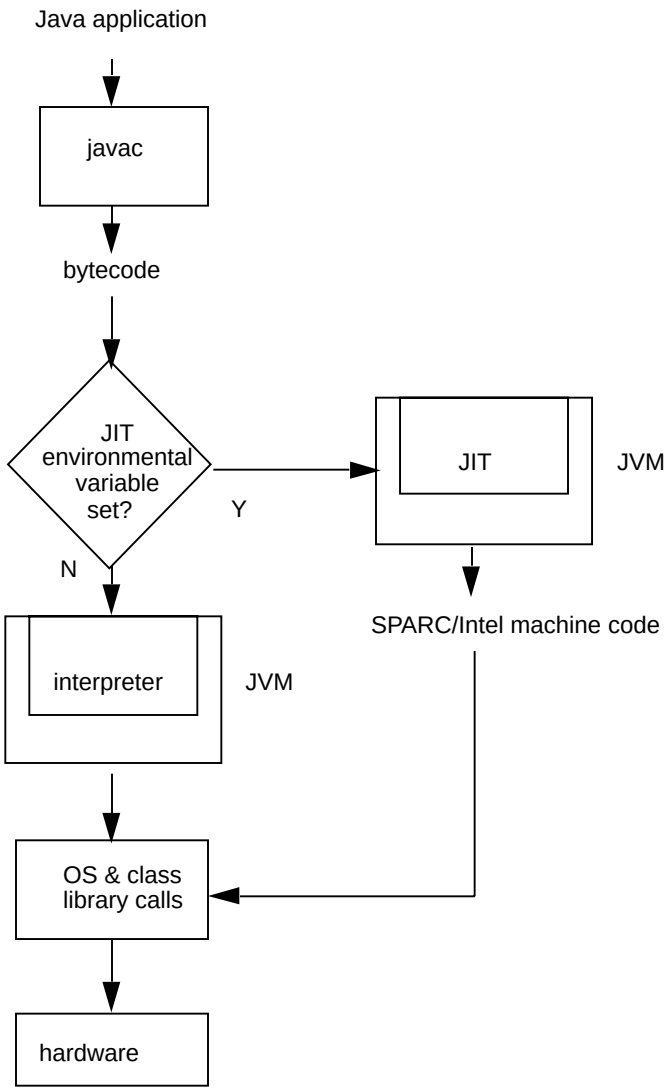

Figure 1–4 JIT Compile Process

The following figure shows the functional relationship of the JIT to the Solaris JVM and JRE.

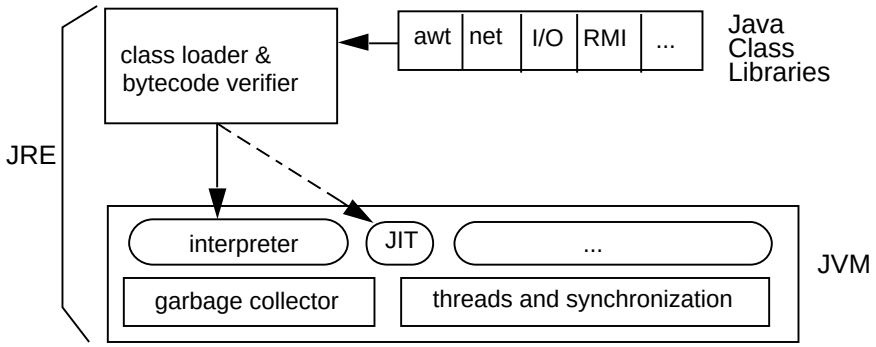

Figure 1–5 Solaris JVM Functional Relationship to the JIT Compiler

CHAPTER 2

Multithreading

This chapter discusses multithreading in general, and specific MT issues related to Java on Solaris software and the native-threaded JVM.

Information for developers new to Java is starred(*).

Definition of Multithreading*

A thread is a sequence of control within a process. A single-threaded process follows a single sequence of control while executing. An MT process has several sequences of control, thus is capable of several independent actions at the same time. When multiple processors are available, those concurrent but independent actions can take place in parallel.

Java Threads in the Solaris Environment — Earlier Releases*

Previous to Java on Solaris 2.6 software, the Java runtime used a user-level threads library called “green threads,” part of the Java runtime thread and system support layer. Because the green threads library was user-level and the Solaris system could process only one green thread at a time, Solaris handled the Java runtime as a many-to-one threading implementation (refer to “Many-to-One Model (Green Threads)” on page 19). As a result, several problems arose:

- Java applications could not interoperate with existing MT applications in the Solaris environment.
- Java threads could not run in parallel on multiprocessors.
- An MT Java application could not harness true OS concurrency for faster applications on either uniprocessors or multiprocessors.

To substantially increase application performance, the green threads library was replaced with native Solaris threads for Java on the Solaris 2.6 platform; this is carried forward on the Solaris 7 and Solaris 8 platforms.

Multithreading Concepts*

MT programming enables you to speed up applications and to leverage the parallelism of hardware and the efficiencies of objects. The Java on Solaris MT implementation is efficient, reliable, and standards-based, offering significant advantages to developers and end-users. The Solaris operating environment provides the best performance, tools, support, and flexibility in developing MT applications. The Solaris operating environment utilizes the following significant MT advances:

- The Solaris MT kernel – the essential component in a complete implementation of an MT architecture.
- The two-level threads model – the Solaris system's proven MT implementation that enables processes to use an unlimited number of threads for optimal performance
- The POSIX `pthreads` standard – an implementation of the MT interface defined by the IEEE POSIX 1003.1c specification.
- The Java threads API, one of the Java APIs, that is fast becoming a standard interface used to program MT applications

Benefits of Multithreading*

This concurrent activity speeds applications up – one of the main benefits of multithreading.

MT allows both the full exploitation of parallel hardware and the effective use of multiple processor subsystems. While MT is essential for taking advantage of the performance of symmetric multiprocessors, it also provides performance benefits on uniprocessor systems by improving the overlap of operations such as computation and I/O.

Some of the most important benefits of MT are:

- Improved throughput. Many concurrent compute operations and I/O requests within a single process.
- Simultaneous and fully symmetric use of multiple processors for computation and I/O
- Superior application responsiveness. If a request can be launched on its own thread, applications do not freeze or show the “hourglass”. An entire application will not block, or otherwise wait, pending the completion of another request.
- Improved server responsiveness. Large or complex requests or slow clients don’t block other requests for service. The overall throughput of the server is much greater.
- Minimized system resource usage. Threads impose minimal impact on system resources. Threads require less overhead to create, maintain, and manage than a traditional process.
- Program structure simplification. Threads can be used to simplify the structure of complex applications, such as server-class and multimedia applications. Simple routines can be written for each activity, making complex programs easier to design and code, and more adaptive to a wide variation in user demands.
- Better communication. Thread synchronization functions can be used to provide enhanced process-to-process communication. In addition, sharing large amounts of data through separate threads of execution within the same address space provides extremely high-bandwidth, low-latency communication between separate tasks within an application.

Multithreading Models

Most multithreading models fall into one of the following categories of threading implementation:

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

Many-to-One Model (Green Threads)

Implementations of the many-to-one model (many user threads to one kernel thread) allow the application to create any number of threads that can execute concurrently. In a many-to-one (user-level threads) implementation, all threads activity is restricted to user space. Additionally, only one thread at a time can access the kernel, so only

one schedulable entity is known to the operating system. As a result, this multithreading model provides limited concurrency and does not exploit multiprocessors. The initial implementation of Java threads on the Solaris system was many-to-one, as shown in the following figure.

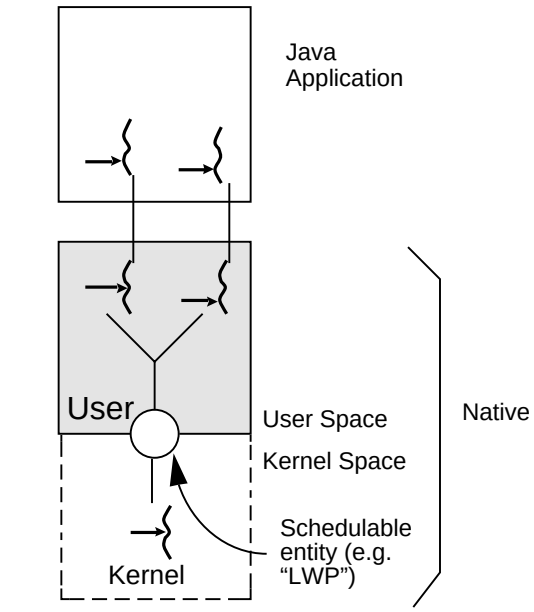

Figure 2–1 Many-to-One Multithreading Model

One-to-One Model

The one-to-one model (one user thread to one kernel thread) is among the earliest implementations of true multithreading. In this implementation, each user-level thread created by the application is known to the kernel, and all threads can access the kernel at the same time. The main problem with this model is that it places a restriction on you to be careful and frugal with threads, as each additional thread adds more “weight” to the process. Consequently, many implementations of this model, such as Windows NT and the OS/2 threads package, limit the number of threads supported on the system.

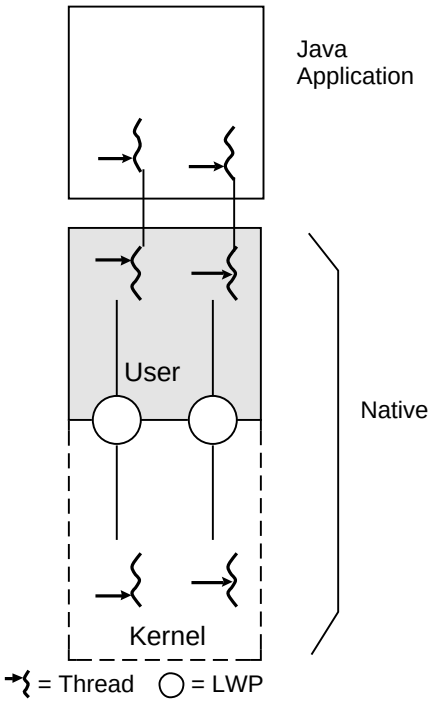

Figure 2–2 One-to-One Multithreading Model

Many-to-Many Model (Java on Solaris—Native Threads)

The many-to-many model (many user-level threads to many kernel-level threads) avoids many of the limitations of the one-to-one model, while extending multithreading capabilities even further. The many-to-many model, also called the two-level model, minimizes programming effort while reducing the cost and weight of each thread.

In the many-to-many model, a program can have as many threads as are appropriate without making the process too heavy or burdensome. In this model, a user-level threads library provides sophisticated scheduling of user-level threads above kernel threads. The kernel needs to manage only the threads that are currently active. A many-to-many implementation at the user level reduces programming effort as it lifts restrictions on the number of threads that can be effectively used in an application.

A many-to-many multithreading implementation thus provides a standard interface, a simpler programming model, and optimal performance for each process. The Java on Solaris operating environment is the first many-to-many commercial implementation of Java on an MT operating system.

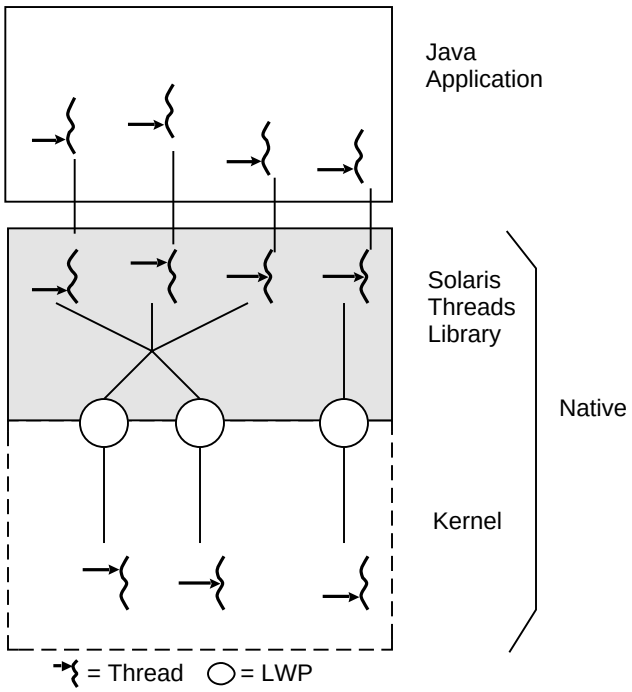

Figure 2–3 Many-to-Many Multithreading Model

Multithreading Kernel

The MT kernel is a critical foundation of a complete multithreading implementation. In an MT kernel such as the one used by the Solaris operating environment, each kernel thread is a single flow of control within the kernel's address space. The kernel threads are fully preemptive and can be scheduled by any of the available scheduling classes in the system, including the real-time class. All execution entities are built using kernel threads, which represent a fully preemptive, real-time "nucleus" within the kernel.

In addition, kernel threads employ synchronization primitives that support protocols for preventing the blocking that results in the inversion of thread and process priority. This ensures that applications execute as expected. Kernel threads also allow kernel-level tasks such as NFS daemons, pageout daemons, and interrupts to execute asynchronously, thus increasing concurrency and overall throughput.

The MT kernel is essential to building an MT application architecture, such as a typical JVM implementation:

- It is fully symmetric in order to maximize multiprocessor performance.

- With its multiple kernel threads, it enables parallelism on multiprocessor machines. This improves the efficiency of hardware subsystems by providing concurrency and parallelism in computation, networking, display, and I/O.
- Traditional (single-threaded) applications run unchanged on an MT kernel.
- It is fully preemptive, providing real-time responsiveness.

Advantages of Java Multithreading in the Solaris Environment

The Solaris MT kernel is one of the most important components of the Solaris operating environment, enabling the Solaris system to be the only standard operating environment that provides this level of concurrency, sophistication, and efficiency.

Java on Solaris software leverages the multithreading capabilities of the kernel while also enabling you to create powerful Java applications using thousands of user-level threads for multiprocessor or uniprocessor systems, through a very simple programming interface.

The Java on Solaris environment supports the many-to-many threads model. As illustrated in Figure 2–4, the Solaris two-level architecture separates the programming interface from the implementation by providing an intermediate layer, called lightweight processes (LWPs). LWPs allow you to create fast and cheap threads through a portable application-level interface. To use LWPs, write applications using threads. The runtime environment, as implemented by a threads library, multiplexes and schedules runnable threads onto "execution resources," the LWPs.

Individual LWPs operate like virtual CPUs that execute code or system calls. LWPs are dispatched separately by the kernel, according to scheduling class and priority, so they can perform independent system calls, incur independent page faults, and run in parallel on multiple processors. The threads library implements a user-level scheduler that is separate from the system scheduler. User-level threads are supported in the kernel by the kernel-schedulable LWPs. Many user threads are multiplexed on a pool of kernel LWPs.

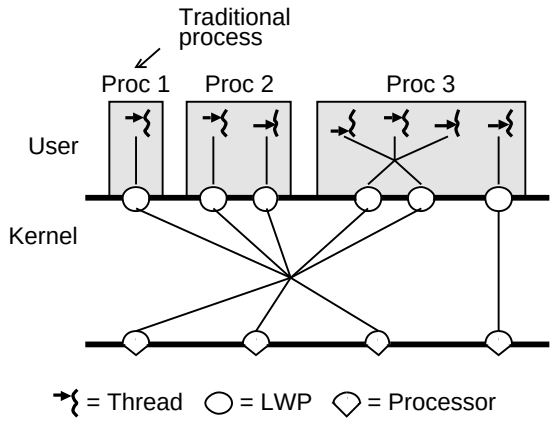

Figure 2–4 Solaris Two-level Architecture

Solaris threads provide an application with the option to bind a user-level thread to an LWP, or to keep a user-level thread unbound. Binding a user-level thread to an LWP establishes an exclusive connection between the two. Thread binding is useful to applications that need to maintain strict control over their own concurrency, such as those that require real-time response. No Java API exists to perform the binding. Most Java applications do not require binding. If binding is required, a Solaris native method call can be made to perform the binding.

Therefore, all Java threads are unbound by default. Unbound user-level threads defer control of their concurrency to the threads library, which automatically expands and shrinks the pool of LWPs to meet the demands of the application's unbound threads.

The following unique features of Java on Solaris threads are available by default to all Java applications on Solaris:

- Unbound Solaris threads: A Java thread is essentially the same as an unbound Solaris thread, with the inherent advantages of unbound threads.
- The ability to share an LWP with several user-level threads
- Automatic concurrency control for unbound threads. The threads library dynamically expands and shrinks the pool of LWPs to meet the demands of the application. See "Programming Compute-Bound, Parallellized Java Applications" on page 32 for more information.
- Extremely lightweight user threads that can be created, used, and discarded in very large numbers without consuming excessive system resources or degrading system performance
- Synchronization primitives are not known to the kernel and do not consume any system resources.
- MT features, unique to Solaris, are accessible by using native methods.

Note - In general, accessing native Solaris features using native methods from a Java application is not recommended. Such usage could make the Java application non-portable, because it would not be 100% Pure Java™ and would be tied to the Solaris platform only.

Though accessing Solaris-specific features from Java applications is not recommended, here is a list of those features to illustrate the richness of the Solaris MT architecture:

- The ability to define bound or unbound threads for user or system level control of application concurrency. Note that a bound Java thread can be created only by way of native methods.
- In addition, the application can control application concurrency through a programmatic interface. See "Programming Compute-Bound, Parallellized Java Applications" on page 32 for more information.
- The ability to bind a user-level thread (through native methods) to an LWP that is dedicated to a single processor. This feature is useful to real-time applications running on multiprocessor systems.
- Synchronization primitives that have interprocess scopes
- Synchronization primitives that can be placed in files and can have lifetimes beyond that of the creating thread.
- Direct native support for Java's daemon threads. Daemon threads are threads that run in the background and have dedicated exit semantics enabling them to terminate independently of the processes that use them. Daemon threads are useful to libraries that need to create threads that are unknown to applications. The Solaris JVM does not utilize direct native support for Java's daemon threads, but might do so eventually.

The Solaris two-level model delivers unprecedented high levels of flexibility for meeting many different programming requirements. Certain programs, such as window programs, demand heavy logical parallelism. Other programs, such as matrix multiplication applications, must map their parallel computation onto the actual number of available processors. The two-level model allows the kernel to accommodate the concurrency demands of all program types without blocking or otherwise restricting thread access to system services.

The Java on Solaris design uses system resources efficiently as they are needed. Applications can have thousands of threads with minimal thread-use overhead. Threads execute independently, share process instructions, and share data transparently with the other threads in a process. Threads also share most of the operating system state of a process, can open files and permit other threads to read them, and allow different processes to synchronize with each other in varying degrees.

The Java on Solaris threaded model delivers the best combination of speed, concurrency, functionality, and kernel resource utilization.

Grouping Threads

Every Java thread is a member of a thread group. Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.

For example, you can start or suspend all the threads within a group with a single method call. Java thread groups are implemented by the `ThreadGroup` [(in the API reference documentation)] class in the `java.lang` package. The runtime system puts a thread into a thread group during thread construction. When you create a thread, you can either allow the runtime system to put the new thread in a reasonable default group or you can explicitly set the new thread's group. The thread is a permanent member of whatever thread group it joins upon its creation; you cannot move a thread to a new group after the thread has been created.

Java Threads Issues

This section discusses Java-generic and Solaris-specific issues that might be of concern if you are writing Java applications for the Solaris product.

Generic Java Issues

Numerous methods have been deprecated for JDK 1.1. Refer to Table 4–1 for a complete list.

Solaris-Specific Issues

Some issues are specific to Solaris, as explained in the following sections.

Using Multithreading-Unsafe Libraries

Caution - This workaround is not trivial and can cause deadlocks if not carefully programmed. Do this only if absolutely unavoidable.

If you try to run a multithreaded Java application that also uses native C/C++ code with previously-released libraries that have not been compiled with the `-D_REENTRANT` flag on, you could encounter problems, as explained here.

With a native-threaded JVM such as 1.1, `libc` stores system call error code in a thread-specific `errno`. When an mt-unsafe library references `errno`, it references the global version, because it was not compiled with the `-D_REENTRANT` flag on. Therefore, the library can't access the thread-specific `errno` and its `errno`-dependent response to a failed system call would be incorrect.

The real solution is to ensure that an MT Java application that also uses native code by way of native methods is linked with MT-safe (or at least `errno`—safe libraries).

However, if you cannot avoid referencing `errno`-unsafe libraries, the following workaround can help: Enable the main thread to enter the Java application and arrange for all calls to the unsafe library to be routed through the main thread. For example, if a thread makes a JNI call, the JVM can marshal all JNI arguments and put them in a queue serviced by the main thread. The thread can wait for the main thread to issue the call and return the results to it.

It is not necessary for calls made from only the main thread to the unsafe library to go through a lock, since calls to the library are single-threaded through the library; only the main thread ever calls the library. The main thread could issue non-blocking calls, and so forth, to ensure some amount of concurrency. The main thread's `errno` is global, and the same `errno` would be referenced by both `libc` and the MT-unsafe library.

`interrupt()` Method

The use of this method is generally discouraged; it is not currently specified as particularly useful. The *Java Language Specification* (JLS) defines it as a way to interrupt a target thread only if and when it calls the `wait()` method.

However, on the Solaris platform, the semantics have been extended so that it also interrupts the target thread's I/O calls. Do not depend on this extension, as it might be discontinued. Additionally, using the extended, I/O interruption semantics of the interrupt method makes the code non-portable across different JVMs.

Thread Priorities

The thread priorities available to Java threads on a native threaded JVM should be treated as hints to the scheduler, especially if the threads are compute-bound. The number of processors available to a process is dynamic and unpredictable. Therefore, an attempt to use priorities to schedule execution on any multi-tasked, multiprocessor system is not likely to succeed.

CHAPTER 3

Java Programming Environment

Programming in Java is supported in the Solaris JVM by any Solaris text editor, `make`(1S), and by the components shown in the following table.

TABLE 3–1 Java Programming Environment Components

Component	Description
`javac`	Java compiler. Translates Java source code files (`name.java`) into bytecode files (`name.class`) that can be processed by the interpreter (`java`(1)). Both Java applications and Java applets are compiled.
`javald`	Wrapper generator. Creates a wrapper that captures the environment needed to compile and run a Java application. Because the specified paths are not bound until the wrapper is invoked, the wrapper allows for relocation of the `JAVA_HOME` and `CLASSPATH` paths.
`java`	Java interpreter. Can be invoked as a command to execute a Java application or from a browser by HTML code to execute an applet.
`appletviewer`	Java applet viewer. This command displays specified document(s) or resource(s) and runs each applet referred to by the document(s).
`javap`	Java class file disassembler. Disassembles a `javac` compiled bytecode class file and prints the result to `stdout`.

(For more information on using `make`(1S) see the chapter “make Utility” in the *Programming Utilities Guide*.)

The normal Java environment variables are shown in the following table.

TABLE 3–2 Java Environment Variables

Variable	Description
JAVA_HOME	Path of the base directory of the Java software. For example, `javac`, `java`, `appletviewer`, `javap`, and `javah` are all contained in `$JAVA_HOME/bin`. Does not need to be set to use Solaris JVM.
CLASSPATH	A colon (:) separated list of paths to directories containing compiled `*.class` files for use with applications and applets. Used by `javac`, `java`, `javap`, and `javah`. If not set, all Solaris JVM executables default to `/usr/java/lib/classes.zip`. Does not need to be set to use Solaris JVM.
PATH	The normal executable search list can contain `$JAVA_HOME/bin`.

Note - The JVM tools are installed in `/usr/java/bin` and symbolic links to each executable are stored in `/usr/bin`. This means that nothing needs to be added to a user's `PATH` variable to use the newly installed JVM package. Also, all Solaris JVM executables default to the path `/usr/java/lib/classes.zip` to find the standard Java class library.

The base Java programming environment provides no debugger. A debugger is included in the optional unbundled Java WorkShop™ from Sun Microsystems.

Java Programs

Java programs are written in two forms: applications and applets.

Java applications are run by invoking the Java interpreter from the command line and specifying the file containing the compiled application.

Java applets are invoked from a browser. The HTML code interpreted by the browser names a file containing the compiled applet. This causes the browser to invoke the Java interpreter which loads and runs the applet.

Sample Application

Code Example 3–1 is the source of an application that displays "Hello World" on `stdout`. The method accepts arguments in the invocation, but does nothing with them.

CODE EXAMPLE 3–1 Sample Java Application Code

```
//
// HelloWorld Application
//
class HelloWorldApp{
    public static void main (String args[]) {
        System.out.println ("Hello World");
    }
}
```

Note that, as in C, the method or function to be initially executed is identified as `main`. The keyword `public` lets the method be run by anyone; `static` makes `main` refer to the class `HelloWorldApp` and no other instance of the class; `void` says that `main` returns nothing; and `args[]` declares an array of type `String`

To compile the application, enter

```
$ javac HelloWorldApp.java
```

It is run by

```
$ java HelloWorldApp arg1 arg2 ...
```

Sample Applet

Code Example 3–2 is the source of the applet that is equivalent to the application in Code Example 3–1.

CODE EXAMPLE 3–2 Sample Java Applet

```
//
// HelloWorld Applet
//
import java.awt.Graphics;
import java.applet.Applet;

public class HelloWorld extends Applet {
    public void paint (Graphics g) {
        g.drawstring ("Hello World", 25, 25);
    }
}
```

In an applet, all referenced classes must be explicitly `import`ed. The keywords `public` and `void` mean the same as in the application; `extend` says that the class `HelloWorld` inherits from the class `Applet`.

To compile the applet, enter

```
$ javac HelloWorld.java
```

The applet is invoked in a browser by HTML code. A minimum HTML page to run the applet is:

```
<title>Test</title>
<hr>
<applet code="HelloWorld.class" width=100 height=50>
</applet>
<hr>
```

javald and Relocatable Applications

Correct execution of many Java applications depends on the values of the JAVA_HOME, CLASSPATH, and LD_LIBRARY_PATH environment variables. Because the values of these environment variables are controlled by each user, they can be set to arbitrary paths, with either path being unusual. Further, it is common for an application to require a unique value in the CLASSPATH variable.

javald(1) is a command that generates wrappers for Java applications. The wrapper can specify the correct paths for any or all of the JAVA_HOME, CLASSPATH, and LD_LIBRARY_PATH environment variables. It does so with no effect on the user's values of these environment variables. It also overrides the user's values for these environment variables during execution of the Java application. Further, the wrapper ensures that the specified paths are not bound until the Java application is actually executed, which maximizes relocatability of applications.

Programming Compute-Bound, Parallellized Java Applications

Caution - In general, avoid using a native method to access Solaris-specific functionality such as thr_setconcurrency(3THR), since the application is then tied to the Solaris environment and is no longer 100% Pure Java.

Note - Most Java applications do not need to use `thr_setconcurrency`(3THR). The only cases in which it might be necessary would be, for instance, a demo with dummy threads that spin interminably or a compute-bound application such as matrix multiplication or a parallelized graphics computation.

`thr_setconcurrency` Example

A compute-bound application, such as parallelized matrix multiplication, must use a native method to call `thr_setconcurrency`(3THR). This insures that sufficient concurrency resources are available to the Java application to fully use multiple processors. This is not necessary for most Java applications and applets. The following code is an example of how to do this.

The first element is the Java application, `MPtest.java`, that will use `MPtest_NativeTSetconc()`. This application creates 10 threads, each of which displays an identifying line, then loops 10,000,000 times to simulate a compute-bound activity.

CODE EXAMPLE 3–3 `MPtest.java`

```
import java.applet.*;
import java.io.PrintStream;
import java.io.*;
import java.net.*;

class MPtest {
   static native void NativeTSetconc();
   static public int THREAD_COUNT = 10;
   public static void main (String args[]) {
      int i;

// set concurrency on Solaris - sets it to
// sysconf (_SC_ NPROCESSORS_ONLN)
      NativeTSetconc();
// start threads
      client_thread clients[] = new client_thread[ THREAD_COUNT ];
      for ( i = 0; i < THREAD_COUNT; ++i ){
         clients[i] = new client_thread(i, System.out);
         clients[i].start();
      }
   }

   static { System.loadLibrary("NativeThreads");
}
class client_thread extends Thread {
   PrintStream out;
   public int LOOP_COUNT = 10000000;
   client_thread(int num, PrintStream out){
```

(continued)

(Continuation)

```
      super( "Client Thread" + Integer.toString( num ) );
      this.out = out;
      out.println("Thread " + num);
   }
   public void run () {
      for( int i = 0; i < this.LOOP_COUNT  ; ++i ) {;
    }
  }
}
```

The second element is the C stub file, `MPtest.c`, generated from `MPtest.java` by the utility `javah`(1). Enter

```
% javah -stubs MPtest.java
```

The third element is the C header file, `MPtest.h`, also generated from `MPtest.java` by the utility `javah`(1). Enter

```
% javah MPtest.java
```

The fourth element is the C function, `NativeThreads.c`, which performs the call to the C library interface.

```
#include <thread.h>
#include <unistd.h>
#include <jni.h>
JNIEXPORT void JNICALL Java_MPtest_NativeTSetconc(JNIEnv *env, jclass obj) {
  thr_setconcurrency(sysconf(_SC_NPROCESSORS_ONLN));
}
```

Finally, combining the four files into the Java application, `MPtest.class`, is most easily done with a `make`(1S) file such as shown in Code Example 3–4.

CODE EXAMPLE 3–4 `MPtest.class`

```
# Make has to be done in two stages:
# first do "make MPtest"
# Then create NativeThreads.c to incorporate the native call
# to "thr_setconcurrency(_SC_NPROCESSORS_ONLN)"
# and then do "make lib".
# After this, you should be able to run "java MPtest" with LD_LIBRARY_PATH
# and CLASSPATH set to "."
JAVA_HOME=/usr/java JH_INC1=${JAVA_HOME}/include JH_INC2=${JAVA_HOME}/include/solaris
CLASSPATH=.;
export CLASSPATH;
```

(continued)

(Continuation)

```
MPtest:
${JAVA_HOME}/bin/javac MPtest.java
(CLASSPATH=.;
export CLASSPATH;
${JAVA_HOME}/bin/javah MPtest)
(CLASSPATH=.;
export CLASSPATH;
${JAVA_HOME}/bin/javah -jni MPtest)cc -G -I${JH_INC1} -I${JH_INC2} NativeThreads.c\
   -lthread -o libNativeThreads.so
clean:
rm -rf *.class libNativeThreads.so NativeThreads.o *.h
```

API Mapping

The mapping table shows the closest possible mapping of the Java threads API to the Solaris and POSIX APIs. This mapping is not exact and does not imply that you can convert a Solaris or POSIX threads program to a Java threads program (or vice versa) using the table. The table serves only to show a loose equivalence between the APIs and to some guidance to developers familiar with one API and interested in knowing its relationship to the corresponding API. A conceptual difference exists between using the Solaris APIs by way of procedural and layered programming in C, and using them by object-oriented programming techniques in Java.

The following examples show why the Java/Solaris API equivalence is loose.

- The Java thread destroy method (`Destroy()`) is shown to correspond to POSIX `pthread_cancel()`. However, POSIX `pthread_cancel()` is incomplete without the concept of cancellation points and the use of `pthread_cleanup_push()` and `pthread_cleanup_pop()` to establish cleanup handlers around cancellation points. The Java threads API does not have a similar conceptual framework about destroying threads. In this sense, the two destroy techniques are very different.

Note - As of JDK 1.1, the `destroy()` method has been deprecated.

- The Java thread `interrupt()` method is shown as corresponding to POSIX `pthread_kill()`, but is quite different. Java has the concept of safe interruption points (for instance, `wait()`), whereas POSIX does not.

Note - The Solaris readers/writer lock interfaces and the POSIX attributes do not have any close equivalent interfaces in Java.

TABLE 3–3 Mapping of Java to Solaris and POSIX APIs

Java Threads API	Solaris Threads API	POSIX Threads API
	thr_create( )	pthread_create( )
activeCount( )		
checkAccess( )		
countStackFrames( )		
currentThread( )	thr_self( )	pthread_self( )
destroy( )		pthread_cancel( )
dumpStack( )		
enumerate( )		
getName( )		
getPriority( )	thr_getprio( )	pthread_ getschedparam( )
getThreadGroup( )		
interrupt( )	thr_kill( )	pthread_kill( )
interrupted( )		
isAlive( )		
isDaemon( )		
isInterrupted( )		
join( )	thr_join( )	pthread_join( )
resume( )	thr_continue( )	
run( )		
setDaemon( )	THR_DAEMON flag	
setName( )		
setPriority( )	thr_setprio( )	pthread_ setschedparam( )
sleep( )	sleep( )	sleep( )
start( )		
stop( )		
suspend( )		

TABLE 3–3 Mapping of Java to Solaris and POSIX APIs *(continued)*

Java Threads API	Solaris Threads API	POSIX Threads API
Synchronization methods		
`wait()`	`cond_wait()`	`pthread_cond_wait()`
`notify()`	`cond_signal()`	`pthread_cond_signal()`
synchronized method synchronized statements	mutexes	`pthread_mutexes`

Thread Group Methods

The following methods operate on a thread group. The thread group feature is available in Java, but there is no corresponding feature in Solaris or POSIX:

- `activeCount()`
- `activeGroupCount()`
- `allowThreadSuspension()`
- `checkAccess()`
- `getMaxPriority()`
- `getParent()`
- `getName()`
- `isDaemon()`
- `list()`
- `parentOf()`
- `resume()`
- `setDaemon()`
- `stop()`
- `suspend()`
- `toString()`
- `uncaughtException()`

Java Development Tools

The following sections describe Java development tools.

Java WorkShop (JWS)

JWS is a powerful, visual development tool for professional Java developers. It offers a complete, easy-to-use toolset for building Java applets and applications quickly and easily.

JWS uses its own Java interpreter and consists of eight applications, as shown in the following table.

TABLE 3–4 Java WorkShop Application List

Application	Description
Portfolio Manager	Creates and customizes portfolios of Java projects. It manages collections of objects and applets from which new applets and applications can be created.
Project Manager	Sets preferences and directories for a project. Organizes and saves locations and preferences so that developers need not memorize paths to components.
Source Editor	A point-and-click tool for creating and editing source code. Other components of Java WorkShop invoke the Source Editor at many points in the creation, compiling, and debugging processes.
Build Manager	Compiles Java source code to Java bytecode and locates errors in the source. In launching the Source Editor, the Build Manager links the developer to the Source Editor, allowing quick correction and compilation.
Source Browser	Displays a tree diagram that shows the class inheritance of all the objects in the project. It also lists all constructor and general methods in the project and allows string and symbol searches. The Source Browser links to the Source Editor to view the code.
Debugger	Provides an array of tools to control and manage the debugging process. By running the application or applet under a control panel, the developer can stop and resume threads, set break points, trap exceptions, view threads in alphabetical order, and see messages.

TABLE 3–4 Java WorkShop Application List *(continued)*

Application	Description
Applet Tester	Similarly to appletviewer, Applet Tester lets the developer run and test the applet. Use Build Manager to compile the applet, then run it with Applet Tester.
Online Help	Is organized into the topics “Getting Started,“ “Debugging Applets,” “Building Applets,” “Managing Applets,” and “Browsing Source”. There are also buttons for a table of contents and index.
Visual Java	An integrated Java GUI builder that has a point-and-click interface with a pallet of customizable pre-built GUI foundation widgets.

For more JWS information, refer to `http://www.sun.com/workshop/java/`.

CHAPTER 4

Deprecated Methods

This chapter has a list of methods deprecated in JDK 1.1, and an explanation of deprecation.

What Is Deprecation?*

A method is deprecated when it is no longer considered important, and should no longer be used because it might be deleted from its class. Deprecation is a result of classes evolving, causing their APIs to change. Methods are renamed, new ones added, attributes change. Deprecated classes and methods are marked "@deprecated" in documentation comments to enable developer transition from that API to the new one. The following table lists the deprecated methods.

TABLE 4–1 Deprecated Methods

Class	Method	Replaced By
java.awt.BorderLayout	addLayoutComponent()	addLayoutComponent(component,object)
java.awt.CardLayout	addLayoutComponent()	addLayoutComponent(component,object)
java.awt.CheckboxGroup	getCurrent()	getSelectedCheckbox()
	setCurrent()	setSelectedCheckbox()
java.awt.Choice	countItems()	getItemCount()
java.awt.Component	getPeer()	No replacement.
	enable()	setEnabled(true)
	disable()	setEnabled(false)

TABLE 4–1 Deprecated Methods *(continued)*

Class	Method	Replaced By
	show()	setVisible(true)
	hide()	setVisible(false)
	location()	getLocation()
	move()	setLocation()
	size()	getSize()
	resize()	setSize()
	bounds()	getBounds()
	reshape()	setBounds()
	preferredSize()	getPreferredSize()
	minimumSize()	getMinimumSize()
	layout()	doLayout()
	inside()	contains()
	locate()	getComponentAt()
	deliverEvent()	dispatchEvent()
	postEvent()	dispatchEvent()
	handleEvent()	processEvent()
	mouseDown()	processMouseEvent()
	mouseDrag()	processMouseMotionEvent()
	mouseUp()	processMouseEvent(MouseEvent)
	mouseMove()	processMouseMotionEvent()
	mouseEnter()	processMouseEvent()
	mouseExit()	processMouseEvent()
	keyDown()	processKeyEvent()
	keyUp()	processKeyEvent()
	action()	Register as `ActionListener` on component firing action events.
	gotFocus()	processFocusEvent()
	lostFocus()	processFocusEvent()
	extFocus()	transferFocus()
java.awt.Container	countComponents()	getComponentCount()

TABLE 4–1 Deprecated Methods *(continued)*

Class	Method	Replaced By
	insets()	getInsets()
	preferredSize()	getPreferredSize()
	minimumSize()	getMinimumSize()
	deliverEvent()	dispatchEvent()
	locate()	getComponentAt()
java.awt.FontMetrics	getMaxDescent()	getMaxDescent()
java.awt.Frame	setCursor()	setCursor()method in `Component`
	getCursorType()	getCursor() method in `Component`
java.awt.Graphics	getClipRect()	getClipBounds()
java.awt.List	countItems()	getItemCount()
	clear()	removeAll()
	isSelected()	isIndexSelected()
	allowsMultipleSelections()	isMultipleMode()
	setMultipleSelections()	setMultipleMode()
	preferredSize()	getPreferredSize()
	minimumSize()	getMinimumSize()
	delItems()	No longer for `public` use; retained as `package private`.
java.awt.Menu	countItems()	getItemCount()
java.awt.MenuBar	countMenus()	getMenuCount()
java.awt.MenuComponents	getPeer()	No replacement.
	postEvent()	dispatchEvent()
java.awt.MenuContainer	postEvent()	dispatchEvent()
java.awt.MenuItem	enable()	setEnabled(true)
	disable()	setEnabled(false)
java.awt.Polygon	getBoundingBox()	getBounds()
	inside()	contains()
java.awt.Rectangle	reshape()	setBounds()
	move()	setLocation()
	resize()	setSize()

TABLE 4–1 Deprecated Methods *(continued)*

Class	Method	Replaced By
	inside()	contains()
java.awt.ScrollPane	layout()	doLayout()
java.awt.Scrollbar	getVisible()	getVisibleAmount()
	setLineIncrement()	setUnitIncrement()
	getLineIncrement()	getUnitIncrement()
	setPageIncrement()	setBlockIncrement()
	getPageIncrement()	getBlockIncrement()
java.awt.TextArea	insertText()	insert()
	appendText()	append()
	replaceText()	replaceRange()
	preferredSize()	getPreferredSize()
	minimumSize()	getMinimumSize()
java.awt.TextField	setEchoCharacter()	setEchoChar()
	preferredSize()	getPreferredSize()
	minimumSize()	getMinimumSize()
java.awt.Window	postEvent()	dispatchEvent()
java.io. ByteArrayOutputStream	toString()	toString(String enc) or toString(), which uses the platform's default character encoding
java.io.DataInputStream	readLine()	BufferedReader.readLine()
java.io.PrintStream	printStream()	PrintWriter class
java.io.StreamTokenizer	streamTokenizer()	Convert input stream to character stream
java.lang.Character	isJavaLetter()	isJavaIdentifierStart(char)
	isJavaLetterOrDigit()	isJavaIdentifierPart(char)
	isSpace()	isWhitespace(char)
java.lang.ClassLoader	defineClass()	defineClass (java.lang.String,byte[],int,int)
java.lang.Runtime	getLocalizedInputStream()	InputStreamReader and BufferedReader classes
	getLocalizedOutputStream()	Use OutputStreamWriter, BufferedWriter, and PrintWriter classes
java.lang.String	string()	Use String constructors that take a character-encoding name or use default encoding.

TABLE 4–1 Deprecated Methods *(continued)*

Class	Method	Replaced By
	getBytes()	getBytes(String enc) or getBytes()
java.lang.System	getenv()	Use java.lang.System.getProperty methods' system properties and corresponding get TypeName methods of Boolean, Integer, and Long primitive types.
java.lang.Thread	resume()	Refer to "Deprecated Threads Methods" on page 46
java.lang.Thread	stop()	Refer to "Deprecated Threads Methods" on page 46
java.lang.Thread	suspend()	Refer to "Deprecated Threads Methods" on page 46
java.util.Date	getYear()	Calendar.get(Calendar.YEAR)-1900
	setYear()	Calendar.set(Calendar.YEAR+1900)
	getMonth()	Calendar.get(Calendar.MONTH)
	setMonth()	Calendar.set(Calendar.MONTH,int month)
	getDate()	Calendar.get(Calendar.DAY_OF_MONTH)
	setDate()	Calendar.set(Calendar.DAY_OF_MONTH,int date)
	getDay()	Calendar.get(Calendar.DAY_OF_WEEK)
	getHours()	Calendar.get(Calendar.HOUR_OF_DAY)
	setHours()	Calendar.set(Calendar.HOUR_OF_DAY,int hours)
	getMinutes()	Calendar.get(Calendar.MINUTE)
	setMinutes()	Calendar.set(Calendar.MINUTE,int minutes)
	getSeconds()	Calendar.get(Calendar.SECOND)
	setSeconds()	Calendar.set(Calendar.SECOND,int seconds)
	parse()	DateFormat.parse(String s)
	getTimezoneOffset()	Calendar.get(Calendar.ZONE_OFFSET) +Calendar.get(Calendar.DST_OFFSET)
	toLocaleString()	DateFormat.format(Date date)

TABLE 4–1 Deprecated Methods *(continued)*

Class	Method	Replaced By
	toGMTString()	DateFormat.format(Date date) using a GMT TimeZone
	UTC()	Calendar.set (year+1900,month,date,hrs,min,sec) or GregorianCalendar (year+1900,month,date,hrs,min,sec), using a UTC TimeZone, followed by Calendar.getTime().getTime().

Deprecated Threads Methods

The `Thread.stop`, `Thread.suspend`, and `Thread.resume` methods are deprecated as of JDK 1.1. `Thread.stop` is being deprecated because it is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the `ThreadDeath` exception propagates up the stack.) If any of the objects previously protected by these monitors was in an inconsistent state, other threads might view these objects in an inconsistent state. Such objects are said to be damaged.

Threads operating on damaged objects can behave arbitrarily, either obviously or not. Unlike other unchecked exceptions, `ThreadDeath` kills threads silently; thus, the user has no warning that the program might be corrupted. The corruption can manifest itself at an unpredictable time after the damage occurs.

Substitute any use of `Thread.stop` with code that provides for a gentler termination. Most uses of `stop()` can and should be replaced by code that modifies a variable indicating that the target thread should stop running. The target thread should check this variable regularly. If the variable indicates that the thread is to stop, the thread should then return from its `run()` method in an orderly fashion. For example, suppose your applet contains the following `start()`, `stop()`, and `run()` methods:

```
public void start() {
  blinker = new Thread(this);
  blinker.start();
}
public void stop() {
```

(continued)

```
  blinker.stop();
// UNSAFE!
 }
public void run() {
  Thread thisThread = Thread.currentThread();
  while (true) {
   try {
    Thread.sleep(interval);
   }
   catch (InterruptedException e){
   }
   repaint();
  }
 }
```

You can avoid the use of `Thread.stop` by replacing the applet's `stop()` and `run()` methods with:

```
public void stop() {
  blinker = null;
}
public void run() {
  Thread thisThread = Thread.currentThread();
  while (blinker == thisThread) {
   try {
    Thread.sleep(interval);
   }
   catch (InterruptedException e){
   }
   repaint();
  }
}
```

`Thread.suspend` is inherently deadlock-prone, so it is also being deprecated. Thus, the deprecation of `Thread.resume` is also necessary. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling `resume()`, deadlock results.

Such deadlocks typically manifest themselves as frozen processes. As with `Thread.stop`, the prudent approach is to have the target thread poll a variable indicating the desired state of the thread (active or suspended). When the correct state is suspended, the thread waits using `Object.wait`. When the thread is resumed, the target thread is notified using `Object.notify`. For example, suppose your applet contains the following `mousePressed` event handler, which toggles the state of a thread called `blinker`:

```
public void mousePressed(MouseEvent e) {
  e.consume();
  if (threadSuspended)
   blinker.resume();
  else
   blinker.suspend();
// DEADLOCK-PRONE!
  threadSuspended = !threadSuspended;
}
```

You can avoid the use of `Thread.suspend` and `Thread.resume` by replacing the event handler above with:

```
public synchronized void mousePressed(MouseEvent e) {
  e.consume();
  threadSuspended = !threadSuspended;
  if (!threadSuspended)
   notify();
}
```

and adding the following code to the run loop:

```
synchronized(this) {
  while (threadSuspended)
   wait();
}
```

The `wait()` method throws the `InterruptedException`, so it must be inside a `try ... catch` clause. You can also put it in the same clause as the `sleep`. The `check` should follow (rather than precede) the `sleep` so the window is immediately repainted when the thread is resumed. The resulting `run()` method follows:

```
public void run() {
  while (true) {
   try {
    Thread.sleep(interval);
    synchronized(this) {
     while (threadSuspended)
      wait();
    }
   }
   catch (InterruptedException e){
   }
   repaint();
  }
}
```

The `notify()` in the `mousePressed()` method and the `wait()` in the `run()` method are inside `synchronized` blocks. This is required by the language, and

ensures that `wait( )` and `notify( )` are properly serialized. In practical terms, this eliminates race conditions that could cause the suspended thread to miss a `notify( )` and remain suspended.

CHAPTER 5

Application Performance Tuning

This chapter provides information about how to improve performance for your Java applications in the Solaris 8 environment. An application's performance can be defined as its usage of resources; therefore, performance tuning is the minimizing of its usage of those resources.

Caution - Many of these performance tuning tips are specific to Java on the Solaris 2.6, Solaris 7, and Solaris 8 platforms. Future releases might have different performance characteristics; therefore, these tips might not continue to be appropriate.

Tuning Techniques

Tuning can exist on several levels, as described in the following sections.

System Interface Level

These areas of the Java system interface level, where tuning can often result in significant performance gains, are discussed here:

- I/O
- Strings
- Arrays
- Vectors
- Painting/drawing
- Hashing

- Images
- Memory usage
- Threads

Compiler Optimization Level

The optimizations for these compilers are listed as:

- Java compiler
- JIT compiler

Code Tuning Level

Code tuning in these areas can be used to increase performance:

- Loops
- Convert `expr` to table lookup
- Caching
- Result pre-computation
- Lazy evaluation
- Class vs. object initialization

I/O Issues

The biggest and most common performance problem in Java applications is often inefficient I/O. Therefore, I/O issues should generally be the first thing to look at when performance-tuning a Java application. Fixing these problems often results in greater performance gains than all the other possible optimizations combined. It is not unusual to see a speed improvement of one order of magnitude achieved by using efficient I/O techniques.

If an application performs a significant amount of I/O, then it is a candidate for I/O performance tuning. This conclusion can be confirmed by profiling the application. To learn how to profile an application, you can use the Java WorkShop (JWS) product. JWS can be obtained from:

`http://www.sun.com/workshop`

Select Help->Help Contents, and click on Profiling Projects. This example involves running a benchmark test reading a 150,000-line file using four different methods:

1. `DataInputStream.readLine( )` alone (unbuffered)
2. `DataInputStream.readLine( )` with a `BufferedInputStream` underneath, which has a buffer size of 2048 bytes

3. `BufferedReader.readline()` with a buffer size of 8192 bytes.
4. `BufferedFileReader(fileName)`

The results were as follows: (times in seconds) :

`DataInputStream`:	178.740
`DataInputStream(BufferedInputStream)`:	21.559
`BufferedReader`	11.150
`BufferedFileReader`	6.991

Note that methods 1 and 2 do not properly handle Unicode characters, while methods 3 and 4 handle them correctly. This makes methods 1 and 2 unacceptable for most product uses. Also, `DataInputStream.readLine()` is deprecated as of JDK 1.1. Method 1 is used in JWS and other programs.

Another way to spot Solaris I/O problems is to use `truss`(1) to look for `read`(1) and `write`(1) system calls.

Strings

When using strings, the most important thing to remember is to use `char` arrays for all character processing in loops, instead of using the `String` or `StringBuffer` classes. Accessing an array element is much faster than using the `charAt()` method to access a character in a string. Also, remember that string constants ("...") are already string objects.

```
//DON'T
String s = new String("hello");
//DO
String s = "hello";
```

In addition:

- `class String`

  Do not use this class for mutable strings, character processing, or `charAt()` method inside a loop.

- `class StringBuffer`

  Use this class only when a string is mutable, accessed concurrently by multiple threads, and no character processing is performed. Do not use for immutable strings, character processing, or `charAt()`, `setCharAt()` methods inside a loop. The default string size is 16 characters. This class is automatically used by the compiler for string concatenation. Set the initial buffer size to the maximum string length, if it is known.

- `class StringTokenizer`

This class is useful for simple parsing or scanning, but is very inefficient. It can be optimized by storing the string and delimiter in a character array instead of in `String`, or by storing the highest delimiter character to allow a quicker check. This will result in a 1.6x to 10x performance increase (2.4x is typical), depending on the delimiter list and target string.

Arrays

Arrays are bounds-checked, which will degrade performance. However, accessing arrays is much faster than accessing `Vector`, `String`, and `StringBuffer`. Use `System.arraycopy( )` to improve performance. This is a native method, and much faster than manual array processing.

Vectors

`Vector` is convenient to use, but inefficient. For best performance, use it only when the structure size is unknown, and efficiency is not a concern. When using `Vector`, ensure that `elementAt( )` is not used inside a loop, as performance will degrade. Use `Vector` only when you have an array with the following characteristics:

- Accessed concurrently by multiple threads
- Dynamic size

Hashing

`HashTable` has these tunable parameters:

- Capacity (usually a prime number), `initialCapacity`; if this is not set large enough, collisions will result, causing hashing to stop and linear list processing to be executed afterwards.
- Load factor (0.0-1.0), `loadFactor`, which is a percentage of capacity beyond which the table will expand. `HashTable` calls `hashCode( )`. These classes have pre-defined `hashCode( )` methods:
- `Color`, `Font`, `Point`
- `File`
- `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`, `String`
- URL
- `BitSet`, `Date`, `GregorianCalendar`, `Locale`, `SimpleTimeZone`. Note that `String.hashCode( )` does not always sample all the characters, depending on the length:
- Length from 1 to 15: all `n` Length from 16 to 23: every other character
- Length from 24 to 31: every third character

Images

You can use the following types of images.

Painting and Drawing

To improve performance in these areas, use the following techniques:

- Double buffering (for instance, for animation, draw the image off-screen and load all at once).
- Overriding the default, `update( )` function

```
public void update(Graphics g) {
    paint(g);
   }
```

- Custom layout managers. If you want custom behavior, GUI performance is best if you write your own.
- Events. The JDK 1.1 has a more efficient event model than JDK 1.0.
- Repaint only the damaged regions (use `ClipRect`).

Asynchronous Loading

To improve (asynchronous) loading performance, use your own `imageUpdate( )` method to override `imageUpdate( )`. `imageUpdate( )` can cause more repainting than you might want..

```
//wait for the width information to be loaded
while (image.getWidth(null) == -1 {
  try {
   Thread.sleep(200);
  }
  catch(InterruptedException e) {
  }
 }
 if (!haveWidth) {
  synchronized (im) {
   if (im.getWidth(this) == -1) {
    try {
     im.wait();
          }
         catch (InterruptedException) {
    }
   }
  }
```

(continued)

(Continuation)

```
//If we got this far, the width is loaded, we will never go thru
// all that checking again.
  haveWidth = true;
 }
...
public boolean imageUpdate(Image img, int flags, int x, int y, int width, \
   int height) {
  boolean moreUpdatesNeeded = true;
  if ((flags&ImageObserver.WIDTH)!= 0 {
   synchronized (img) {
    img.notifyAll();
    moreUpdatesNeeded = false;
   }
  }
  return
  moreUpdatesNeeded;
}
```

Pre-Decoding

Pre-decoding and storing the image in an array will improve performance. Image decoding time is greater than loading time. Pre-decoding using `PixelGrabber` and `MemoryImageSource` should combine multiple images into one file for maximum speed. These techniques are more efficient than polling.

Memory Usage

You can dramatically improve application performance by reducing the amount of garbage collection performed during execution. The following practices can also increase performance:

- Increase the initial heap size from the 1 MByte default with:

  `java -ms number . java -mx number .`

  The default maximum heap size is 16 MBytes.

- Find areas where too much memory is being used with:

  `java -verbosegc`

- Take size into account when allocating arrays (for instance, if `short` is big enough, use it instead of `int`).
- Avoid allocating objects in loops (`readLine( )` is a common example)

Threads

As discussed in “Java Threads in the Solaris Environment — Earlier Releases*” on page 17, performance is increased dramatically by using native threads. Green threads are not time-sliced and might require calls to `Thread.yield()` in loops, slowing execution. Other techniques to avoid:

- Overuse of synchronization increases the possibility of deadlock (because of coding errors) and increases the likelihood of delays due to lock contention. Also, the overhead of synchronizing might frequently overcome the advantages. Minimizing synchronization takes work, but it pays off well.
- Polling: it is acceptable only when waiting for outside events and should be performed in a "side" thread. Use `wait()`/`notify()` instead.

Compiler Optimizations

The following compilers automatically perform the listed optimizations.

Java Compiler

- Inlining
- Constant folding

JIT Compiler

- Elimination of some array bounds checking
- Elimination of common sub-expressions within blocks
- Empty method elimination
- Some register allocation for locals
- No flow analysis
- Limited inlining

Code Optimization

Loops

Use these techniques for performance improvements:

- Move loop invariants outside the loop.
- Make the tests as simple as possible.

- Use only local variables inside a loop; assign class fields to local variables before the loop.
- Move constant conditionals outside loops.
- Combine similar loops.
- If loops are interchangeable, nest the busiest one.
- As a last resort, unroll the loop.

Convert `expr` *to Table Lookup*

When a value is being selected based on a single expression with a range of small integers, convert it to a table lookup. Conditional branches defeat many compiler optimizations.

Caching

Though caching takes more memory, it can be used for performance improvement. Use the technique of caching values that are expensive to fetch or compute.

Precompute Results

Increase performance by precomputing values known at compile time.

Lazy Evaluation

Save startup time by delaying computation of results until they are needed.

Class as Opposed to Object Initialization

Speed performance up by putting all one-time initializations into a class initializer.