Preface

This document helps in troubleshooting problems that might occur with applications that are developed using the release of Java Platform, Standard Edition Development Kit 7 (JDK 7 release). In particular, this guide addresses possible problems between the application and the Java HotSpot virtual machine.

See Appendix D, Summary of Tools in This Release for a list of changes to the troubleshooting tools for this release of the JDK compared with the previous release.

This document is currently focused on providing information about the tools and options available for diagnostics and monitoring. It does not yet include information on garbage collection or diagnosing performance issues.

For help in troubleshooting Java applications that use desktop technologies, see the following guide: *Troubleshooting Guide for Java SE 7 Desktop Technologies*.

Java HotSpot Virtual Machine

The Java HotSpot virtual machine provides a runtime environment for instructions that were generated by a Java compiler. The JDK provides two implementations of the Java virtual machine: client VM and server VM. These two systems are essentially two different just-in-time compilers interfacing with the same runtime system.

- Client VM. This implementation is installed on platforms that are typically used for client, or desktop, applications. The client VM is tuned to reduce startup time and memory footprint. It is invoked by using the `-client` command-line option when launching an application.
- Server VM. This implementation is installed on all platforms. The server VM is designed for maximum program execution speed. It is invoked by using the `-server`command-line option when launching an application.

The default implementation depends on the platform and the class of the machine.

More information on the Java Hotspot VM can be found at the following locations:

- Java SE HotSpot at a Glance
- Java Virtual Machine Technology, on the Java SE 7 web site

Who Should Use This Guide

The target audience for this document comprises developers who are working with JDK 7, as well as support or administration personnel who maintain applications that are deployed with JDK 7.

This document is intended for readers with a high-level understanding of the components of the Java Virtual Machine, as well as some understanding of concepts such as garbage collection, threads, native libraries, and so on. In addition, it is assumed that the reader is reasonably proficient on the operating system where the Java application is installed.

How This Guide Is Organized

The first chapter of this document introduces the various diagnostic and monitoring tools, utilities, options, and system properties that are available for troubleshooting in JDK 7. The chapter also provides a summary of tools and options by category. Note that tool availability depends on the platform: Solaris Operating System (Solaris OS), Linux, or Windows. Read this chapter to get acquainted with the capabilities of the utilities and options that are available.

The second chapter describes the troubleshooting tools in detail. The chapter also provides a list of the operating system tools and utilities that may be used in conjunction with the JDK utilities and options. Finally, the chapter describes in detail how you can develop new tools using the APIs provided in the JDK.

The third through fifth chapters suggest procedures to try when you encounter a problem with memory leaks, crashes, or hangs.

The sixth chapter deals with applications that use signal handlers.

The last chapter provides suggestions on what to try before submitting a bug report, guidance on how to submit a report, and suggestions on what data to collect for the report.

Finally, there is an appendix for each of the following reference areas: environment variables, command line options, details about the format of the fatal error report, and a list of tools in this release.

Feedback and Suggestions

Troubleshooting is a very important topic. If you have feedback on this document or if you have suggestions for topics that could be covered in a future version, please send your comments.

Other Resources

Troubleshooting information for J2SE 1.5 is described in the Java 2 Platform, Standard Edition 5.0, Troubleshooting and Diagnostic Guide.

In addition, the following online troubleshooting resources are available:

- The article Monitoring and Managing Java SE 6 Platform Applications describes the most common problems in Java applications and suggests how to use several tools for troubleshooting these problems.
- The article Troubleshooting Java SE 6 Deployment explores several ways to troubleshoot running Java applications, with examples.
- The Java Tuning White Paper provides performance tuning information, techniques, and pointers for the Java programming language.
- The Documentation for the Java HotSpot VM provides information about performance tuning of the Java HotSpot virtual machine, including garbage collection tuning.
- The Java Plug-in Guide provides troubleshooting FAQ as well as information on how to enable tracing when trying to diagnose issues with the Java Plug-in.
- The Java Web Start FAQ has a troubleshooting section dealing with issues that might arise with Java Web Start.
- The bug database can be a useful resource to search for problems and solutions.
- The Solaris Operating System Freeware site is the source for freeware that is packaged for the Solaris Operating System.
- The Sunfreeware.com site (Freeware for Solaris) explains how to download freeware that is packaged for the Solaris Operating System.
- The OpenSolaris site is the principle open source community site for OpenSolaris technology.
- The blastwave.org site supports a collective effort to assemble free software that can be automatically installed on a Solaris computer.

Community Support

Community support can often be obtained using the Java Technology Forums. The forums provide a way to share information and locate solutions to problems. The forums are located at http://forums.oracle.com/forums/category.jspa?categoryID=285.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A **cache** is a copy that is stored locally. Do **not** save the file. **Note:** Some emphasized items appear

		bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Solaris release.

Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	`$`
Bash shell, Korn shell, and Bourne shell for superuser	`#`
C shell	`machine_name%`
C shell for superuser	`machine_name#`

Chapter 1

Diagnostic Tools and Options

This chapter introduces the various diagnostic and monitoring tools which can be used with Java Platform, Standard Edition Development Kit 7 (JDK 7). The tools are described in detail in Chapter 2, Detailed Tool Descriptions.

See Appendix D, Summary of Tools in This Release for a list of tools available in this release of the JDK, as well as the changes since the previous release.

NOTE - SOME OF THE COMMAND-LINE UTILITIES DESCRIBED IN THIS CHAPTER ARE EXPERIMENTAL.
THE `JSTACK`, `JINFO`, AND `JMAP` UTILITIES ARE EXAMPLES OF UTILITIES THAT ARE EXPERIMENTAL. THESE UTILITIES ARE SUBJECT TO CHANGE IN FUTURE JDK RELEASES, AND MIGHT NOT BE INCLUDED IN FUTURE RELEASES.

1.1 Introduction

Most of the command-line utilities described in this chapter are either included in the JDK release or are operating system tools and utilities. Although the JDK command-line utilities are included in the JDK download, it is important to note that they can be used to diagnose issues and monitor applications that are deployed with the Java runtime environment (JRE).

In general, the diagnostic tools and options described in this chapter use various mechanisms to obtain the information they report. In many cases the mechanisms are specific to the virtual machine implementation, operating system, and version of each. Consequently, there is some overlap of the information reported by some of the tools. This should be viewed in the context of the various problems and issues for which these tools are intended. In many cases only a subset of the tools will be applicable to a given issue at a particular point in time.

1.1.1 Command-Line Options With `-XX`

Command-line options that are prefixed with `-XX` are specific to the Java HotSpot Virtual Machine. Many of these options are important for performance tuning and diagnostic purposes, and are therefore described in this guide. See B.1 HotSpot VM Command-Line Options.

However, it is important to note that these `-XX` options are not part of the Java API and can vary from one release to the next.

1.1.2 Limitations

In some cases, the tools described here are available only for some operating systems. In addition, Solaris 10 OS introduced many advanced diagnostic features and tools that can be used in production environments, and many of the native tools are capable of providing information that is specific to the Java runtime environment.

The format of log files and of other output from command-line utilities or options is version-specific. For example, if you develop a script that relies on the format of the fatal error log, then this script might cease to work as expected if the format of the log file changes in the future.

1.1.3 Developing New Tools

In addition to the tools described in this document, you can develop new tools using the APIs that are provided with the JDK release. See 2.17 Developing Diagnostic Tools.

1.2 Summary of Tools, Options, and Commands

The tools and options are divided into the following categories, where certain tools might fall into more than one category. The tools and options are described in detail in further sections.

- Post-mortem diagnostics. These tools and options can be used to diagnose a problem after an application has crashed. See 1.2.1 Tools and Options for Post-mortem Diagnostics.
- Hung processes. These tools can be used to investigate a hung or deadlocked process. See 1.2.2 Tools and Options for Hung Processes.
- Monitoring. These tools can be used to monitor a running application. See 1.2.3 Tools and Options for Monitoring.

- Other. These tools and options can be used to help diagnose other issues. See 1.2.4 Other Tools, Options, Variables, and Properties.
- Operating system tools. These tools are provided by the specific operating systems. See 2.16 Operating-System-Specific Tools.

1.2.1 Tools and Options for Post-mortem Diagnostics

This section summarizes the options and tools that are designed for post-mortem diagnostics. If an application crashes, these options and tools can be used to obtain additional information, either at the time of the crash or later using information from the crash dump.

Tool or Option	Description and Usage
Fatal Error Log	When a fatal error occurs, an error log is created. This file contains much information obtained at the time of the fatal error. In many cases it is the first item to examine when a crash occurs. See Appendix C, Fatal Error Log.
`-XX:+HeapDumpOnOutOfMemoryError` option	This command-line option specifies the generation of a heap dump when the VM detects a native out-of-memory error. See B.1.2 `-XX:+HeapDumpOnOutOfMemoryError` Option.
`-XX:OnError` option	This command-line option specifies a sequence of user-supplied scripts or commands to be executed when a fatal error occurs. For example, on Windows, this option can execute a command to force a crash dump. This option is very useful on systems where a post-mortem debugger is not configured. See B.1.3 `-XX:OnError=` Option.
`-XX:+ShowMessageBoxOnError`	This command-line option suspends a process when a fatal error occurs. Depending on the user response, the option can launch the native debugger (for example, `dbx`, `gdb`, `msdev`) to attach to the VM. See B.1.4 `-XX:+ShowMessageBoxOnError` Option.
Other `-XX` options	Several other `-XX` command-line options can be useful in troubleshooting. See B.1.5 Other `-XX` Options.
Java VisualVM (post-mortem use on Solaris OS and Linux only)	This utility can analyze a core dump by providing a readable display of the core dump in the form of a heap dump and a thread dump, as well as overview information (for example, JVM arguments, system properties, and so forth).
`jdb` utility	Debugger support includes an `AttachingConnector`, which allows `jdb` and other Java language debuggers to attach to a core file. This can be useful when trying to understand what each thread was doing at the time of a crash. See 2.4 `jdb` Utility.
`jhat` utility	This utility provides a convenient means to browse the object topology in a heap dump. See 2.5 `jhat` Utility.
`jinfo` utility (post-mortem use on Solaris OS and Linux	This utility can obtain configuration information from a core file obtained from a crash or from a core file obtained using

only)	the `gcore` utility. See 2.6 `jinfo` Utility.
`jmap` utility (post-mortem use on Solaris OS and Linux only)	This utility can obtain memory map information, including a heap histogram, from a core file obtained from a crash or from a core obtained using the `gcore` utility. See 2.7 `jmap` Utility.
`jsadebugd` daemon (Solaris OS and Linux only)	The Serviceability Agent Debug Daemon (`jsadebugd`) attaches to a Java process or to a core file and acts as a debug server. See 2.10 `jsadebugd` Daemon.
`jstack` utility	This utility can obtain Java and native stack information from a Java process. On Solaris OS and Linux the utility can get the information also from a core file or a remote debug server. See 2.11 `jstack` Utility.
Native tools	Each operating system has native tools and utilities that can be used for post-mortem diagnosis. See 2.16 Operating-System-Specific Tools.

1.2.2 Tools and Options for Hung Processes

The options and tools in this list can help in scenarios involving a hung or deadlocked process. These tools do not require any special options to start the application.

Tool or Option	**Description and Usage**
Ctrl-Break handler (Ctrl-\ or `kill -QUIT` *pid* on Solaris OS and Linux, Ctrl-Break on Windows)	This key combination performs a thread dump as well as deadlock detection. The Ctrl-Break handler can optionally print a list of concurrent locks and their owners, as well as a heap histogram. See 2.15 Ctrl-Break Handler.
`jdb` utility	Debugger support includes attaching connectors, which allow `jdb` and other Java language debuggers to attach to a process. This can help show what each thread is doing at the time of a hang or deadlock. See 2.4 `jdb` Utility.
`jhat` utility	This utility provides a convenient means to browse the object topology in a heap dump. See 2.5 `jhat` Utility.
`jinfo` utility	This utility can obtain configuration information from a Java process. See 2.6 `jinfo` Utility.
`jmap` utility	This utility can obtain memory map information, including a heap histogram, from a Java process. On Solaris OS and Linux, the `-F` option can be used if the process is hung. See 2.7 `jmap` Utility
`jsadebugd` daemon (Solaris OS and Linux only)	The Serviceability Agent Debug Daemon (`jsadebugd`) attaches to a Java process or to a core file and acts as a debug server. See 2.10 `jsadebugd` Daemon.
`jstack` utility	This utility can obtain Java and native stack information from a Java process. On Solaris OS and Linux the `-F` option can be used if the process is hung. See

	2.11 `jstack` Utility.
Native tools	Each operating system has native tools and utilities that can be useful in hang or deadlock situations. See 2.16 Operating-System-Specific Tools.

1.2.3 Tools and Options for Monitoring

These tools are designed for monitoring applications that are running at the time.

Tool or Option	Description and Usage
Java VisualVM	This utility provides a visual interface for viewing detailed information about Java applications while they are running on a Java virtual machine. This information can be used in troubleshooting local and remote applications, as well as for profiling local applications. See 2.2 Java VisualVM.
JConsole utility	This utility is a monitoring tool that is based on Java Management Extensions (JMX). The tool uses the built-in JMX instrumentation in the Java virtual machine to provide information on performance and resource consumption of running applications. See 2.3 JConsole Utility.
`jmap` utility	This utility can obtain memory map information, including a heap histogram, from a Java process, a core file, or a remote debug server. See 2.7 `jmap` Utility.
`jps` utility	This utility lists the instrumented HotSpot Virtual Machines on the target system. The utility is very useful in environments where the VM is embedded, that is, it is started using the JNI Invocation API rather than the`java` launcher. See 2.8 `jps` Utility.
`jstack` utility	This utility can obtain Java and native stack information from a Java process. On Solaris OS and Linux the utility can get the information also from a core file or a remote debug server. See 2.11 `jstack` Utility.
`jstat` utility	This utility uses the built-in instrumentation in the HotSpot VM to provide information on performance and resource consumption of running applications. The tool can be used when diagnosing performance issues, and in particular issues related to heap sizing and garbage collection. See 2.12 `jstat` Utility.
`jstatd` daemon	This tool is an RMI server application that monitors the creation and termination of instrumented Java virtual machines and provides an interface to allow remote monitoring tools to attach to VMs running on the local host. See 2.13 `jstatd` Daemon.
`visualgc` utility	This utility provides a graphical view of the garbage collection system. As with `jstat`, it uses the built-in instrumentation of the HotSpot VM. See 2.14 `visualgc` Tool.
Native tools	Each operating system has native tools and utilities that can be useful for monitoring purposes. For example, the dynamic tracing (DTrace) capability introduced in Solaris 10 OS performs advanced monitoring. See 2.16 Operating-System-Specific Tools.

1.2.4 Other Tools, Options, Variables, and Properties

In addition to the tools that are designed for specific types of problems, these tools, options, variables, and properties can help in diagnosing other issues.

Tool or Option	Description and Usage
HPROF profiler	This simple profiler can present CPU usage, heap allocation statistics, contention profiles, heap dumps, and states of all the monitors and threads in the Java virtual machine. HPROF is useful in analyzing performance, lock contention, memory leaks, and other issues. See 2.1 HPROF - Heap Profiler.
`jhat` utility	This utility is useful in diagnosing unnecessary object retention (or memory leaks). It can be used to browse an object dump, view all reachable objects in the heap, and show which references are keeping an object alive. See 2.5 `jhat` Utility.
`jinfo` utility	This utility can dynamically set, unset, and change the values of certain Java VM flags for a specified Java process. On Solaris OS and Linux, it can also print configuration information. See 2.6 `jinfo` Utility.
`jrunscript` utility	This utility is a command-line script shell, which supports both interactive and batch-mode script execution. See 2.9 `jrunscript` Utility.
Sun Studio `dbx` debugger	This is an interactive, command-line debugging tool, which allows you to have complete control of the dynamic execution of a program, including stopping the program and inspecting its state. For details, see the latest `dbx` documentation, located at the Sun Studio Program Debugging site.
Sun Studio Performance Analyzer	This tool can help you assess the performance of your code, identify potential performance problems, and locate the part of the code where the problems occur. The Performance Analyzer can be used from the command line or from a graphical user interface. For details, see the Sun Studio Performance Analyzer site.
Sun's Dataspace Profiling: DProfile	This tool provides insight into the flow of data within Sun computing systems, helping you identify bottlenecks in both software and hardware. DProfile is supported in the Sun Studio 11 compiler suite through the Performance Analyzer GUI. For information, see the Cool Tools Community site under Other Sun Tools.
`-Xcheck:jni` option	This option is useful in diagnosing problems with applications that use the Java Native Interface (JNI) or that employ third-party libraries (some JDBC drivers, for example). See B.2.1 `-Xcheck:jni` Option.
`-verbose:class` option	This option enables logging of class loading and unloading. See B.2.2 `-verbose:class` Option.
`-verbose:gc` option	This option enables logging of garbage collection information. See B.2.3 `-verbose:gc` Option.
`-verbose:jni` option	This option enables logging of JNI. See B.2.4 `-verbose:jni` Option.
`JAVA_TOOL_OPTIONS` environment variable	This environment variable allows you to specify the initialization of tools, specifically the launching of native or Java programming language agents using the `-agentlib` or `-javaagent` options. See

	A.2 `JAVA_TOOL_OPTIONS` Environment Variable.
`java.security.debug` system property	This system property controls whether the security checks in the JRE of the Java print trace messages during execution. See A.3 `java.security.debug` System Property.

Detailed Tool Descriptions

This chapter describes in detail the troubleshooting tools that are available in JDK 7. In addition, the chapter lists operating-system-specific tools that may be used in conjunction with these troubleshooting tools. Finally, the chapter explains how you can develop new tools using the APIs provided with JDK 7.

The chapter contains the following sections:

2.1 HPROF - Heap Profiler

The Heap Profiler (HPROF) tool is a simple profiler agent shipped with the JDK release. It is a dynamically linked library that interfaces with the Java VM using the Java Virtual Machine Tools Interface (JVM TI). The tool writes profiling information either to a file or to a socket in ASCII or binary format. This information can be further processed by a profiler front-end tool.

The HPROF tool is capable of presenting CPU usage, heap allocation statistics, and monitor contention profiles. In addition, it can report complete heap dumps and states of all the monitors and threads in the Java virtual machine. In terms of diagnosing problems, HPROF is useful when analyzing performance, lock contention, memory leaks, and other issues.

In addition to the HPROF library, the JDK release includes the source for HPROF as JVM TI demonstration code. This code is located in the`$JAVA_HOME/demo/jvmti/hprof` directory.

The HPROF tool is invoked as follows:

```
$ java -agentlib:hprof ToBeProfiledClass
```

Depending on the type of profiling requested, HPROF instructs the virtual machine to send it the relevant events. The tool then processes the event data into profiling information. For example, the following command obtains the heap allocation profile:

```
$ java -agentlib:hprof=heap=sites ToBeProfiledClass
```

The complete list of options is printed when the HPROF agent is provided with the `help` option, as shown below.

```
$ java -agentlib:hprof=help
     HPROF: Heap and CPU Profiling Agent (JVMTI Demonstration Code)
hprof usage: java -agentlib:hprof=[help]|[<option>=<value>, ...]
Option Name and Value  Description                    Default
---------------------  -----------                    -------
heap=dump|sites|all    heap profiling                 all
cpu=samples|times|old  CPU usage                      off
```

```
monitor=y|n            monitor contention             n
format=a|b             text(txt) or binary output     a
file=<file>            write data to file             java.hprof[{.txt}]
net=<host>:<port>      send data over a socket        off
depth=<size>           stack trace depth              4
interval=<ms>          sample interval in ms          10
cutoff=<value>         output cutoff point            0.0001
lineno=y|n             line number in traces?         y
thread=y|n             thread in traces?              n
doe=y|n                dump on exit?                  y
msa=y|n                Solaris micro state accounting n
force=y|n              force output to <file>         y
verbose=y|n            print messages about dumps     y
Obsolete Options
----------------
gc_okay=y|n
<>
Examples
--------
  - Get sample cpu information every 20 millisec, with a stack depth of 3:
      java -agentlib:hprof=cpu=samples,interval=20,depth=3 classname
  - Get heap usage information based on the allocation sites:
      java -agentlib:hprof=heap=sites classname
Notes
-----
  - The option format=b cannot be used with monitor=y.
  - The option format=b cannot be used with cpu=old|times.
  - Use of the -Xrunhprof interface can still be used, e.g.
       java -Xrunhprof:[help]|[<option>=<value>, ...]
    will behave exactly the same as:
       java -agentlib:hprof=[help]|[<option>=<value>, ...]
Warnings
--------
  - This is demonstration code for the JVMTI interface and use of BCI,
    it is not an official product or formal part of the JDK.
  - The -Xrunhprof interface will be removed in a future release.
  - The option format=b is considered experimental, this format may change
    in a future release.
```

By default, heap profiling information (sites and dump) is written out to `java.hprof.txt` (in ASCII) in the current working directory.

The output is normally generated when the VM exits, although this can be disabled by setting the “dump on exit” option to “n” (`doe=n`). In addition, a profile is generated when Ctrl-\ or Ctrl-Break (depending on platform) is pressed. On Solaris OS and Linux a profile is also generated when a QUIT signal is received (`kill -QUIT` *pid*). If Ctrl-\ or Ctrl-Break is pressed multiple times, multiple profiles are generated to the one file.

The output in most cases will contain IDs for traces, threads, and objects. Each type of ID will typically start with a different number than the other IDs. For example, traces might start with 300000.

2.1.1 Heap Allocation Profiles (`heap=sites`)

The following output is the heap allocation profile generated by running the Java compiler (`javac`) on a set of input files. Only parts of the profiler output are shown here.

```
$ javac -J-agentlib:hprof=heap=sites Hello.java
SITES BEGIN (ordered by live bytes) Wed Oct 4 13:13:42 2006
          percent          live          alloc'ed  stack class
 rank   self  accum     bytes objs     bytes  objs trace name
    1 44.13% 44.13%   1117360 13967  1117360 13967 301926 java.util.zip.ZipEntry
    2  8.83% 52.95%    223472 13967   223472 13967 301927 com.sun.tools.javac.util.List
    3  5.18% 58.13%    131088    1    131088     1 300996 byte[]
    4  5.18% 63.31%    131088    1    131088     1 300995 com.sun.tools.javac.util.Name[]
```

A crucial piece of information in the heap profile is the amount of allocation that occurs in various parts of the program. The SITES record above shows that 44.13% of the total space was allocated for `java.util.zip.ZipEntry` objects.

A good way to relate allocation sites to the source code is to record the dynamic stack traces that led to the heap allocation. The following output shows another part of the profiler output. This output illustrates the stack traces referred to by the four allocation sites in output shown above.

```
TRACE 301926:
        java.util.zip.ZipEntry.<init>(ZipEntry.java:101)
        java.util.zip.ZipFile+3.nextElement(ZipFile.java:417)
        com.sun.tools.javac.jvm.ClassReader.openArchive(ClassReader.java:1374)
        com.sun.tools.javac.jvm.ClassReader.list(ClassReader.java:1631)
TRACE 301927:
        com.sun.tools.javac.util.List.<init>(List.java:42)
        com.sun.tools.javac.util.List.<init>(List.java:50)
        com.sun.tools.javac.util.ListBuffer.append(ListBuffer.java:94)
        com.sun.tools.javac.jvm.ClassReader.openArchive(ClassReader.java:1374)
TRACE 300996:
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:379)
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:481)
        com.sun.tools.javac.util.Name$Table.make(Name.java:332)
        com.sun.tools.javac.util.Name$Table.instance(Name.java:349)
TRACE 300995:
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:378)
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:481)
        com.sun.tools.javac.util.Name$Table.make(Name.java:332)
        com.sun.tools.javac.util.Name$Table.instance(Name.java:349)
```

Each frame in the stack trace contains class name, method name, source file name, and the line number. The user can set the maximum number of frames collected by the HPROF agent. The default limit is four. Stack traces reveal not only which methods performed heap allocation, but also which methods were ultimately responsible for making calls that resulted in memory allocation.

2.1.2 Heap Dump (heap=dump)

A heap dump is obtained using the `heap=dump` option. The heap dump is in either ASCII or binary format, depending on the setting of the `format` option. Tools such as `jhat`(see 2.5 `jhat` Utility) use the binary format and therefore the `format=b` option is required. When the binary format is specified, the dump includes primitive type instance fields and primitive array content.

The following command produces a dump from executing the `javac` compiler.

```
$ javac -J-agentlib:hprof=heap=dump Hello.java
```

The output is a large file. It consists of the root set as determined by the garbage collector, and an entry for each Java object in the heap that can be reached from the root set. The following is a selection of records from a sample heap dump.

```
HEAP DUMP BEGIN (39793 objects, 2628264 bytes) Wed Oct 4 13:54:03 2006
ROOT 50000114 (kind=<thread>, id=200002, trace=300000)
ROOT 50000006 (kind=<JNI global ref>, id=8, trace=300000)
ROOT 50008c6f (kind=<Java stack>, thread=200000, frame=5)
:
CLS 50000006 (name=java.lang.annotation.Annotation, trace=300000)
    loader        90000001
OBJ 50000114 (sz=96, trace=300001, class=java.lang.Thread@50000106)
    name        50000116
    group        50008c6c
    contextClassLoader    50008c53
    inheritedAccessControlContext    50008c79
    blockerLock    50000115
OBJ 50008c6c (sz=48, trace=300000, class=java.lang.ThreadGroup@50000068)
    name        50008c7d
    threads    50008c7c
    groups        50008c7b
ARR 50008c6f (sz=16, trace=300000, nelems=1,
     elem type=java.lang.String[]@5000008e)
    [0]        500007a5
CLS 5000008e (name=java.lang.String[], trace=300000)
    super        50000012
```

```
    loader        90000001
:
HEAP DUMP END
```

Each record is a ROOT, OBJ, CLS, or ARR to represent a root, an object instance, a class, or an array. The hexadecimal numbers are identifiers assigned by HPROF. These numbers are used to show the references from an object to another object. For example, in the above sample, the `java.lang.Thread instance` 50000114 has a reference to its thread group (50008c6c) and other objects.

In general, as the output is very large, it is necessary to use a tool to visualize or process the output of a heap dump. One such tool is `jhat`. See 2.5 `jhat` Utility.

2.1.3 CPU Usage Sampling Profiles (`cpu=samples`)

The HPROF tool can collect CPU usage information by sampling threads. Below is part of the output collected from a run of the `javac` compiler.

```
$ javac -J-agentlib:hprof=cpu=samples Hello.java
CPU SAMPLES BEGIN (total = 462) Wed Oct 4 13:33:07 2006
rank   self  accum   count trace method
   1 49.57% 49.57%     229 300187 java.util.zip.ZipFile.getNextEntry
   2  6.93% 56.49%      32 300190 java.util.zip.ZipEntry.initFields
   3  4.76% 61.26%      22 300122 java.lang.ClassLoader.defineClass2
   4  2.81% 64.07%      13 300188 java.util.zip.ZipFile.freeEntry
   5  1.95% 66.02%       9 300129 java.util.Vector.addElement
   6  1.73% 67.75%       8 300124 java.util.zip.ZipFile.getEntry
   7  1.52% 69.26%       7 300125 java.lang.ClassLoader.findBootstrapClass
   8  0.87% 70.13%       4 300172 com.sun.tools.javac.main.JavaCompiler.<init>
   9  0.65% 70.78%       3 300030 java.util.zip.ZipFile.open
  10  0.65% 71.43%       3 300175 com.sun.tools.javac.main.JavaCompiler.<init>

...
CPU SAMPLES END
```

The HPROF agent periodically samples the stack of all running threads to record the most frequently active stack traces.
The `count` field above indicates how many times a particular stack trace was found to be active. These stack traces correspond to the CPU usage hot spots in the application.

2.1.4 CPU Usage Times Profile (`cpu=times`)

The HPROF tool can collect CPU usage information by injecting code into every method entry and exit, thereby keeping track of exact method call counts and the time spent in each method. This process uses Byte Code Injection (BCI) and runs considerably slower than the `cpu=samples` option. Below is part of the output collected from a run of the `javac` compiler.

```
$ javac -J-agentlib:hprof=cpu=times Hello.java
CPU TIME (ms) BEGIN (total = 2082665289) Wed oct 4 13:43:42 2006
rank   self  accum   count trace method
   1  3.70%  3.70%       1 311243 com.sun.tools.javac.Main.compile
   2  3.64%  7.34%       1 311242 com.sun.tools.javac.main.Main.compile
   3  3.64% 10.97%       1 311241 com.sun.tools.javac.main.Main.compile
   4  3.11% 14.08%       1 311173 com.sun.tools.javac.main.JavaCompiler.compile
   5  2.54% 16.62%       8 306183 com.sun.tools.javac.jvm.ClassReader.listAll
   6  2.53% 19.15%      36 306182 com.sun.tools.javac.jvm.ClassReader.list
   7  2.03% 21.18%       1 307195 com.sun.tools.javac.comp.Enter.main
   8  2.03% 23.21%       1 307194 com.sun.tools.javac.comp.Enter.complete
   9  1.68% 24.90%       1 306392 com.sun.tools.javac.comp.Enter.classEnter
  10  1.68% 26.58%       1 306388 com.sun.tools.javac.comp.Enter.classEnter
...
CPU TIME (ms) END
```

In this output the count represents the true count of the number of times this method was entered, and the percentages represent a measure of thread CPU time spent in those methods.

2.2 Java VisualVM

Java VisualVM is one of the tools included in the JDK download (starting with JDK release 7 update 7). This tool is useful to Java application developers to troubleshoot applications and to monitor and improve the applications' performance. With Java VisualVM you can generate and analyze heap dumps, track down memory leaks, perform and monitor garbage collection, and perform lightweight memory and CPU profiling. The tool is also useful for tuning, heap sizing, offline analysis, and post-mortem diagnosis.

In addition, you can use existing plug-ins that expand the functionality of Java VisualVM. For example, most of the functionality of the JConsole tool is available via the MBeans tab and the JConsole plug-in wrapper tab. You can choose from a catalog of standard Java VisualVM plug-ins by choosing Plugins from the Tools menu in the main Java VisualVM window.

For comprehensive documentation for Java VisualVM,
see http://download.oracle.com/javase/7/docs/technotes/guides/visualvm/index.html

Java VisualVM allows you to perform the following troubleshooting activities:

- View a list of local and remote Java applications.
- View application configuration and runtime environment. For each application, the tool shows basic runtime information: PID, host, main class, arguments passed to the process, JVM version, JDK home, JVM flags, JVM arguments, system properties.
- Enable and disable the creation of a heap dump when a specified application encounters an `OutOfMemoryError` exception.
- Monitor application memory consumption, running threads, and loaded classes.
- Trigger a garbage collection immediately.
- Create a heap dump immediately. You can then view the heap dump in several views: summary, by class, by instance. You can also save the heap dump to your local file system.
- Profile application performance or analyze memory allocation (for local applications only). You can also save the profiling data.
- Create a thread dump (stack trace of the application's active threads) immediately. You can then view the thread dump.
- Analyze core dumps (with Solaris OS and Linux).
- Analyze applications offline, by taking application snapshots.
- Get additional plug-ins contributed by the community.
- Write and share your own plug-ins.
- Display and interact with MBeans (after installing the MBeans tab plug-in).

When you start Java VisualVM, the main Application window opens, displaying a list of Java applications running on the local machine, a list of Java applications running on any connected remote machines, a list of any VM core dumps that were taken and saved (with Solaris OS and Linux), and a list of any application snapshots that were taken and saved.

Java VisualVM will automatically detect and connect to JMX agents for Java applications that are running on JDK 7 or that have been started with the correct system properties on version 5.0. In order for the tool to detect and connect to the agents on a remote machine, the `jstatd` daemon must be running on the remote machine (see 2.13 `jstatd` Daemon). In cases where Java VisualVM cannot automatically discover and connect to JMX agents that are running in a target application, the tool provides a means for you to explicitly create these connections.

2.3 JConsole Utility

Another useful tool included in the JDK download is the JConsole monitoring tool. This tool is compliant with Java Management Extensions (JMX). The tool uses the built-in JMX instrumentation in the Java Virtual Machine to provide information on the performance and resource consumption of running applications. Although the tool is included in the JDK download, it can also be used to monitor and manage applications deployed with the Java runtime environment.

The JConsole tool can attach to any Java application in order to display useful information such as thread usage, memory consumption, and details about class loading, runtime compilation, and the operating system.

This output helps with high-level diagnosis on problems such as memory leaks, excessive class loading, and running threads. It can also be useful for tuning and heap sizing.

In addition to monitoring, JConsole can be used to dynamically change several parameters in the running system. For example, the setting of the `-verbose:gc` option can be changed so that garbage collection trace output can be dynamically enabled or disabled for a running application.

The following list provides an idea of the data that can be monitored using the JConsole tool. Each heading corresponds to a tab pane in the tool.

- Overview

This pane displays graphs showing, over time, heap memory usage, number of threads, number of classes, and CPU usage. This overview allows you to visualize the activity of several resources at once.

- Memory
  - For a selected memory area (heap, non-heap, various memory pools):
    - Graph of memory usage over time
    - Current memory size
    - Amount of committed memory
    - Maximum memory size
  - Garbage collector information, including the number of collections performed, and the total time spent performing garbage collection.
  - Graph showing percentage of heap and non-heap memory currently used.

  In addition, on this pane you can request garbage collection to be performed.

- Threads
  - Graph of thread usage over time.
  - Live threads - Current number of live threads.
  - Peak - Highest number of live threads since the Java VM started.
  - For a selected thread, the name, state, and stack trace, as well as, for a blocked thread, the synchronizer that the thread is waiting to acquire and the thread owning the lock.
  - Deadlock Detection button - Sends a request to the target application to perform deadlock detection and displays each deadlock cycle in a separate tab.
- Classes
  - Graph of number of loaded classes over time.
  - Number of classes currently loaded into memory.
  - Total number of classes loaded into memory since the Java VM started, including those subsequently unloaded.
  - Total number of classes unloaded from memory since the Java VM started.
- VM Summary
  - General information, such as the JConsole connection data, uptime for the Java VM, CPU time consumed by the Java VM, complier name and total compile time, and so forth.
  - Thread and class summary information.
  - Memory and garbage collection information, including number of objects pending finalization, and so forth.
  - Information about the operating system, including physical characteristics, the amount of virtual memory for the running process, swap space, and so forth.
  - Information about the virtual machine itself, such as arguments, class path, and so forth.
- MBeans

  This pane displays a tree structure showing all platform and application MBeans that are registered in the connected JMX agent. When you select an MBean in the tree, its attributes, operations, notifications, and other information are displayed.

  - You can invoke operations, if any. For example, the operation `dumpHeap` for the `HotSpotDiagnostic` MBean, which is in the `com.sun.management` domain, performs a heap dump. The input parameter for this operation is the pathname of the heap dump file on the machine where the target VM is running.
  - As another example of invoking an operation, you can set the value of writable attributes. For example, you can set, unset, or change the value of certain VM flags by invoking the `setVMOption` operation of the `HotSpotDiagnostic` MBean. The flags are indicated by the list of values of the `DiagnosticOptions` attribute.
  - You can subscribe to notifications, if any, by using the Subscribe and Unsubscribe buttons.

JConsole can monitor both local applications and remote applications. If you start the tool with an argument specifying a JMX agent to connect to, the tool will automatically start monitoring the specified application.

To monitor a local application, execute the command `jconsole` *pid*, where *pid* is the process ID of the application.

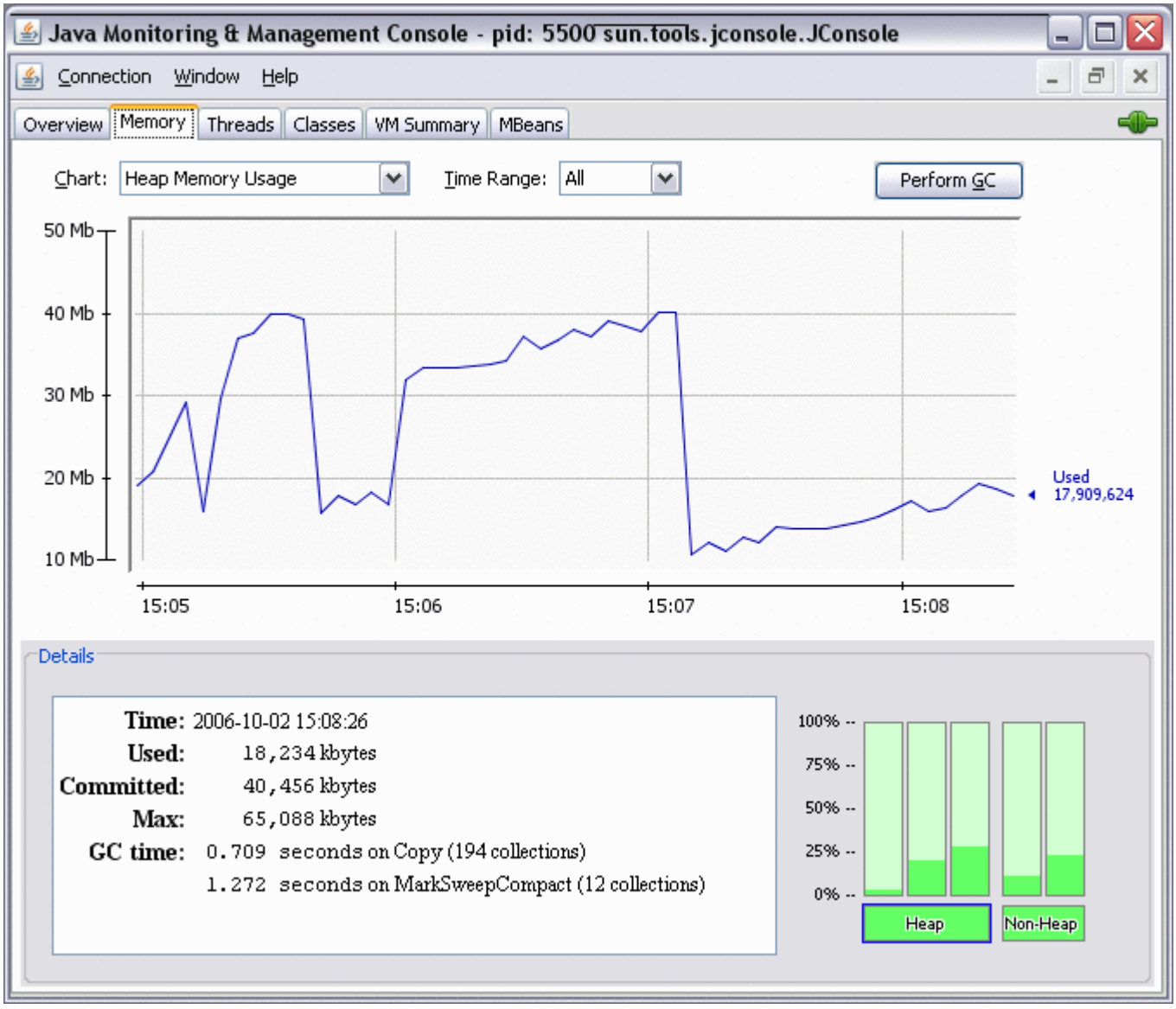
To monitor a remote application, execute the command `jconsole` *hostname*`:`*portnumber*, where *hostname* is the name of the host running the application, and *portnumber* is the port number you specified when you enabled the JMX agent.

If you execute the `jconsole` command without arguments, the tool will start by displaying the New Connection window, where you specify the local or remote process to be monitored. You can connect to a different host at any time by using the Connection menu.

With the JDK 1.5 release, you must start the application to be monitored with the `-Dcom.sun.management.jmxremote` option. With the JDK 7 release, no option is necessary when starting the application to be monitored.

As an example of the output of the monitoring tool, the following screen shows a chart of heap memory usage.

Sample Output from JConsole



A complete tutorial on the JConsole tool is beyond the scope of this document. However, the following documents describe in more detail the monitoring and management capabilities, and how to use JConsole:

- *Monitoring and Management for the Java Platform*

  http://download.oracle.com/javase/7/docs/technotes/guides/management/index.html

- *Monitoring and Management Using JMX*

  http://download.oracle.com/javase/7/docs/technotes/guides/management/agent.html

- Using JConsole

  http://download.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html

- Manual page for `jconsole`

  http://download.oracle.com/javase/7/docs/technotes/tools/share/jconsole.html

2.4 `jdb` Utility

The `jdb` utility is included in the JDK release as the example command-line debugger. The `jdb` utility uses the Java Debug Interface (JDI) to launch or connect to the target VM. The source code for `jdb` is included in `$JAVA_HOME/demo/jpda/examples.jar`.

The Java Debug Interface (JDI) is a high-level Java API that provides information useful for debuggers and similar systems that need access to the running state of a (usually remote) virtual machine. JDI is a component of the Java Platform Debugger Architecture (JPDA). See 2.17.5 Java Platform Debugger Architecture.

In JDI a connector is the means by which the debugger connects to the target virtual machine. The JDK release has traditionally shipped with connectors that launch and establish a debugging session with a target VM, as well as connectors that are used for remote debugging (using TCP/IP or shared memory transports).

This JDK release also ships with several Serviceability Agent (SA) connectors that allow a Java language debugger to attach to a crash dump or hung process. This can be useful in determining what the application was doing at the time of the crash or hang.

These connectors are SACoreAttachingConnector, SADebugServerAttachingConnector, and SAPIDAttachingConnector.

These connectors are generally used with enterprise debuggers, such as as NetBeans IDE or commerical IDEs. The following subsections demonstrate how these connectors can be used with the `jdb` command-line debugger.

For detailed information about the connectors, see http://download.oracle.com/javase/7/docs/technotes/guides/jpda/conninv.html#Connectors.

The command `jdb -listconnectors` prints a list of the available connectors. The command `jdb -help` prints the command usage.

For more information on the `jdb` utility, refer to the manual pages:

- Solaris OS and Linux: `jdb` man page

  http://download.oracle.com/javase/7/docs/technotes/tools/solaris/jdb.html

- Windows: `jdb` man page

  http://download.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html

2.4.1 Attaching to a Process

This example uses the SA PID Attaching Connector to attach to a process. The target process is not started with any special options, that is, the `-agentlib:jdwp` option is not required. When this connector attaches to a process it does so in read-only mode: the debugger can examine threads and the running application but it cannot change anything. The process is frozen while the debugger is attached.

The command in the following example instructs `jdb` to use a connector named `sun.jvm.hotspot.jdi.SAPIDAttachingConnector`. This is a connector name rather than a class name. The connector takes one argument called `pid`, whose value is the process ID of the target process (9302 in this example).

```
$ jdb -connect sun.jvm.hotspot.jdi.SAPIDAttachingConnector:pid=9302

Initializing jdb ...
> threads
Group system:
  (java.lang.ref.Reference$ReferenceHandler)0xa Reference Handler unknown
  (java.lang.ref.Finalizer$FinalizerThread)0x9  Finalizer         unknown
  (java.lang.Thread)0x8                         Signal Dispatcher running
  (java.lang.Thread)0x7                         Java2D Disposer   unknown
  (java.lang.Thread)0x2                         TimerQueue        unknown
Group main:
  (java.lang.Thread)0x6                         AWT-XAWT          running
  (java.lang.Thread)0x5                         AWT-Shutdown      unknown
  (java.awt.EventDispatchThread)0x4             AWT-EventQueue-0  unknown
```

```
  (java.lang.Thread)0x3                         DestroyJavaVM     running
  (sun.awt.image.ImageFetcher)0x1               Image Animator 0  sleeping
  (java.lang.Thread)0x0                         Intro             running
> thread 0x7
Java2D Disposer[1] where
  [1] java.lang.Object.wait (native method)
  [2] java.lang.ref.ReferenceQueue.remove (ReferenceQueue.java:116)
  [3] java.lang.ref.ReferenceQueue.remove (ReferenceQueue.java:132)
  [4] sun.java2d.Disposer.run (Disposer.java:125)
  [5] java.lang.Thread.run (Thread.java:619)
Java2D Disposer[1] up 1
Java2D Disposer[2] where
  [2] java.lang.ref.ReferenceQueue.remove (ReferenceQueue.java:116)
  [3] java.lang.ref.ReferenceQueue.remove (ReferenceQueue.java:132)
  [4] sun.java2d.Disposer.run (Disposer.java:125)
  [5] java.lang.Thread.run (Thread.java:619)
```

In this example the `threads` command is used to get a list of all threads. Then a specific thread is selected with the `thread 0x7` command, and the `where` command is used to get a thread dump. Next the `up 1` command is used to move up one frame in the stack, and the `where` command is used again to get a thread dump.

2.4.2 Attaching to a Core File on the Same Machine

The SA Core Attaching Connector is used to attach the debugger to a core file. The core file may have been created after a crash (see Chapter 4, Troubleshooting System Crashes). The core file can also be obtained by using the `gcore` command on Solaris OS or the `gcore` command in `gdb` on Linux. Because the core file is a snapshot of the process at the time the core file was created, the connector attaches in read-only mode: the debugger can examine threads and the running application at the time of the crash.

The following command is an example of using this connector:

```
$ jdb -connect sun.jvm.hotspot.jdi.SACoreAttachingConnector:\
javaExecutable=$JAVA_HOME/bin/java,core=core.20441
```

This command instructs `jdb` to use a connector named `sun.jvm.hotspot.jdi.SACoreAttachingConnector`. The connector takes two arguments called`javaExecutable` and `core`. The `javaExecutable` argument indicates the name of the Java binary. The `core` argument is the core file name (the core from the process with PID 20441 in this example).

2.4.3 Attaching to a Core File or a Hung Process from a Different Machine

In order to debug a core file that has been transported from another machine, the OS versions and libraries must match. In this case you can first run a proxy server called the SA Debug Server. Then, on the machine where the debugger is installed, you can use the SA Debug Server Attaching Connector to connect to the debug server.

In the example below, there are two machines, machine 1 and machine 2. A core file is available on machine 1 and the debugger is available on machine 2. The SA Debug Server is started on machine 1 as follows.

```
$ jsadebugd $JAVA_HOME/bin/java core.20441
```

The `jsadebugd` command takes two arguments. The first argument is the name of the executable. In most cases this is `java`, but it can be another name (in the case of embedded VMs, for example). The second argument is the name of the core file. In this example the core file was obtained for a process with PID 20441 using the `gcore`utility.

On machine 2, the debugger connects to the remote SA Debug Server using the SA Debug Server Attaching Connector, as with the following command:

```
$ jdb -connect sun.jvm.hotspot.jdi.SADebugServerAttachingConnector:\ debugServerName=machine1
```

This command instructs `jdb` to use a connector named `sun.jvm.hotspot.jdi.SADebugServerAttachingConnector`. The connector has one argument`debugServerName`, which is the hostname or IP address of the machine where the SA Debug Server is running.

Note that the SA Debug Server can also be used to remotely debug a hung process. In that case it takes a single argument which is the process ID of the process. In addition, if it is required to run multiple debug servers on the same machine, each one must be provided with a unique ID. With the SA Debug Server Attaching Connector, this ID is provided as an additional connector argument. These details are described in the JPDA documentation.

2.5 `jhat` Utility

The `jhat` tool provides a convenient means to browse the object topology in a heap snapshot. This tool was introduced in the JDK 6 release to replace the Heap Analysis Tool (HAT).

For more information about the `jhat` utility, see the man page for jhat- Java Heap Analysis Tool.

The tool parses a heap dump in binary format, for example, a heap dump produced by `jmap -dump`.

This utility can help debug unintentional object retention. This term is used to describe an object that is no longer needed but is kept alive due to references through some path from the rootset. This can happen, for example, if an unintentional static reference to an object remains after the object is no longer needed, if an Observer or Listener fails to de-register itself from its subject when it is no longer needed, or if a Thread that refers to an object does not terminate when it should. Unintentional object retention is the Java language equivalent of a memory leak.

The tool provides a number of standard queries. For example, the Roots query displays all reference paths from the rootset to a specified object and is particularly useful for finding unnecessary object retention.

In addition to the standard queries, you can develop your own custom queries with the Object Query Language (OQL) interface.

When you issue the `jhat` command, the utility starts an HTTP server on a specified TCP port. You can then use any browser to connect to the server and execute queries on the specified heap dump.

The following example shows how to execute `jhat` to analyze a heap dump file named `snapshot.hprof`:

```
$ jhat snapshot.hprof
Started HTTP server on port 7000
Reading from java_pid2278.hprof...
Dump file created Fri May 19 17:18:38 BST 2006
Snapshot read, resolving...
Resolving 6162194 objects...
Chasing references, expect 12324 dots.............................
Eliminating duplicate references.................................
Snapshot resolved.
Server is ready.
```

At this point `jhat` has started a HTTP server on port 7000. Point your browser to `http://`*localhost*`:7000` to connect to the `jhat` server.

When you are connected to the server, you can execute the standard queries (see the following subsection) or create an OQL query (see 2.5.2 Custom Queries). The All Classes query is displayed by default.

2.5.1 Standard Queries

The standard queries are described in these subsections.

2.5.1.1 All Classes Query

The default page is the All Classes query, which displays all of the classes present in the heap, excluding platform classes. This list is sorted by fully-qualified class name, and broken out by package. Click on the name of a class to go to the Class query.

The second variant of this query includes the platform classes. Platform classes include classes whose fully-qualified names start with prefixes such as `java`, `sun.`,`javax.swing.`, or `char[`. The list of prefixes is in a system resource file

called `resources/platform_names.txt`. You can override this list by replacing it in the JAR file, or by arranging for your replacement to occur first on the classpath when `jhat` is invoked.

2.5.1.2 Class Query

The Class query displays information about a class. This includes its superclass, any subclasses, instance data members, and static data members. From this page you can navigate to any of the classes that are referenced, or you can navigate to an Instances query.

2.5.1.3 Object Query

The Object query provides information about an object that was on the heap. From here, you can navigate to the class of the object and to the value of any object members of the object. You can also navigate to objects that refer to the current object. Perhaps the most valuable query is at the end: the Roots query (“Reference Chains from Rootset”).

Note that the object query also provides a stack backtrace of the point of allocation of the object.

2.5.1.4 Instances Query

The instances query displays all instances of a given class. The `allInstances` variant includes instances of subclasses of the given class as well. From here, you can navigate back to the source class, or you can navigate to an Object query on one of the instances.

2.5.1.5 Roots Query

The Roots query displays reference chains from the rootset to a given object. It provides one chain for each member of the rootset from which the given object is reachable. When calculating these chains, the tool does a depth-first search, so that it will provide reference chains of minimal length.

There are two kinds of Roots query: one that excludes weak references (Roots), and one that includes them (All Roots). A **weak reference** is a reference object that does not prevent its referent from being made finalizable, finalized, and then reclaimed. If an object is only referred to by a weak reference, it usually isn't considered to be retained, because the garbage collector can collect it as soon as it needs the space.

This is probably the most valuable query in `jhat` for debugging unintentional object retention. Once you find an object that is being retained, this query tells you **why** it is being retained.

2.5.1.6 Reachable Objects Query

This query is accessible from the Object query and shows the transitive closure of all objects reachable from a given object. This list is sorted in decreasing size, and alphabetically within each size. At the end, the total size of all of the reachable objects is given. This can be useful for determining the total runtime footprint of an object in memory, at least in systems with simple object topologies.

This query is most valuable when used in conjunction with the `-exclude` command line option. This is useful, for example, if the object being analyzed is an Observable. By default, all of its Observers would be reachable, which would count against the total size. The `-exclude` option allows you to exclude the data members`java.util.Observable.obs` and `java.util.Observable.arr`.

2.5.1.7 Instance Counts for All Classes Query

This query shows the count of instances for every class in the system, excluding platform classes. It is sorted in descending order, by instance count. A good way to spot a problem with unintentional object retention is to run a program for a long time with a variety of input, then request a heap dump. Looking at the instance counts for all classes, you may recognize a number of classes because there are more instances than you expect. Then you can analyze them to determine why they are being retained (possibly using the Roots query). A variant of this query includes platform classes.

The section on the All Classes query defines platform classes.

2.5.1.8 All Roots Query

This query shows all members of the rootset, including weak references.

2.5.1.9 New Instances Query

The New Instances query is available only if you invoke the `jhat` server with two heap dumps. This query is similar to the Instances query, except that it shows only new instances. An instance is considered new if it is in the second heap dump and there is no object of the same type with the same ID in the baseline heap dump. An object's ID is a 32–bit or 64–bit integer that uniquely identifies the object.

2.5.1.10 Histogram Queries

The built-in histogram and finalizer histogram queries also provide useful information.

2.5.2 Custom Queries

You can develop your own custom queries with the built-in Object Query Language (OQL) interface. Click on the Execute OQL Query button on the first page to display the OQL query page, where you can create and execute your custom queries. The OQL Help facility describes the built-in functions, with examples.

The syntax of the `select` statement is as follows:

```
select JavaScript-expression-to-select
   [ from [instanceof] classname identifier
   [ where JavaScript-boolean-expression-to-filter ] ]
```

The following is an example of a `select` statement:

```
select s from java.lang.String s where s.count >= 100
```

2.5.3 Heap Analysis Hints

To get useful information from `jhat` often requires some knowledge of the application and in addition some knowledge about the libraries and APIs that it uses. However in general the tool can be used to answer two important questions:

- What is keeping an object alive?
- Where was this object allocated?

2.5.3.1 What is keeping an object alive?

When viewing an object instance, you can check the objects listed in the section entitled “References to this object” to see which objects directly reference this object. More importantly you use a Roots query to determine the reference chains from the root set to the given object. These reference chains show a path from a root object to this object. With these chains you can quickly see how an object is reachable from the root set.

As noted earlier, there are two kinds of Roots queries: one that excludes weak references (Roots), and one that includes them (All Roots). A weak reference is a reference object that does not prevent its referent from being made finalizable, finalized, and then reclaimed. If an object is only referred to by a weak reference, it usually is not considered to be retained, because the garbage collector can collect it as soon as it needs the space.

The `jhat` tool sorts the rootset reference chains by the type of the root, in the following order:

- Static data members of Java classes.
- Java local variables. For these roots, the thread responsible for them is shown. Because a Thread is a Java object, this link is clickable. This allows you, for example, to easily navigate to the name of the thread.
- Native static values.
- Native local variables. Again, such roots are identified with their thread.

2.5.3.2 Where was this object allocated?

When an object instance is being displayed, the section entitled “Objects allocated from” shows the allocation site in the form of a stack trace. In this way, you can see where the object was created.

Note that this allocation site information is available only if the heap dump was created with HPROF using the `heap=all` option. This HPROF option includes both the`heap=dump` option and the `heap=sites` option.

If the leak cannot be identified using a single object dump, then another approach is to collect a series of dumps and to focus on the objects created in the interval between each dump. The `jhat` tool provides this capability using the `-baseline` option.

The `-baseline` option allows two dumps to be compared if they were produced by HPROF and from the same VM instance. If the same object appears in both dumps it will be excluded from the list of new objects reported. One dump is specified as a baseline and the analysis can focus on the objects that are created in the second dump since the baseline was obtained.

The following example show how to specify the baseline:

```
$ jhat -baseline snapshot.hprof#1 snapshot.hprof#2
```

In the above example, the two dumps are in the file `snapshot.hprof`, and they are distinguished by appending `#1` and `#2` to the file name.

When `jhat` is started with two heap dumps, the Instance Counts for All Classes query includes an additional column that is the count of the number of new objects for that type. An instance is considered new if it is in the second heap dump and there is no object of the same type with the same ID in the baseline. If you click on a new count, then`jhat` lists the new objects of that type. Then for each instance you can view where it was allocated, which objects these new objects reference, and which other objects reference the new object.

In general, the `-baseline` option can be very useful if the objects that need to be identified are created in the interval between the successive dumps.

2.6 `jinfo` Utility

The `jinfo` command-line utility gets configuration information from a running Java process or crash dump and prints the system properties or the command-line flags that were used to start the virtual machine.

The utility can also use the `jsadebugd` daemon to query a process or core file on a remote machine. Note that the output takes longer to print in this case.

With the `-flag` option, the utility can dynamically set, unset, or change the value of certain Java VM flags for the specified Java process. See B.1.1 Dynamic Changing of Flag Values.

For more information on the `jinfo` utility, refer to the man page.

The following is an example of the output from a Java process.

```
$ jinfo 29620
Attaching to process ID 29620, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100
Java System Properties:

java.runtime.name = Java(TM) SE Runtime Environment
sun.boot.library.path = /usr/jdk/instances/jdk1.6.0/jre/lib/sparc
java.vm.version = 1.6.0-rc-b100
java.vm.vendor = Sun Microsystems Inc.
java.vendor.url = http://java.sun.com/
path.separator = :
java.vm.name = Java HotSpot(TM) Client VM
file.encoding.pkg = sun.io
sun.java.launcher = SUN_STANDARD
sun.os.patch.level = unknown
java.vm.specification.name = Java Virtual Machine Specification
user.dir = /home/js159705
java.runtime.version = 1.6.0-rc-b100
java.awt.graphicsenv = sun.awt.X11GraphicsEnvironment
java.endorsed.dirs = /usr/jdk/instances/jdk1.6.0/jre/lib/endorsed
os.arch = sparc
```

```
java.io.tmpdir = /var/tmp/
line.separator =

java.vm.specification.vendor = Sun Microsystems Inc.
os.name = SunOS
sun.jnu.encoding = ISO646-US
java.library.path =
/usr/jdk/instances/jdk1.6.0/jre/lib/sparc/client:/usr/jdk/instances/jdk1.6.0/jre/lib/sparc:
/usr/jdk/instances/jdk1.6.0/jre/../lib/sparc:/net/gtee.sfbay/usr/sge/sge6/lib/sol-sparc64:
/usr/jdk/packages/lib/sparc:/lib:/usr/lib
java.specification.name = Java Platform API Specification
java.class.version = 50.0
sun.management.compiler = HotSpot Client Compiler
os.version = 5.10
user.home = /home/js159705
user.timezone = US/Pacific
java.awt.printerjob = sun.print.PSPrinterJob
file.encoding = ISO646-US
java.specification.version = 1.6
java.class.path = /usr/jdk/jdk1.6.0/demo/jfc/Java2D/Java2Demo.jar
user.name = js159705
java.vm.specification.version = 1.0
java.home = /usr/jdk/instances/jdk1.6.0/jre
sun.arch.data.model = 32
user.language = en
java.specification.vendor = Sun Microsystems Inc.
java.vm.info = mixed mode, sharing
java.version = 1.6.0-rc
java.ext.dirs = /usr/jdk/instances/jdk1.6.0/jre/lib/ext:/usr/jdk/packages/lib/ext
sun.boot.class.path = /usr/jdk/instances/jdk1.6.0/jre/lib/resources.jar:
/usr/jdk/instances/jdk1.6.0/jre/lib/rt.jar:/usr/jdk/instances/jdk1.6.0/jre/lib/sunrsasign.jar:
/usr/jdk/instances/jdk1.6.0/jre/lib/jsse.jar:
/usr/jdk/instances/jdk1.6.0/jre/lib/jce.jar:/usr/jdk/instances/jdk1.6.0/jre/lib/charsets.jar:
/usr/jdk/instances/jdk1.6.0/jre/classes
java.vendor = Sun Microsystems Inc.
file.separator = /
java.vendor.url.bug = http://java.sun.com/cgi-bin/bugreport.cgi
sun.io.unicode.encoding = UnicodeBig
sun.cpu.endian = big
sun.cpu.isalist =

VM Flags:
```

If you start the target Java VM with the `-classpath` and `-Xbootclasspath` arguments, the output from `jinfo` provides the settings for `java.class.path` and`sun.boot.class.path`. This information might be needed when investigating class loader issues.

In addition to obtaining information from a process, the `jinfo` tool can use a core file as input. On Solaris OS, for example, the `gcore` utility can be used to get a core file of the process in the above example. The core file will be named `core.29620` and will be generated in the working directory of the process. The path to the Java executable and the core file must be specified as arguments to the `jinfo` utility, as in the following example:

```
$ jinfo $JAVA_HOME/bin/java core.29620
```

Sometimes the binary name will not be `java`. This occurs when the VM is created using the JNI invocation API. The `jinfo` tool requires the binary from which the core file was generated.

2.7 `jmap` Utility

The `jmap` command-line utility prints memory related statistics for a running VM or core file.

The utility can also use the `jsadebugd` daemon to query a process or core file on a remote machine. Note that the output takes longer to print in this case.

If `jmap` is used with a process or core file without any command-line options, then it prints the list of shared objects loaded (the output is similar to the `pmap` utility on Solaris OS). For more specific information, you can use the options `-heap`, `-histo`, or `-permstat`. These options are described in the subsections that follow.

In addition, the JDK 7 release introduced the `-dump:format=b,file=`*filename* option, which causes `jmap` to dump the Java heap in binary `HPROF` format to a specified file. This file can then be analyzed with the `jhat` tool.

If the `jmap` *pid* command does not respond because of a hung process, the `-F` option can be used (on Solaris OS and Linux only) to force the use of the Serviceability Agent.

For more information on the `jmap` utility, refer to the manual page.

2.7.1 Heap Configuration and Usage

The `-heap` option is used to obtain the following Java heap information:

- Information specific to the garbage collection (GC) algorithm , including the name of the GC algorithm (Parallel GC for example) and algorithm specific details (such as number of threads for parallel GC).
- Heap configuration. The heap configuration might have been specified as command line options or selected by the VM based on the machine configuration.
- Heap usage summary. For each generation (area of the heap), the tool prints the total heap capacity, in-use memory, and available free memory. If a generation is organized as a collection of spaces (the new generation for example), then a space-wise memory size summary is included.

The following example shows output from the `jmap -heap` command.

```
$ jmap -heap 29620
Attaching to process ID 29620, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100

using thread-local object allocation.
Mark Sweep Compact GC

Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70
   MaxHeapSize      = 67108864 (64.0MB)
   NewSize          = 2228224 (2.125MB)
   MaxNewSize       = 4294901760 (4095.9375MB)
   OldSize          = 4194304 (4.0MB)
   NewRatio         = 8
   SurvivorRatio    = 8
   PermSize         = 12582912 (12.0MB)
   MaxPermSize      = 67108864 (64.0MB)

Heap Usage:
New Generation (Eden + 1 Survivor Space):
   capacity = 2031616 (1.9375MB)
   used     = 70984 (0.06769561767578125MB)
   free     = 1960632 (1.8698043823242188MB)
   3.4939673639112905% used
Eden Space:
   capacity = 1835008 (1.75MB)
   used     = 36152 (0.03447723388671875MB)
   free     = 1798856 (1.7155227661132812MB)
   1.970127650669642 8% used
From Space:
   capacity = 196608 (0.1875MB)
   used     = 34832 (0.0332183837890625MB)
   free     = 161776 (0.1542816162109375MB)
   17.716471354166668% used
To Space:
   capacity = 196608 (0.1875MB)
   used     = 0 (0.0MB)
   free     = 196608 (0.1875MB)
```

```
   0.0% used
tenured generation:
   capacity = 15966208 (15.2265625MB)
   used     = 9577760 (9.134063720703125MB)
   free     = 6388448 (6.092498779296875MB)
   59.98769400974859% used
Perm Generation:
   capacity = 12582912 (12.0MB)
   used     = 1469408 (1.401336669921875MB)
   free     = 11113504 (10.598663330078125MB)
   11.677805582682291% used
```

2.7.2 Heap Histogram of Running Process

The `-histo` option can be used to obtain a class-wise histogram of the heap.

When the command is executed on a running process, the tool prints the number of objects, memory size in bytes, and fully qualified class name for each class. Internal classes in the HotSpot VM are enclosed in angle brackets. The histogram is useful in understanding how the heap is used. To get the size of an object you must divide the total size by the count of that object type.

The following example shows output from the `jmap -histo` command when it is executed on a process.

```
$ jmap -histo 29620
num   #instances    #bytes  class name
----------------------------------------
  1:      1414     6013016  [I
  2:       793      482888  [B
  3:      2502      334928  <constMethodKlass>
  4:       280      274976  <instanceKlassKlass>
  5:       324      227152  [D
  6:      2502      200896  <methodKlass>
  7:      2094      187496  [C
  8:       280      172248  <constantPoolKlass>
  9:      3767      139000  [Ljava.lang.Object;
 10:       260      122416  <constantPoolCacheKlass>
 11:      3304      112864  <symbolKlass>
 12:       160       72960  java2d.Tools$3
 13:       192       61440  <objArrayKlassKlass>
 14:       219       55640  [F
 15:      2114       50736  java.lang.String
 16:      2079       49896  java.util.HashMap$Entry
 17:       528       48344  [S
 18:      1940       46560  java.util.Hashtable$Entry
 19:       481       46176  java.lang.Class
 20:        92       43424  javax.swing.plaf.metal.MetalScrollButton
... more lines removed here to reduce output...
1118:         1           8  java.util.Hashtable$EmptyIterator
1119:         1           8  sun.java2d.pipe.SolidTextRenderer
Total    61297    10152040
```

2.7.3 Heap Histogram of Core File

When the `jmap -histo` command is executed on a core file, the tool prints the size, count, and class name for each class. Internal classes in the HotSpot VM are prefixed with an asterisk (*).

```
& jmap -histo /net/koori.sfbay/onestop/jdk/6.0/promoted/all/b100/binaries/ solaris-
sparcv9/bin/java core
Attaching to core core from executable /net/koori.sfbay/onestop/jdk/6.0/
promoted/all/b100/binaries/solaris-sparcv9/bin/java, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 1.6.0-rc-b100
Iterating over heap. This may take a while...
Heap traversal took 8.902 seconds.

Object Histogram:
```

```
Size    Count    Class description
-------------------------------------------------------
4151816    2941    int[]
2997816    26403    * ConstMethodKlass
2118728    26403    * MethodKlass
1613184    39750    * SymbolKlass
1268896    2011    * ConstantPoolKlass
1097040    2011    * InstanceKlassKlass
882048    1906    * ConstantPoolCacheKlass
758424    7572    char[]
733776    2518    byte[]
252240    3260    short[]
214944    2239    java.lang.Class
177448    3341    * System ObjArray
176832    7368    java.lang.String
137792    3756    java.lang.Object[]
121744    74    long[]
72960    160    java2d.Tools$3
63680    199    * ObjArrayKlassKlass
53264    158    float[]
... more lines removed here to reduce output...
```

2.7.4 Getting Information on the Permanent Generation

The permanent generation is the area of heap that holds all the reflective data of the virtual machine itself, such as class and method objects (also called “method area” in The Java Virtual Machine Specification).

Configuring the size of the permanent generation can be important for applications that dynamically generate and load a very large number of classes (for example, Java Server Pages/web containers). If an application loads “too many” classes, then it is possible it will abort with an `OutOfMemoryError`. The specific error is `Exception in thread XXXX java.lang.OutOfMemoryError: PermGen space`. See 3.1 Meaning of `OutOfMemoryError` for a description of this and other reasons for`OutOfMemoryError`.

To get further information about the permanent generation, you can use the `-permstat` option to print statistics for the objects in the permanent generation. The following example shows output from the `jmap -permstat` command.

```
$ jmap -permstat 29620
Attaching to process ID 29620, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100
12674 intern Strings occupying 1082616 bytes.
finding class loader instances ..Unknown oop at 0xd0400900
Oop's klass is 0xd0bf8408
Unknown oop at 0xd0401100
Oop's klass is null
done.
computing per loader stat ..done.
please wait.. computing liveness.....................................done.
class_loader    classes bytes   parent_loader   alive?  type

<bootstrap>     1846 5321080  null        live   <internal>
0xd0bf3828  0      0      null         live    sun/misc/Launcher$ExtClassLoader@0xd8c98c78
0xd0d2f370  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0c99280  1   1440      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0b71d90  0      0   0xd0b5b9c0    live java/util/ResourceBundle$RBClassLoader@0xd8d042e8
0xd0d2f4c0  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0b5bf98  1    920   0xd0b5bf38      dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0c99248  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f488  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0b5bf38  6   11832  0xd0b5b9c0      dead    sun/reflect/misc/MethodUtil@0xd8e8e560
0xd0d2f338  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f418  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f3a8  1    904     null          dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0b5b9c0  317 1397448 0xd0bf3828     live    sun/misc/Launcher$AppClassLoader@0xd8cb83d8
0xd0d2f300  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f3e0  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
```

```
0xd0ec3968  1   1440      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0e0a248  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0c99210  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f450  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0d2f4f8  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50
0xd0e0a280  1    904      null         dead    sun/reflect/DelegatingClassLoader@0xd8c22f50

total = 22      2186    6746816   N/A   alive=4, dead=18       N/A
```

For each class loader object, the following details are printed:

- The address of the class loader object at the snapshot when the utility was run
- The number of classes loaded
- The approximate number of bytes consumed by meta-data for all classes loaded by this class loader
- The address of the parent class loader (if any)
- A “live” or “dead” indication of whether the loader object will be garbage collected in the future
- The class name of this class loader

2.8 `jps` Utility

The `jps` utility lists the instrumented HotSpot Virtual Machines for the current user on the target system. The utility is very useful in environments where the VM is embedded, that is, it is started using the JNI Invocation API rather than the `java` launcher. In these environments it is not always easy to recognize the Java processes in the process list.

The following example demonstrates the usage of the `jps` utility.

```
$ jps
16217 MyApplication
16342 jps
```

The utility lists the virtual machines for which the user has access rights. This is determined by operating-system-specific access-control mechanisms. On Solaris OS, for example, if a non-root user executes the `jps` utility, the output is a list of the virtual machines that were started with that user's `uid`.

In addition to listing the process ID, the utility provides options to output the arguments passed to the application's main method, the complete list of VM arguments, and the full package name of the application's main class. The `jps` utility can also list processes on a remote system if the remote system is running the `jstat` daemon (`jstatd`).

If you are running several Java Web Start applications on a system, they tend to look the same, as shown in the following example:

```
$ jps
1271 jps
     1269 Main
     1190 Main
```

In this case, use `jps -m` to distinguish them, as follows:

```
$ jps -m
1271 jps -m
     1269 Main http://bugster.central.sun.com/bugster.jnlp
     1190 Main http://webbugs.sfbay/IncidentManager/incident.jnlp
```

For more information on the `jps` utility, refer to the man page.

The utility is included in the JDK download for all operating system platforms supported by Sun.

NOTE - THE HOTSPOT INSTRUMENTATION IS NOT ACCESSIBLE ON WINDOWS 98 OR WINDOWS ME. IN ADDITION, THE INSTRUMENTATION MIGHT NOT BE ACCESSIBLE ON WINDOWS IF THE TEMPORARY DIRECTORY IS ON A FAT32 FILE SYSTEM.

2.9 `jrunscript` Utility

The `jrunscript` utility is a command-line script shell. It supports script execution in both interactive mode and in batch mode. By default, the shell uses JavaScript, but you can specify any other scripting language for which you supply the path to the script engines's JAR file of `.class files.`

Thanks to the communication between the Java language and the scripting language, the `jrunscript` utility supports an **exploratory programming** style.

For more information on the `jrunscript` utility, refer to the man page.

2.10 `jsadebugd` Daemon

The Serviceability Agent Debug Daemon (`jsadebugd`) attaches to a Java process or to a core file and acts as a debug server. This utility is currently available only on Solaris OS and Linux. Remote clients such as `jstack`, `jmap`, and `jinfo` can attach to the server using Java Remote Method Invocation (RMI).

For more information on `jsadebugd`, refer to the man page.

2.11 `jstack` Utility

The `jstack` command-line utility attaches to the specified process or core file and prints the stack traces of all threads that are attached to the virtual machine, including Java threads and VM internal threads, and optionally native stack frames. The utility also performs deadlock detection.

The utility can also use the `jsadebugd` daemon to query a process or core file on a remote machine. Note that the output takes longer to print in this case.

A stack trace of all threads can be useful in diagnosing a number of issues such as deadlocks or hangs.

The JDK 7 release introduced the `-l` option, which instructs the utility to look for ownable synchronizers in the heap and print information about`java.util.concurrent.locks`. Without this option, the thread dump includes information only on monitors.

Starting with JDK 7, the output from the `jstack` *pid* option is the same as that obtained by pressing Ctrl-\ at the application console (standard input) or by sending the process a QUIT signal. See 2.15 Ctrl-Break Handler for an output example.

Thread dumps can also be obtained programmatically using the `Thread.getAllStackTraces` method, or in the debugger using the debugger option to print all thread stacks (the `where` command in the case of the `jdb` sample debugger).

For more information on the `jstack` utility , refer to the man page.

2.11.1 Forcing a Stack Dump

If the `jstack` *pid* command does not respond because of a hung process, the `-F` option can be used (on Solaris OS and Linux only) to force a stack dump, as in the following example:

```
$ jstack -F 8321
Attaching to process ID 8321, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100
Deadlock Detection:

Found one Java-level deadlock:
============================
```

```
"Thread2":
  waiting to lock Monitor@0x000af398 (Object@0xf819aa10, a java/lang/String),
  which is held by "Thread1"
"Thread1":
  waiting to lock Monitor@0x000af400 (Object@0xf819aa48, a java/lang/String),
  which is held by "Thread2"

Found a total of 1 deadlock.

Thread t@2: (state = BLOCKED)

Thread t@11: (state = BLOCKED)
 - Deadlock$DeadlockMakerThread.run() @bci=108, line=32 (Interpreted frame)

Thread t@10: (state = BLOCKED)
 - Deadlock$DeadlockMakerThread.run() @bci=108, line=32 (Interpreted frame)

Thread t@6: (state = BLOCKED)

Thread t@5: (state = BLOCKED)
 - java.lang.Object.wait(long) @bci=-1107318896 (Interpreted frame)
 - java.lang.Object.wait(long) @bci=0 (Interpreted frame)
 - java.lang.ref.ReferenceQueue.remove(long) @bci=44, line=116 (Interpreted frame)
 - java.lang.ref.ReferenceQueue.remove() @bci=2, line=132 (Interpreted frame)
 - java.lang.ref.Finalizer$FinalizerThread.run() @bci=3, line=159 (Interpreted frame)

Thread t@4: (state = BLOCKED)
 - java.lang.Object.wait(long) @bci=0 (Interpreted frame)
 - java.lang.Object.wait(long) @bci=0 (Interpreted frame)
 - java.lang.Object.wait() @bci=2, line=485 (Interpreted frame)
 - java.lang.ref.Reference$ReferenceHandler.run() @bci=46, line=116 (Interpreted frame)
```

2.11.2 Printing Stack Trace From Core Dump

To obtain stack traces from a core dump, execute the following command:

```
$ jstack $JAVA_HOME/bin/java core
```

2.11.3 Printing a Mixed Stack

The `jstack` utility can also be used to print a mixed stack, that is, it can print native stack frames in addition to the Java stack. Native frames are the C/C++ frames associated with VM code and JNI/native code.

To print a mixed stack, use the `-m` option, as in the following example:

```
$ jstack -m 21177
Attaching to process ID 21177, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100
Deadlock Detection:

Found one Java-level deadlock:
=============================

"Thread1":
  waiting to lock Monitor@0x0005c750 (Object@0xd4405938, a java/lang/String),
  which is held by "Thread2"
"Thread2":
  waiting to lock Monitor@0x0005c6e8 (Object@0xd4405900, a java/lang/String),
  which is held by "Thread1"

Found a total of 1 deadlock.

----------------- t@1 -----------------
```

```
0xff2c0fbc    __lwp_wait + 0x4
0xff2bc9bc    _thrp_join + 0x34
0xff2bcb28    thr_join + 0x10
0x00018a04    ContinueInNewThread + 0x30
0x00012480    main + 0xeb0
0x000111a0    _start + 0x108
----------------- t@2 -----------------
0xff2c1070    ___lwp_cond_wait + 0x4
0xfec03638    bool Monitor::wait(bool,long) + 0x420
0xfec9e2c8    bool Threads::destroy_vm() + 0xa4
0xfe93ad5c    jni_DestroyJavaVM + 0x1bc
0x00013ac0    JavaMain + 0x1600
0xff2bfd9c    _lwp_start
----------------- t@3 -----------------
0xff2c1070    ___lwp_cond_wait + 0x4
0xff2ac104    _lwp_cond_timedwait + 0x1c
0xfec034f4    bool Monitor::wait(bool,long) + 0x2dc
0xfece60bc    void VMThread::loop() + 0x1b8
0xfe8b66a4    void VMThread::run() + 0x98
0xfec139f4    java_start + 0x118
0xff2bfd9c    _lwp_start
----------------- t@4 -----------------
0xff2c1070    ___lwp_cond_wait + 0x4
0xfec195e8    void os::PlatformEvent::park() + 0xf0
0xfec88464    void ObjectMonitor::wait(long long,bool,Thread*) + 0x548
0xfe8cb974    void ObjectSynchronizer::wait(Handle,long long,Thread*) + 0x148
0xfe8cb508    JVM_MonitorWait + 0x29c
0xfc40e548    * java.lang.Object.wait(long) bci:0 (Interpreted frame)
0xfc40e4f4    * java.lang.Object.wait(long) bci:0 (Interpreted frame)
0xfc405a10    * java.lang.Object.wait() bci:2 line:485 (Interpreted frame)
... more lines removed here to reduce output...
----------------- t@12 -----------------
0xff2bfe3c    __lwp_park + 0x10
0xfe9925e4    AttachOperation*AttachListener::dequeue() + 0x148
0xfe99115c    void attach_listener_thread_entry(JavaThread*,Thread*) + 0x1fc
0xfec99ad8    void JavaThread::thread_main_inner() + 0x48
0xfec139f4    java_start + 0x118
0xff2bfd9c    _lwp_start
----------------- t@13 -----------------
0xff2c1500    _door_return + 0xc
----------------- t@14 -----------------
0xff2c1500    _door_return + 0xc
```

Frames that are prefixed with '*' are Java frames, while frames that are not prefixed with '*' are native C/C++ frames.

The output of the utility can be piped through `c++filt` to demangle C++ mangled symbol names. Because the HotSpot Virtual Machine is developed in the C++ language, the `jstack` utility prints C++ mangled symbol names for the HotSpot internal functions. The `c++filt` utility is delivered with the native `c++` compiler suite: `SUNWspro` on Solaris OS and `gnu` on Linux.

2.12 `jstat` Utility

The `jstat` utility uses the built-in instrumentation in the HotSpot VM to provide information on performance and resource consumption of running applications. The tool can be used when diagnosing performance issues, and in particular issues related to heap sizing and garbage collection. The `jstat` utility does not require the VM to be started with any special options. The built-in instrumentation in the HotSpot VM is enabled by default. The utility is included in the JDK download for all operating system platforms supported by Sun.

NOTE - THE INSTRUMENTATION IS NOT ACCESSIBLE ON WINDOWS 98 OR WINDOWS ME. IN ADDITION, INSTRUMENTATION IS NOT ACCESSIBLE ON WINDOWS NT, 2000, OR XP IF A FAT32 FILE SYSTEM IS USED.

The following list presents the options for the `jstat` utility.

- `class` - prints statistics on the behavior of the class loader.
- `compiler` - prints statistics of the behavior of the HotSpot compiler.
- `gc` - prints statistics of the behavior of the garbage collected heap.
- `gccapacity` - prints statistics of the capacities of the generations and their corresponding spaces.
- `gccause` - prints the summary of garbage collection statistics (same as -gcutil), with the cause of the last and current (if applicable) garbage collection events.
- `gcnew` - prints statistics of the behavior of the new generation.
- `gcnewcapacity` - prints statistics of the sizes of the new generations and its corresponding spaces.
- `gcold` - prints statistics of the behavior of the old and permanent generations.
- `gcoldcapacity` - prints statistics of the sizes of the old generation.
- `gcpermcapacity` - prints statistics of the sizes of the permanent generation.
- `gcutil` - prints a summary of garbage collection statistics.
- `printcompilation` - prints HotSpot compilation method statistics.

For a complete description of the `jstat` utility, refer to the man page.

The documentation includes a number of examples, and a few of those examples are repeated here in this document.

The `jstat` utility uses a `vmid` to identify the target process. The documentation describes the syntax of a `vmid` but in the simplest case a `vmid` can be a local virtual machine identifier. In the case of Solaris OS, Linux, and Windows, it can be considered to be the process ID. Note that this is typical but may not always be the case.

The `jstat` tool provides data similar to the data provided by the tools `vmstat` and `iostat` on Solaris OS and Linux.

For a graphical representation of the data, you can use the `visualgc` tool. See 2.14 `visualgc` Tool.

2.12.1 Example of `-gcutil` Option

Below is an example of the `-gcutil` option. The utility attaches to `lvmid` 2834, takes nine samples at 250 millisecond intervals, and displays the output.

```
$ jstat -gcutil 2834 250 9
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00  87.14  46.56  96.82     54    1.197   140   86.559   87.757
  0.00   0.00  91.90  46.56  96.82     54    1.197   140   86.559   87.757
  0.00   0.00 100.00  46.56  96.82     54    1.197   140   86.559   87.757
  0.00  27.12   5.01  54.60  96.82     55    1.215   140   86.559   87.774
  0.00  27.12  11.22  54.60  96.82     55    1.215   140   86.559   87.774
  0.00  27.12  13.57  54.60  96.82     55    1.215   140   86.559   87.774
  0.00  27.12  18.05  54.60  96.82     55    1.215   140   86.559   87.774
  0.00  27.12  23.85  54.60  96.82     55    1.215   140   86.559   87.774
  0.00  27.12  27.32  54.60  96.82     55    1.215   140   86.559   87.774
```

The output of this example shows that a young generation collection occurred between the third and fourth samples. The collection took 0.017 seconds and promoted objects from the `eden` space (E) to the `old` space (O), resulting in an increase of old space utilization from 46.56% to 54.60%.

2.12.2 Example of `-gcnew` Option

The following example illustrates the `-gcnew` option. The utility attaches to `lvmid` 2834, takes samples at 250 millisecond intervals, and displays the output. In addition, it uses the `-h3` option to display the column header after every 3 lines of data.

```
$ jstat -gcnew -h3 2834 250
S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
 192.0  192.0    0.0    0.0 15  15   96.0   1984.0    942.0    218    1.999
 192.0  192.0    0.0    0.0 15  15   96.0   1984.0   1024.8    218    1.999
 192.0  192.0    0.0    0.0 15  15   96.0   1984.0   1068.1    218    1.999
 S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
 192.0  192.0    0.0    0.0 15  15   96.0   1984.0   1109.0    218    1.999
 192.0  192.0    0.0  103.2  1  15   96.0   1984.0      0.0    219    2.019
 192.0  192.0    0.0  103.2  1  15   96.0   1984.0     71.6    219    2.019
```

```
 S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
 192.0  192.0    0.0  103.2  1  15   96.0   1984.0     73.7    219    2.019
 192.0  192.0    0.0  103.2  1  15   96.0   1984.0     78.0    219    2.019
 192.0  192.0    0.0  103.2  1  15   96.0   1984.0    116.1    219    2.019
```

In addition to showing the repeating header string, this example shows that between the fourth and fifth samples, a young generation collection occurred, whose duration was 0.02 seconds. The collection found enough live data that the survivor space 0 utilization (S1U) would have exceeded the desired survivor size (DSS). As a result, objects were promoted to the old generation (not visible in this output), and the tenuring threshold (TT) was lowered from 15 to 1.

2.12.3 Example of `-gcoldcapacity` Option

The following example illustrates the `-gcoldcapacity` option. The utility attaches to `lvmid` 21891 and takes 3 samples at 250 millisecond intervals. The `-t` option is used to generate a time stamp for each sample in the first column.

```
$ jstat -gcoldcapacity -t 21891 250 3
Timestamp    OGCMN     OGCMX       OGC         OC   YGC   FGC     FGCT     GCT
    150.1   1408.0   60544.0   11696.0   11696.0   194    80    2.874   3.799
    150.4   1408.0   60544.0   13820.0   13820.0   194    81    2.938   3.863
    150.7   1408.0   60544.0   13820.0   13820.0   194    81    2.938   3.863
```

The Timestamp column reports the elapsed time in seconds since the start of the target Java VM. In addition, the `-gcoldcapacity` output shows the old generation capacity (OGC) and the old space capacity (OC) increasing as the heap expands to meet allocation or promotion demands. The old generation capacity (OGC) has grown from 11696 KB to 13820 KB after the 81st Full GC (FGC). The maximum capacity of the generation (and space) is 60544 KB (OGCMX), so it still has room to expand.

2.13 `jstatd` Daemon

The `jstatd` daemon is a Remote Method Invocation (RMI) server application that monitors the creation and termination of instrumented Java HotSpot virtual machines and provides an interface to allow remote monitoring tools to attach to Java VMs running on the local host. For example, this daemon allows the `jps` utility to list processes on a remote system.

NOTE - THE INSTRUMENTATION IS NOT ACCESSIBLE ON WINDOWS 98 OR WINDOWS ME. IN ADDITION, INSTRUMENTATION IS NOT ACCESSIBLE ON WINDOWS NT, 2000, OR XP IF A FAT32 FILE SYSTEM IS USED.

For more information about the `jstatd` daemon, including detailed usage examples, refer to the man page.

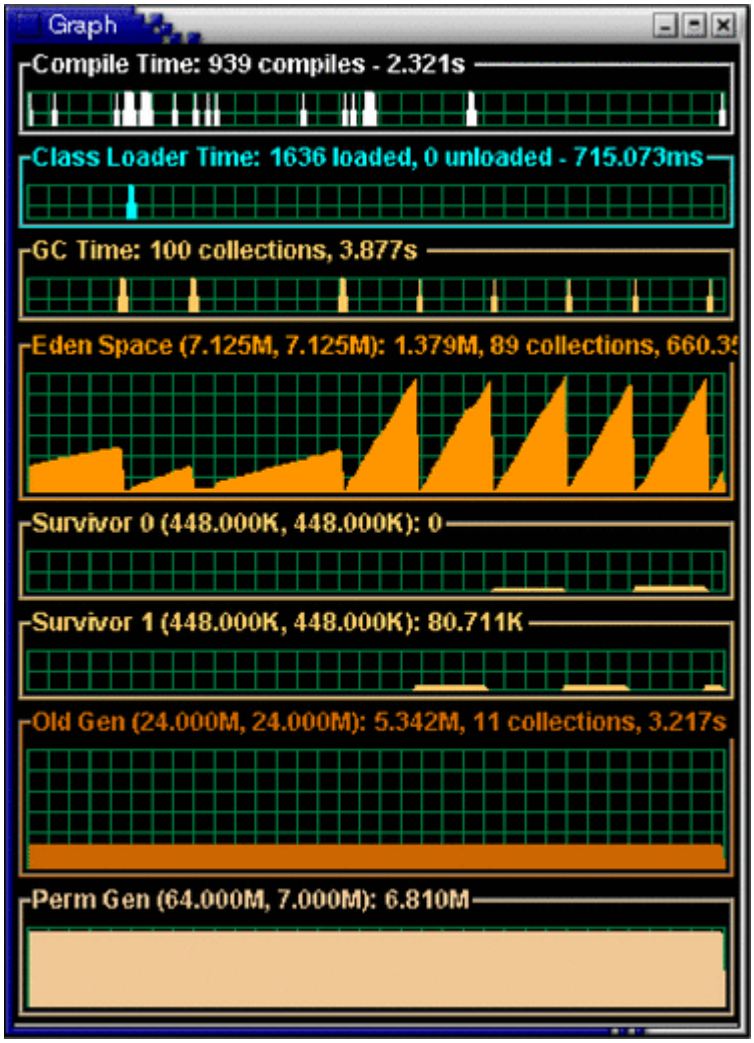
2.14 `visualgc` Tool

The `visualgc` tool is related to the `jstat` tool. (See 2.12 `jstat` Utility.) The `visualgc` tool provides a graphical view of the garbage collection (GC) system. As with`jstat`, it uses the built-in instrumentation of the HotSpot VM.

The `visualgc` tool is not included in the JDK release but is available as a separate download from the jvmstat 3.0 site.

The following screen output demonstrates how the GC and heap are visualized.

Sample Output from `visualgc`



2.15 Ctrl-Break Handler

On Solaris OS or Linux, the combination of pressing the Ctrl key and the backslash (\) key at the application console (standard input) causes the HotSpot VM to print a thread dump to the application's standard output. On Windows the equivalent key sequence is the Ctrl and Break keys. The general term for these key combinations is the Ctrl-Break handler.

On Solaris OS and Linux, a thread dump is printed if the Java process receives a QUIT signal. Therefore, the `kill -QUIT` *pid* command causes the process with ID *pid* to print a thread dump to the standard output.

The following subsections explain in detail the output from the Ctrl-Break handler:

2.15.1 Thread Dump

The thread dump consists of the thread stack, including thread state, for all Java threads in the virtual machine. The thread dump does not terminate the application: it continues after the thread information is printed.

The following example illustrates a thread dump.

```
Full thread dump Java HotSpot(TM) Client VM (1.6.0-rc-b100 mixed mode):

"DestroyJavaVM" prio=10 tid=0x00030400 nid=0x2 waiting on condition [0x00000000..0xfe77fbf0]
   java.lang.Thread.State: RUNNABLE

"Thread2" prio=10 tid=0x000d7c00 nid=0xb waiting for monitor entry [0xf36ff000..0xf36ff8c0]
```

```
   java.lang.Thread.State: BLOCKED (on object monitor)
        at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
        - waiting to lock <0xf819a938> (a java.lang.String)
        - locked <0xf819a970> (a java.lang.String)

"Thread1" prio=10 tid=0x000d6c00 nid=0xa waiting for monitor entry [0xf37ff000..0xf37ffbc0]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
        - waiting to lock <0xf819a970> (a java.lang.String)
        - locked <0xf819a938> (a java.lang.String)

"Low Memory Detector" daemon prio=10 tid=0x000c7800 nid=0x8 runnable [0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x000c5400 nid=0x7 waiting on condition
[0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=10 tid=0x000c4400 nid=0x6 waiting on condition
[0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"Finalizer" daemon prio=10 tid=0x000b2800 nid=0x5 in Object.wait() [0xf3f7f000..0xf3f7f9c0]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xf4000b40> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
        - locked <0xf4000b40> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=10 tid=0x000ae000 nid=0x4 in Object.wait()
[0xfe57f000..0xfe57f940]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xf4000a40> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:485)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
        - locked <0xf4000a40> (a java.lang.ref.Reference$Lock)

"VM Thread" prio=10 tid=0x000ab000 nid=0x3 runnable

"VM Periodic Task Thread" prio=10 tid=0x000c8c00 nid=0x9 waiting on condition
```

The output consists of a header and a stack trace for each thread. Each thread is separated by an empty line. The Java threads (threads that are capable of executing Java language code) are printed first, and these are followed by information on VM internal threads.

The header line contains the following information about the thread:

- Thread name
- Indication if the thread is a daemon thread
- Thread priority (`prio`)
- Thread ID (`tid`), which is the address of a thread structure in memory
- ID of the native thread (`nid`)
- Thread state, which indicates what the thread was doing at the time of the thread dump
- Address range, which gives an estimate of the valid stack region for the thread

The following table lists the possible thread states that can be printed.

Thread State	Meaning
NEW	The thread has not yet started.
RUNNABLE	The thread is executing in the Java virtual machine.

BLOCKED	The thread is blocked waiting for a monitor lock.
WAITING	The thread is waiting indefinitely for another thread to perform a particular action.
TIMED_WAITING	The thread is waiting for another thread to perform an action for up to a specified waiting time.
TERMINATED	The thread has exited.

The thread header is followed by the thread stack.

2.15.2 Deadlock Detection

In addition to the thread stacks, the Ctrl-Break handler executes a deadlock detection algorithm. If any deadlocks are detected, it prints additional information after the thread dump on each deadlocked thread.

```
Found one Java-level deadlock:
=============================
"Thread2":
  waiting to lock monitor 0x000af330 (object 0xf819a938, a java.lang.String),
  which is held by "Thread1"
"Thread1":
  waiting to lock monitor 0x000af398 (object 0xf819a970, a java.lang.String),
  which is held by "Thread2"

Java stack information for the threads listed above:
===================================================
"Thread2":
        at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
        - waiting to lock <0xf819a938> (a java.lang.String)
        - locked <0xf819a970> (a java.lang.String)
"Thread1":
        at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
        - waiting to lock <0xf819a970> (a java.lang.String)
        - locked <0xf819a938> (a java.lang.String)

Found 1 deadlock.
```

If the Java VM flag `-XX:+PrintConcurrentLocks` is set, Ctrl-Break will also print the list of concurrent locks owned by each thread.

2.15.3 Heap Summary

Starting with JDK 7, the Ctrl-Break handler also prints a heap summary. This output shows the different generations (areas of the heap), with the size, the amount used, and the address range. The address range is especially useful if you are also examining the process with tools such as `pmap`.

```
Heap
 def new generation   total 1152K, used 435K [0x22960000, 0x22a90000, 0x22e40000
)
  eden space 1088K,  40% used [0x22960000, 0x229ccd40, 0x22a70000)
  from space 64K,   0% used [0x22a70000, 0x22a70000, 0x22a80000)
  to   space 64K,   0% used [0x22a80000, 0x22a80000, 0x22a90000)
 tenured generation   total 13728K, used 6971K [0x22e40000, 0x23ba8000, 0x269600
00)
   the space 13728K,  50% used [0x22e40000, 0x2350ecb0, 0x2350ee00, 0x23ba8000)
 compacting perm gen  total 12288K, used 1417K [0x26960000, 0x27560000, 0x2a9600
00)
   the space 12288K,  11% used [0x26960000, 0x26ac24f8, 0x26ac2600, 0x27560000)
    ro space 8192K,  62% used [0x2a960000, 0x2ae5ba98, 0x2ae5bc00, 0x2b160000)
    rw space 12288K,  52% used [0x2b160000, 0x2b79e410, 0x2b79e600, 0x2bd60000)
```

If the Java VM flag `-XX:+PrintClassHistogram` is set, then the Ctrl-Break handler will produce a heap histogram.

2.16 Operating-System-Specific Tools

This section lists a number of operating-system-specific tools that are useful for troubleshooting or monitoring purposes. A brief description is provided for each tool. For further details, refer to the operating system documentation (or man pages in the case of Solaris OS and Linux).

2.16.1 Solaris Operating System

The following tools are provided by the Solaris Operating System. See also 2.16.4 Tools Introduced in Solaris 10 OS, which gives details for some of the tools that were introduced in version 10 of Solaris OS.

Tool	Description
`coreadm`	Specify name and location of core files produced by the Java VM.
`cpustat`	Monitor system behavior using CPU performance counters.
`cputrack`	Per-process monitor process, LWP behavior using CPU performance counters.
`c++filt`	Demangle C++ mangled symbol names. This utility is delivered with the native c++ compiler suite: `SUNWspro` on Solaris OS.
DTrace tool `dtrace` command	Introduced in Solaris 10 OS: Dynamic tracing of kernel functions, system calls, and user functions. This tool allows arbitrary, safe scripting to be executed at entry, exit, and other probe points. The script is written in C-like but safe pointer semantics language called the **D programming language**. See also 2.16.4.3 Using the DTrace Tool.
`gcore`	Force a core dump of a process. The process continues after the core dump is written.
`intrstat`	Report statistics on CPU consumed by interrupt threads.
`iostat`	Report I/O statistics.
`libumem`	Introduced in Solaris 9 OS update 3: User space slab allocator. This tool can be used to find and fix memory management bugs (see 3.4.5 Using `libumem` to Find Leaks).
`mdb`	Modular debugger for kernel and user applications and crash dumps
`netstat`	Display the contents of various network-related data structures.
`pargs`	Print process arguments, environment variables, or auxiliary vector. Long output is not truncated as it would be by other commands, such as `ps`.
`pfiles`	Print information on process file descriptors. Starting with Solaris 10 OS, the tool prints the filename also.
`pldd`	Print shared objects loaded by a process.
`pmap`	Print memory layout of a process or core file, including heap, data, text sections. Starting with Solaris 10 OS, stack segments are clearly identified with the text `[stack]` along with the thread ID. See also 2.16.4.1 Improvements in `pmap` Tool.
`prstat`	Report statistics for active Solaris OS processes. (Similar to `top`.).
`prun`	Set the process to running mode (reverse of `pstop`).

`ps`	List all processes.
`psig`	List the signal handlers of a process.
`pstack`	Print stack of threads of a given process or core file. Starting with Solaris 10 OS, Java method names can be printed for Java frames. See also 2.16.4.2 Improvements in `pstack` Tool.
`pstop`	Stop the process (suspend).
`ptree`	Print process tree containing the given `pid`.
`sar`	System activity reporter.
`sdtprocess`	Display most CPU-intensive processes. (Similar to `top`.).
`sdtperfmeter`	Display graphs showing system performance, for example, CPU, disks, network, and so forth.
`top`	Display most CPU-intensive processes. This tool is available as freeware for Solaris OS but is not installed by default.
`trapstat`	Display runtime trap statistics. (SPARC only)
`truss`	Trace entry and exit events for system calls, user-mode functions, and signals; optionally stop the process at one of these events. This tool also prints the arguments of system calls and user functions.
`vmstat`	Report system virtual memory statistics.
`watchmalloc`	Track memory allocations.

2.16.2 Linux Operating System

The following tools are provided by the Linux Operating System.

Tool	**Description**
`c++filt`	Demangle C++ mangled symbol names. This utility is delivered with the native `c++` compiler suite: `gnu` on Linux OS.
`gdb`	GNU debugger.
`libnjamd`	Memory allocation tracking.
`lsstack`	Print thread stack (similar to `pstack` in Solaris OS). Not all distributions provide this tool by default; therefore, you might have to download it from the `sourceforge.net`web site.
`ltrace`	Library call tracer (equivalent to `truss -u` in Solaris OS). Not all distributions provide this tool by default; therefore, you might have to download it separately.
`mtrace` and `muntrace`	GNU malloc tracer.
`proc` tools such	Some but not all of the `proc` tools on Solaris OS have equivalent tools on Linux. In

as `pmap` and `pstack`	addition, core file support is not as good as for Solaris OS; for example, `pstack` does not work for core dumps.
`strace`	System call tracer (equivalent to `truss -t` in Solaris OS).
`top`	Display most CPU-intensive processes.
`vmstat`	Report information about processes, memory, paging, block I/O, traps, and CPU activity.

2.16.3 Windows Operating System

The following tools are provided by the Windows Operating System. In addition, you can access the MSDN Library site and search for debug support.

Tool	Description
`dumpchk`	Command-line utility to verify that a memory dump file has been created correctly This tool is included in Debugging Tools for Windows download available from the Microsoft web site (see 7.4.4 Collecting Crash Dumps on Windows).
`msdev` debugger	Command-line utility that can be used to launch Visual C++ and the Win32 debugger.
`userdump`	User Mode Process Dump utility. This tool is included in the OEM Support Tools download available from the Microsoft web site (see section 7.4.4 Collecting Crash Dumps on Windows).
`windbg`	Windows debugger which can be used to debug Windows applications or crash dumps. This tool is included in Debugging Tools for Windows download available from the Microsoft web site (see section 7.4.4 Collecting Crash Dumps on Windows).
`/Md` and `/Mdd` compiler options	Compiler options that automatically include extra support for tracking memory allocations.

2.16.4 Tools Introduced in Solaris 10 OS

This section provides details for some of the diagnostic tools that were introduced in version 10 of the Solaris Operating System.

2.16.4.1 Improvements in `pmap` Tool

The `pmap` utility was improved in Solaris 10 OS to print stack segments with the text `[stack]`. This text helps you to locate the stack easily.

The following example shows some output from this tool.

```
19846:    /net/myserver/export1/user/j2sdk6/bin/java -Djava.endorsed.d
00010000      72K r-x--  /export/disk09/jdk/6/rc/b63/binaries/solsparc/bin/java
00030000      16K rwx--  /export/disk09/jdk/6/rc/b63/binaries/solsparc/bin/java
00034000   32544K rwx--    [ heap ]
D1378000      32K rwx-R    [ stack tid=44 ]
D1478000      32K rwx-R    [ stack tid=43 ]
D1578000      32K rwx-R    [ stack tid=42 ]
D1678000      32K rwx-R    [ stack tid=41 ]
D1778000      32K rwx-R    [ stack tid=40 ]
D1878000      32K rwx-R    [ stack tid=39 ]
D1974000      48K rwx-R    [ stack tid=38 ]
```

```
D1A78000      32K rwx-R    [ stack tid=37 ]
D1B78000      32K rwx-R    [ stack tid=36 ]
[.. more lines removed here to reduce output ..]
FF370000       8K r-x--  /usr/lib/libsched.so.1
FF380000       8K r-x--  /platform/sun4u-us3/lib/libc_psr.so.1
FF390000      16K r-x--  /lib/libthread.so.1
FF3A4000       8K rwx--  /lib/libthread.so.1
FF3B0000       8K r-x--  /lib/libdl.so.1
FF3C0000     168K r-x--  /lib/ld.so.1
FF3F8000       8K rwx--  /lib/ld.so.1
FF3FA000       8K rwx--  /lib/ld.so.1
FFB80000      24K -----    [ anon ]
FFBF0000      64K rwx--    [ stack ]
 total    167224K
```

2.16.4.2 Improvements in `pstack` Tool

Prior to the Solaris 10 OS release, the `pstack` utility did not support the Java language. It printed hexadecimal addresses for both interpreted and (HotSpot) compiled Java methods.

Starting in Solaris 10 OS, the `pstack` command-line tool prints mixed mode stack traces (Java and C/C++ frames) from a core file or a live process. The tool prints Java method names for interpreted, compiled and inlined Java methods.

2.16.4.3 Using the DTrace Tool

Solaris 10 OS includes the DTrace tool, which allows dynamic tracing of the operating system kernel and user-level programs. This tool supports scripting at system-call entry and exit, at user-mode function entry and exit, and at many other probe points. The scripts are written in the **D programming language**, which is a C-like language with safe pointer semantics. These scripts can help you in troubleshooting problems or solving performance issues.

The `dtrace` command is a generic front-end to the DTrace tool. This command provides a simple interface to invoke the D language, to retrieve buffered trace data, and to access a set of basic routines to format and print traced data.

You may write your own customized DTrace scripts, using the D language, or download and use one or more of the many scripts that are already available on various sites.

The probes are delivered and instrumented by kernel modules called providers. The types of tracing offered by the probe providers include user instruction tracing, function boundary tracing, kernel lock instrumentation, profile interrupt, system call tracing, and much more. If you write your own scripts, you use the D language to enable the probes; this language also allows conditional tracing and output formatting.

You can use the `dtrace -l` option to explore the set of providers and probes that are available on your Solaris OS.

The DTrace Toolkit is a collection of useful documented scripts developed by the OpenSolaris DTrace community. Seehttp://www.opensolaris.org/os/community/dtrace/dtracetoolkit/

Details on DTrace are provided at the following locations:

- *Solaris Dynamic Tracing Guide*: http://docs.sun.com/app/docs/doc/817-6223/
- BigAdmin System Administration Portal for DTrace: http://www.sun.com/bigadmin/content/dtrace/

Probe Providers in Java HotSpot VM

Starting with JDK 7, the Java HotSpot VM contains two built-in probe providers: `hotspot` and `hotspot_jni`. These providers deliver probes that can be used to monitor the internal state and activities of the VM, as well as the Java application that is running.

The `hotspot` provider probes can be categorized as follows:

- VM lifecycle: VM initialization begin and end, and VM shutdown.
- Thread lifecycle: thread start and stop, thread name, thread ID, and so on.
- Class-loading: Java class loading and unloading.
- Garbage collection: start and stop of garbage collection, system-wide or by memory pool.
- Method compilation: method compilation begin and end, and method loading and unloading.
- Monitor probes: wait events, notification events, contended monitor entry and exit.

- Application tracking: method entry and return, allocation of a Java object.

In order to call from native code to Java code, the native code must make a call through the JNI interface.
The `hotspot_jni` provider manages DTrace probes at the entry point and return point for each of the methods that the JNI interface provides for invoking Java code and examining the state of the VM.

Example of Using DTrace

At probe points, you can print the stack trace current thread using the `ustack` built-in function. This function prints Java method names in addition to C/C++ native function names. The following is a simple D script that prints a full stack trace whenever a thread calls the read system call.

```
#!/usr/sbin/dtrace -s
syscall::read:entry
/pid == $1 && tid == 1/ {
   ustack(50, 0x2000);
}
```

The above script is stored in a file named `read.d` and is run with the following command:

```
read.d pid-of-the-Java-process-that-is-traced
```

If your Java application generated a lot of I/O or had some unexpected latency, the use of the DTrace tool and its `ustack()` action can help you diagnose the problem.

2.17 Developing Diagnostic Tools

The JDK software has extensive Application Programing Interfaces (APIs) which can be used to develop tools to observe, monitor, profile, debug, and diagnose issues in applications that are deployed on the Java runtime environment. The development of new tools is beyond the scope of this document. Instead this section provides a brief overview of the programming interfaces available. Refer also to example and demonstration code that is included in the JDK download.

2.17.1 `java.lang.management` Package

The `java.lang.management` package provides the management interface for monitoring and management of the Java Virtual Machine and the operating system. Specifically it covers interfaces for the following systems:

- Class loading
- Compilation
- Garbage collection
- Memory manager
- Runtime
- Threads

The `java.lang.management` package is fully described in the Java SE API documentation.

The JDK release includes example code that demonstrates the usage of the `java.lang.management` package. These examples can be found in the`$JAVA_HOME/demo/management` directory. Some of these examples are as follows:

- `MemoryMonitor` - demonstrates the use of the `java.lang.management` API to observe the memory usage of all memory pools consumed by the application.
- `FullThreadDump` - demonstrates the use of the `java.lang.management` API to get a full thread dump and detect deadlocks programmatically.
- `VerboseGC` - demonstrates the use of the `java.lang.management` API to print the garbage collection statistics and memory usage of an application.

In addition to the `java.lang.management` package, the JDK release includes platform extensions in the `com.sun.management` package. The platform extensions include a management interface to obtain detailed statistics from garbage collectors that perform collections in cycles. These extensions also include a management interface to obtain additional memory statistics from the operating system.

Details on the platform extensions can be found at Java SE API documentation: Monitoring and Management Interface for the Java Platform.

2.17.2 `java.lang.instrument` Package

The `java.lang.instrument` package provides services that allow Java programming language agents to instrument programs running on the Java VM. Instrumentation is used by tools such as profilers, tools for tracing method calls, and many others. The package facilitates both load-time and dynamic instrumentation. It also includes methods to obtain information on the loaded classes and information about the amount of storage consumed by a given object.

The `java.lang.instrument` package is fully described in the Java SE API documentation.

2.17.3 `java.lang.Thread` Class

The `java.lang.Thread` class has a static method called `getAllStackTraces`, which returns a map of stack traces for all live threads. The `Thread` class also has a method called `getState`, which returns the thread state; states are defined by the `java.lang.Thread.State` enumeration. These methods can be useful when adding diagnostic or monitoring capabilities to an application. These methods are fully described in the API documentation.

2.17.4 Java Virtual Machine Tools Interface

The Java Virtual Machine Tools Interface (JVM TI) is a native (C/C++) programming interface that can be used to develop a wide range of developing and monitoring tools. JVM TI provides an interface for the full breadth of tools that need access to VM state, including but not limited to profiling, debugging, monitoring, thread analysis, and coverage analysis tools.

Some examples of agents that rely on JVM TI are the following:

- HPROF profiler (see 2.1 HPROF - Heap Profiler)
- Java Debug Wire Protocol (JDWP) agent (see 2.17.5 Java Platform Debugger Architecture)
- `java.lang.instrument` implementation (see 2.17.2 `java.lang.instrument` Package)

The specification for JVM TI can be found in the JVM Tool Interface documentation.

The JDK release includes example code that demonstrates the usage of JVM TI. These examples can be found in the `$JAVA_HOME/demo/jvmti directory`. Some of the examples are as follows:

- `mtrace` - an agent library that tracks method call and return counts. It uses byte-code instrumentation to instrument all classes loaded into the virtual machine and prints a sorted list of the frequently used methods.
- `heapTracker` - an agent library that tracks object allocation. It uses byte-code instrumentation to instrument constructor methods.
- `heapViewer` - an agent library that prints heap statistics when Ctrl-\ or Ctrl-Break is pressed. For each loaded class it prints an instance count of that class and the space used.

2.17.5 Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) is the architecture designed for use by debuggers and debugger-like tools. It consists of two programming interfaces and a wire protocol.

- The Java Virtual Machine Tools Interface (JVM TI) is the interface to the virtual machine (as described in 2.17.4 Java Virtual Machine Tools Interface).
- The Java Debug Interface (JDI) defines information and requests at the user code level. It is a pure Java programming language interface for debugging Java programming language applications. In JPDA, the JDI is a remote view in the debugger process of a virtual machine in the debuggee process. It is implemented by the front-end, while a debugger-like application (for example, IDE, debugger, tracer, monitoring tool, and so forth) is the client.
- The Java Debug Wire Protocol (JDWP) defines the format of information and requests transferred between the process being debugged and the debugger front end, which implements the JDI.

A complete description (including specifications) for JPDA is located in the Java Platform Debugger Architecture (JPDA) documentation.

A graphic view of the JPDA structure is presented in the Java Platform Debugger Architecture description.

The `jdb` utility is included in the JDK release as the example command-line debugger. The `jdb` utility uses the Java Debug Interface (JDI) to launch or connect to the target VM. See 2.4 `jdb` Utility.

In addition to traditional debugger-type tools, JDI can also be used to develop tools that help in post-mortem diagnostics and scenarios where the tool needs to attach to a process in a non-cooperative manner (a hung process, for example). See 2.4 `jdb` Utility for a description of the JDI connectors which can be used to attach a JDI-based tool to a crash dump or hung process.

Troubleshooting Memory Leaks

If your application's execution time becomes longer and longer, or if the operating system seems to be performing slower and slower, this could be an indication of a memory leak. In other words, virtual memory is being allocated but is not being returned when it is no longer needed. Eventually the application or the system runs out of memory, and the application terminates abnormally.

This chapter provides some suggestions on diagnosing problems involving possible memory leaks.

3.1 Meaning of `OutOfMemoryError`

One common indication of a memory leak is the `java.lang.OutOfMemoryError` error. This error is thrown when there is insufficient space to allocate an object in the Java heap or in a particular area of the heap. The garbage collector cannot make any further space available to accommodate a new object, and the heap cannot be expanded further.

When the `java.lang.OutOfMemoryError` error is thrown, a stack trace is printed also.

A `java.lang.OutOfMemoryError` can also be thrown by native library code when a native allocation cannot be satisfied, for example, if swap space is low.

An early step to diagnose an `OutOfMemoryError` is to determine what the error means. Does it mean that the Java heap is full, or does it mean that the native heap is full? To help you answer this question, the following subsections explain some of the possible error messages, with reference to the detail part of the message:

- `Exception in thread "main": java.lang.OutOfMemoryError: Java heap space`

  See 3.1.1 Detail Message: `Java heap space`.

- `Exception in thread "main": java.lang.OutOfMemoryError: PermGen space`

  See 3.1.2 Detail Message: `PermGen space`.

- `Exception in thread "main": java.lang.OutOfMemoryError: Requested array size exceeds VM limit`

  See 3.1.3 Detail Message: `Requested array size exceeds VM limit`.

- `Exception in thread "main": java.lang.OutOfMemoryError: request <size> bytes for <reason>. Out of swap space?`

  See 3.1.4 Detail Message: `request <size> bytes for <reason>. Out of swap space?`.

- `Exception in thread "main": java.lang.OutOfMemoryError: <reason> <stack trace> (Native method)`

  See 3.1.5 Detail Message: `<reason> <stack trace> (Native method)`.

3.1.1 Detail Message: `Java heap space`

The detail message `Java heap space` indicates that an object could not be allocated in the Java heap. This error does not necessarily imply a memory leak. The problem can be as simple as a configuration issue, where the specified heap size (or the default size, if not specified) is insufficient for the application.

In other cases, and in particular for a long-lived application, the message might be an indication that the application is unintentionally holding references to objects, and this prevents the objects from being garbage collected. This is the Java language equivalent of a memory leak. Note that APIs that are called by an application could also be unintentionally holding object references.

One other potential source of `OutOfMemoryError` arises with applications that make excessive use of finalizers. If a class has a finalize method, then objects of that type do not have their space reclaimed at garbage collection time. Instead, after garbage collection the objects are queued for finalization, which occurs at a later time. In the Sun implementation, finalizers are executed by a daemon thread that services the finalization queue. If the finalizer thread cannot keep up with the finalization queue, then the Java heap could fill up and `OutOfMemoryError` would be thrown. One scenario that can cause this situation is when an application creates high-priority threads that cause the finalization queue to increase at a rate that is faster than the rate at which the finalizer thread is servicing that queue. Section 3.3.6 Monitoring the Number of Objects Pending Finalization discusses how to monitor objects for which finalization is pending.

3.1.2 Detail Message: `PermGen space`

The detail message `PermGen space` indicates that the permanent generation is full. The permanent generation is the area of the heap where class and method objects are stored. If an application loads a very large number of classes, then the size of the permanent generation might need to be increased using the `-XX:MaxPermSize` option.

Interned `java.lang.String` objects are no longer stored in the permanent generation. The `java.lang.String` class maintains a pool of strings. When the `intern` method is invoked, the method checks the pool to see if an equal string is already in the pool. If there is, then the `intern` method returns it; otherwise it adds the string to the pool. In more precise terms, the `java.lang.String.intern` method is used to obtain the canonical representation of the string; the result is a reference to the same class instance that would be returned if that string appeared as a literal.

When this kind of error occurs, the text `ClassLoader.defineClass` might appear near the top of the stack trace that is printed.

The `jmap -permgen` command prints statistics for the objects in the permanent generation and also information about internalized String instances. See 2.7.4 Getting Information on the Permanent Generation.

3.1.3 Detail Message: `Requested array size exceeds VM limit`

The detail message `Requested array size exceeds VM limit` indicates that the application (or APIs used by that application) attempted to allocate an array that is larger than the heap size. For example, if an application attempts to allocate an array of 512MB but the maximum heap size is 256MB then `OutOfMemoryError` will be thrown with the reason `Requested array size exceeds VM limit`. In most cases the problem is either a configuration issue (heap size too small), or a bug that results in an application attempting to create a huge array, for example, when the number of elements in the array are computed using an algorithm that computes an incorrect size.

3.1.4 Detail Message: `request <size> bytes for <reason>. Out of swap space?`

The detail message `request <size> bytes for <reason>. Out of swap space?` appears to be an `OutOfMemoryError`. However, the HotSpot VM code reports this apparent exception when an allocation from the native heap failed and the native heap might be close to exhaustion. The message indicates the size (in bytes) of the request that failed and the reason for the memory request. In most cases the `<reason>` part of the message is the name of a source module reporting the allocation failure, although in some cases it indicates a reason.

When this error message is thrown, the VM invokes the fatal error handling mechanism, that is, it generates a fatal error log file, which contains useful information about the thread, process, and system at the time of the crash. In the case of native heap exhaustion, the heap memory and memory map information in the log can be useful. See Appendix C, Fatal Error Log for detailed information about this file.

If this type of `OutOfMemoryError` is thrown, you might need to use troubleshooting utilities on the operating system to diagnose the issue further. See 2.16 Operating-System-Specific Tools.

The problem might not be related to the application, for example:

- The operating system is configured with insufficient swap space.
- Another process on the system is consuming all memory resources.

If neither of the above issues is the cause, then it is possible that the application failed due to a native leak, for example, if application or library code is continuously allocating memory but is not releasing it to the operating system.

3.1.5 Detail Message: `<reason> <stack trace> (Native method)`

If the detail part of the error message is `<reason> <stack trace> (Native method)` and a stack trace is printed in which the top frame is a native method, then this is an indication that a native method has encountered an allocation failure. The difference between this and the previous message is that the allocation failure was detected in a JNI or native method rather than in Java VM code.

If this type of `OutOfMemoryError` is thrown, you might need to use utilities on the operating system to further diagnose the issue. See 2.16 Operating-System-Specific Tools.

3.2 Crash Instead of `OutOfMemoryError`

Sometimes an application crashes soon after an allocation from the native heap fails. This occurs with native code that does not check for errors returned by memory allocation functions.

For example, the `malloc` system call returns NULL if there is no memory available. If the return from `malloc` is not checked, then the application might crash when it attempts to access an invalid memory location. Depending on the circumstances, this type of issue can be difficult to locate.

However, in some cases the information from the fatal error log or the crash dump might be sufficient to diagnose this issue. The fatal error log is covered in detail in Appendix C, Fatal Error Log. If the cause of a crash is determined to be the failure to check an allocation failure, then the reason for the allocation failure must be examined. As with any other native heap issue, the system might be configured with insufficient swap space, another process on the system might be consuming all memory resources, or there might be a leak in the application (or in the APIs that it calls) that causes the system to run out of memory.

3.3 Diagnosing Leaks in Java Language Code

Diagnosing leaks in Java language code can be a difficult task. In most cases it requires very detailed knowledge of the application. In addition the process is often iterative and lengthy. This section provides the following subsections:

3.3.1 NetBeans Profiler

The NetBeans Profiler (previously known as `JFluid`) is an excellent profiler, which can locate memory leaks very quickly. Most commercial memory leak debugging tools can often take a long time to locate a leak in a large application. The NetBeans Profiler, however, uses the pattern of memory allocations and reclamations that such objects typically demonstrate. This process includes also the lack of memory reclamations. The profiler can check where these objects were allocated, which in many cases is sufficient to identify the root cause of the leak.

More details can be found at http://profiler.netbeans.org.

3.3.2 Using the `jhat` Utility

The `jhat` utility (see 2.5 `jhat` Utility) is useful when debugging unintentional object retention (or memory leaks). It provides a way to browse an object dump, view all reachable objects in the heap, and understand which references are keeping an object alive.

To use `jhat` you must obtain one or more heap dumps of the running application, and the dumps must be in binary format. Once the dump file is created, it can be used as input to `jhat`, as described in 2.5 `jhat` Utility.

3.3.3 Creating a Heap Dump

A heap dump provides detailed information on the allocation of heap memory. The following sections describe several ways to produce a heap dump:

3.3.3.1 HPROF Profiler

The HPROF profiler agent can create a heap dump while the application is executing. The following is an example of the command line:

```
$ java -agentlib:hprof=file=snapshot.hprof,format=b application
```

If the VM is embedded or is not started using a command line launcher that allows additional options to be provided, then it might be possible to use the `JAVA_TOOLS_OPTIONS` environment variable so that the `-agentlib` option is automatically added to the command line. See A.2 `JAVA_TOOL_OPTIONS` Environment Variable for further information on this environment variable.

Once the application is running with HPROF, a heap dump is created by pressing Ctrl-\ or Ctrl-Break (depending on the platform) on the application console. An alternative approach on Solaris OS and Linux is to send a QUIT signal with the `kill -QUIT` *pid* command. When the signal is received, a heap dump is created; in the above example the file `snapshot.hprof` is created.

The heap dump file contains all the primitive data and stack traces.

A dump file can contain multiple heap dumps. If Ctrl-\ or Ctrl-Break is pressed a number of times then the subsequent dumps are appended to the file. The `jhat` utility uses the #*n* syntax to distinguish the dumps, where *n* is the dump number.

3.3.3.2 `jmap` Utility

A heap dump can also be obtained using the `jmap` utility (see 2.7 `jmap` Utility). The following is an example of the command line:

```
$ jmap -dump:format=b,file=snapshot.jmap process-pid
```

Regardless of how the Java VM was started, the `jmap` tool will produce a head dump snapshot, in the above example in a file called `snapshot.jmap`. The `jmap` output files should contain all the primitive data, but will not include any stack traces showing where the objects have been created.

3.3.3.3 JConsole Utility

Another way to obtain a heap dump is with the JConsole utility. In the MBeans tab, select the `HotSpotDiagnostic` MBean, then the Operations display, and choose the `dumpHeap` operation.

3.3.3.4 `-XX:+HeapDumpOnOutOfMemoryError` Command-line Option

If you specify the `-XX:+HeapDumpOnOutOfMemoryError` command-line option, and if an `OutOfMemoryError` is thrown, the VM generates a heap dump.

3.3.4 Obtaining a Heap Histogram on a Running Process

You can try to quickly narrow down a memory leak by examining a heap histogram. This information can be obtained in several ways:

- A heap histogram can be obtained from a running process using the command `jmap -histo` *pid*. The output shows the total size and instance count for each class type in the heap. If a sequence of histograms is obtained (for example, every 2 minutes), then you might be able to observe a trend that can lead to further analysis.
- On Solaris OS and Linux, the `jmap` utility can also provide a histogram from a core file.
- If the Java process is started with the `-XX:+PrintClassHistogram` command-line option, then the Ctrl-Break handler will produce a heap histogram.

3.3.5 Obtaining a Heap Histogram at OutOfMemoryError

If you specify the `-XX:+HeapDumpOnOutOfMemoryError` command-line option, and if an `OutOfMemoryError` is thrown, the VM generates a heap dump. You can then use the `jmap` utility to obtain a histogram from the heap dump.

If a core file is produced when the `OutOfMemoryError` is thrown, you can execute `jmap` on the core file to get a histogram, as in the following example.

```
$ jmap -histo \ /java/re/javase/6/latest/binaries/solaris-sparc/bin/java core.27421

Attaching to core core.27421 from executable
/java/re/javase/6/latest/binaries/solaris-sparc/bin/java, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 1.6.0-beta-b63
Iterating over heap. This may take a while...
Heap traversal took 8.902 seconds.

Object Histogram:

Size      Count   Class description
-------------------------------------------------------
86683872  3611828 java.lang.String
20979136  204     java.lang.Object[]
403728    4225    * ConstMethodKlass
306608    4225    * MethodKlass
220032    6094    * SymbolKlass
152960    294     * ConstantPoolKlass
108512    277     * ConstantPoolCacheKlass
104928    294     * InstanceKlassKlass
68024     362     byte[]
65600     559     char[]
31592     359     java.lang.Class
27176     462     java.lang.Object[]
25384     423     short[]
17192     307     int[]
:
```

The example shows that the `OutOfMemoryError` is caused by the number of `java.lang.String` objects (3611828 instances in the heap). Without further analysis it is not clear where the strings are allocated. However, the information is still useful and the investigation can continue with tools such as `HPROF` or `jhat` to find out where the strings are allocated, as well as what references are keeping them alive and preventing them from being garbage collected.

3.3.6 Monitoring the Number of Objects Pending Finalization

As noted in 3.1.1 Detail Message: `Java heap space`, excessive use of finalizers can be the cause of `OutOfMemoryError`. You have several options for monitoring the number of objects that are pending finalization.

- The JConsole management tool (see 2.3 JConsole Utility) can be used to monitor the number of objects that are pending finalization. This tool reports the pending finalization count in the memory statistics on the “Summary” tab pane. The count is approximate but it can be used to characterize an application and understand if it relies a lot on finalization.
- On Solaris OS and Linux, the `jmap -finalizerinfo` option prints information on objects awaiting finalization.
- An application can report the approximate number of objects pending finalization using the `getObjectPendingFinalizationCount` method in the`java.lang.management.MemoryMXBean` class. Links to the API documentation and example code can be found in 2.17 Developing Diagnostic Tools. The example code can easily be extended to include the reporting of the pending finalization count.

3.3.7 Third Party Memory Debuggers

In addition to the tools mentioned in the previous chapters, there are a large number of third-party memory debuggers available. JProbe from Quest Software, and OptimizeIt from Borland are two examples of commercial tools with memory debugging capability. There are many others and no specific product is recommended.

3.4 Diagnosing Leaks in Native Code

Several techniques can be used to find and isolate native code memory leaks. In general there is no single ideal solution for all platforms.

3.4.1 Tracking All Memory Allocation and Free Calls

A very common practice is to track all allocation and free calls of the native allocations. This can be a fairly simple process or a very sophisticated one. Many products over the years have been built up around the tracking of native heap allocations and the use of that memory.

Tools like Purify and Sun's `dbx` Run Time Checking (see 3.4.4 Using `dbx` to Find Leaks) functionality can be used to find these leaks in normal native code situations and also find any access to native heap memory that represents assignments to uninitialized memory or accesses to freed memory.

Not all these types of tools will work with Java applications that use native code, and usually these tools are platform-specific. Since the virtual machine dynamically creates code at runtime, these tools can wrongly interpret the code and fail to run at all, or give false information. Check with your tool vendor to make sure the version of the tool works with the version of the virtual machine you are using.

Many simple and portable native memory leak detecting examples can be found at http://sourceforge.net/. Most of these libraries and tools assume that you can recompile or edit the source of the application and place wrapper functions over the allocation functions. The more powerful of these tools allow you to run your application unchanged by interposing over these allocation functions dynamically. This is the case with the library `libumem.so`, starting with Solaris 9 OS update 3; see 3.4.5 Using `libumem` to Find Leaks.

3.4.2 Tracking Memory Allocation in a JNI Library

If you write a JNI library, it would probably be wise to create some kind of localized way to make sure your library does not leak memory, using a simple wrapper approach.

The following procedure is an easy localized allocation tracking approach for a JNI library. First, define the following lines in all source files:

```
#include <stdlib.h>
#define malloc(n) debug_malloc(n, __FILE__, __LINE__)
#define free(p) debug_free(p, __FILE__, __LINE__)
```

Then you can use the following functions to watch for leaks.

```
/* Total bytes allocated */
static int total_allocated;
/* Memory alignment is important */
typedef union { double d; struct {size_t n; char *file; int line;} s; } Site;
void *
debug_malloc(size_t n, char *file, int line)
{
    char *rp;
    rp = (char*)malloc(sizeof(Site)+n);
    total_allocated += n;
    ((Site*)rp)->s.n = n;
    ((Site*)rp)->s.file = file;
    ((Site*)rp)->s.line = line;
    return (void*)(rp + sizeof(Site));
}
void
debug_free(void *p, char *file, int line)
{
    char *rp;
    rp = ((char*)p) - sizeof(Site);
    total_allocated -= ((Site*)rp)->s.n;
    free(rp);
}
```

The JNI library would then need to periodically (or at shutdown) check the value of the `total_allocated` variable to make sure that it made sense. The above code could also be expanded to save in a linked list the allocations that remained and report where

the leaked memory was allocated. This is a localized and portable way to track memory allocations in a single set of sources. You would need to make sure that `debug_free()` was called only with a pointer that came from `debug_malloc()`, and you would also need to create similar functions for `realloc()`, `calloc()`, `strdup()`, and so forth, if they were used.

A more global way to look for native heap memory leaks would involve interposition of the library calls for the entire process.

3.4.3 Tracking Memory Allocation With OS Support

Most operating systems include some form of global allocation tracking support.

- On Windows, go to http://msdn.microsoft.com/library/default.asp and search for debug support. The Microsoft C++ compiler has the `/Md` and `/Mdd` compiler options that will automatically include extra support for tracking memory allocations.
- Linux systems have tools such as `mtrace` and `libnjamd` to help in dealing with allocation tracking.
- Solaris Operating Systems provide the `watchmalloc` tool. Solaris 9 OS update 3 started providing the `libumem` tool (see 3.4.5 Using `libumem` to Find Leaks).

3.4.4 Using *dbx* to Find Leaks

The Sun debugger `dbx` includes the Run Time Checking (RTC) functionality, which can find leaks. The `dbx` debugger is also available on Linux.

Below is a sample `dbx` session.

```
$ dbx ${java_home}/bin/java
Reading java
Reading ld.so.1
Reading libthread.so.1
Reading libdl.so.1
Reading libc.so.1
(dbx) dbxenv rtc_inherit on
(dbx) check -leaks
leaks checking - ON
(dbx) run HelloWorld
Running: java HelloWorld
(process id 15426)
Reading rtcapihook.so
Reading rtcaudit.so
Reading libmapmalloc.so.1
Reading libgen.so.1
Reading libm.so.2
Reading rtcboot.so
Reading librtc.so
RTC: Enabling Error Checking...
RTC: Running program...
dbx: process 15426 about to exec("/net/bonsai.sfbay/export/home2/user/ws/j2se/build/solaris-
i586/bin/java")
dbx: program "/net/bonsai.sfbay/export/home2/user/ws/j2se/build/solaris-i586/bin/java"
just exec'ed
dbx: to go back to the original program use "debug $oprog"
RTC: Enabling Error Checking...
RTC: Running program...
t@1 (l@1) stopped in main at 0x0805136d
0x0805136d: main       :        pushl    %ebp
(dbx) when dlopen libjvm { suppress all in libjvm.so; }
(2) when dlopen libjvm { suppress all in libjvm.so; }
(dbx) when dlopen libjava { suppress all in libjava.so; }
(3) when dlopen libjava { suppress all in libjava.so; }
(dbx) cont
Reading libjvm.so
Reading libsocket.so.1
Reading libsched.so.1
Reading libCrun.so.1
Reading libm.so.1
Reading libnsl.so.1
Reading libmd5.so.1
Reading libmp.so.2
Reading libhpi.so
```

```
Reading libverify.so
Reading libjava.so
Reading libzip.so
Reading en_US.ISO8859-1.so.3
hello world
hello world
Checking for memory leaks...

Actual leaks report    (actual leaks:           27  total size:      46851 bytes)

  Total     Num of  Leaked     Allocation call stack
  Size      Blocks  Block
                    Address
==========  ====== =========== =======================================
     44376       4      -      calloc < zcalloc
      1072       1  0x8151c70  _nss_XbyY_buf_alloc < get_pwbuf < _getpwuid <
                               GetJavaProperties < Java_java_lang_System_initProperties <
                               0xa740a89a< 0xa7402a14< 0xa74001fc
       814       1  0x8072518  MemAlloc < CreateExecutionEnvironment < main
       280      10      -      operator new < Thread::Thread
       102       1  0x8072498  _strdup < CreateExecutionEnvironment < main
        56       1  0x81697f0  calloc < Java_java_util_zip_Inflater_init < 0xa740a89a<
                               0xa7402a6a< 0xa7402aeb< 0xa7402a14< 0xa7402a14< 0xa7402a14
        41       1  0x8072bd8  main
        30       1  0x8072c58  SetJavaCommandLineProp < main
        16       1  0x806f180  _setlocale < GetJavaProperties <
                               Java_java_lang_System_initProperties < 0xa740a89a< 0xa7402a14<
                               0xa74001fc< JavaCalls::call_helper < os::os_exception_wrapper
        12       1  0x806f2e8  operator new < instanceKlass::add_dependent_nmethod <
                               nmethod::new_nmethod < ciEnv::register_method <
                               Compile::Compile #Nvariant 1 < C2Compiler::compile_method <
                               CompileBroker::invoke_compiler_on_method <
                               CompileBroker::compiler_thread_loop
        12       1  0x806ee60  CheckJvmType < CreateExecutionEnvironment < main
        12       1  0x806ede8  MemAlloc < CreateExecutionEnvironment < main
        12       1  0x806edc0  main
         8       1  0x8071cb8  _strdup < ReadKnownVMs < CreateExecutionEnvironment < main
         8       1  0x8071cf8  _strdup < ReadKnownVMs < CreateExecutionEnvironment < main
```

The output shows that the `dbx` debugger reports memory leaks if memory is not freed at the time the process is about to exit. However, memory that is allocated at initialization time and needed for the life of the process is often never freed in native code. Therefore, in such cases the `dbx` debugger can report memory leaks that are not leaks in reality.

Note that the example used two `suppress` commands to suppress the leaks reported in the virtual machine (`libjvm.so`) and the Java support library (`libjava.so`).

3.4.5 Using *libumem* to Find Leaks

Starting with Solaris 9 OS update 3, the `libumem.so` library and the modular debugger (`mdb`) can be used to debug memory leaks. Before using `libumem`, you must preload the `libumem` library and set an environment variable as follows:

```
$ LD_PRELOAD=libumem.so
$ export LD_PRELOAD

$ UMEM_DEBUG=default
$ export UMEM_DEBUG
```

Now, run the Java application but stop it before it exits. The following example uses `truss` to stop the process when it calls the `_exit` system call:

```
$ truss -f -T _exit java MainClass arguments
```

At this point you can attach the `mdb` debugger, as follows:

```
$ mdb -p pid
>::findleaks
```

The `::findleaks` command is the `mdb` command to find memory leaks. If a leak is found, the `findleaks` command prints the address of the allocation call, buffer address, and nearest symbol.

It is also possible to get the stack trace for the allocation which resulted in the memory leak by dumping the `bufctl` structure. The address of this structure can be obtained from the output of the `::findleaks` command. The description of the commands to perform these functions, as well as more information on using `libumem` to identify memory managements bugs, is located at the following address: http://download.oracle.com/docs/cd/E19424-01/820-4814/geogv/index.html.

Chapter 4

Troubleshooting System Crashes

This chapter provides information and guidance on some specific procedures for troubleshooting system crashes.

A crash, or fatal error, causes a process to terminate abnormally. There are various possible reasons for a crash. For example, a crash can occur due to a bug in the HotSpot VM, in a system library, in a Java SE library or API, in application native code, or even in the operating system. External factors, such as resource exhaustion in the operating system can also cause a crash.

Crashes caused by bugs in the HotSpot VM or in the Java SE library code are rare. This chapter provides suggestions on how to examine a crash. In some cases it is possible work around a crash until the cause of the bug is diagnosed and fixed.

In general the first step with any crash is to locate the fatal error log. This is a text file that the HotSpot VM generates in the event of a crash. See Appendix C, Fatal Error Logfor an explanation of how to locate this file, as well as a detailed description of the file.

4.1 Sample Crashes

This section presents a number of examples which demonstrate how the error log can be used to suggest the cause of a crash.

4.1.1 Determining Where the Crash Occurred

The error log header indicates the problematic frame. See C.3 Header Format.

If the top frame type is a native frame and not one of the operating system native frames, then this indicates that the problem is likely in that native library and not in the Java virtual machine. The first step to solving this crash is to investigate the source of the native library where the crash occurred. There are three options, depending on the source of the native library.

1. If the native library is provided by your application, then investigate the source code of your native library. The option `-Xcheck:jni` can help find many native bugs. See B.2.1 `-Xcheck:jni` Option.
2. If the native library has been provided by another vendor and is used by your application, then file a bug report against this third-party application and provide the fatal error log information.
3. Determine if the native library is part of the Java runtime environment (JRE) by looking in the `jre/lib` or `jre/bin` directories in the JRE distribution. If so, file a bug report, and ensure that this library name is prominently indicated so that the bug report can be routed to the appropriate developers.

If the top frame indicated in the error log is another type of frame, file a bug report and include the fatal error log as well as any information on how to reproduce the problem.

See also the remaining sections in this chapter.

4.1.2 Crash in Native Code

If the fatal error log indicates that the crash was in a native library, there might be a bug in native code or JNI library code. The crash could of course be caused by something else, but analysis of the library and any core file or crash dump is a good starting place. For example, consider the following extract from the header of a fatal error log:

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
#  SIGSEGV (0xb) at pc=0x417789d7, pid=21139, tid=1024
#
# Java VM: Java HotSpot(TM) Server VM (6-beta2-b63 mixed mode)
# Problematic frame:
# C  [libApplication.so+0x9d7]
```

In this case a `SIGSEGV` occurred with a thread executing in the library `libApplication.so`.

In some cases a bug in a native library manifests itself as a crash in Java VM code. Consider the following crash where a `JavaThread` fails while in the `_thread_in_vm` state (meaning that it is executing in Java VM code) :

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
```

```
#  EXCEPTION_ACCESS_VIOLATION (0xc0000005) at pc=0x08083d77, pid=3700, tid=2896
#
# Java VM: Java HotSpot(TM) Client VM (1.5-internal mixed mode)
# Problematic frame:
# V  [jvm.dll+0x83d77]

---------------  T H R E A D  ---------------

Current thread (0x00036960):  JavaThread "main" [_thread_in_vm, id=2896]
 :
Stack: [0x00040000,0x00080000),  sp=0x0007f9f8,  free space=254k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [jvm.dll+0x83d77]
C  [App.dll+0x1047]          <========= C/native frame
j  Test.foo()V+0
j  Test.main([Ljava/lang/String;)V+0
v  ~StubRoutines::call_stub
V  [jvm.dll+0x80f13]
V  [jvm.dll+0xd3842]
V  [jvm.dll+0x80de4]
V  [jvm.dll+0x87cd2]
C  [java.exe+0x14c0]
C  [java.exe+0x64cd]
C  [kernel32.dll+0x214c7]
 :
```

In this case the stack trace shows that a native routine in `App.dll` has called into the VM (probably with JNI).

If you get a crash in a native application library (as in the above examples), then you might be able to attach the native debugger to the core file or crash dump, if it is available. Depending on the operating system, the native debugger is `dbx`, `gdb`, or `windbg`.

Another approach is to run with the `-Xcheck:jni` option added to the command line (see B.2.1 `-Xcheck:jni` Option). This option is not guaranteed to find all issues with JNI code, but it can help identify a significant number of issues.

If the native library where the crash occurred is part of the Java runtime environment (for example `awt.dll`, `net.dll`, and so forth), then it is possible that you have encountered a library or API bug. If after further analysis you conclude this is a library or API bug, then gather a much data as possible and submit a bug or support call. See Chapter 7, Submitting Bug Reports.

4.1.3 Crash due to Stack Overflow

A stack overflow in Java language code will normally result in the offending thread throwing `java.lang.StackOverflowError`. On the other hand, C and C++ write past the end of the stack and provoke a stack overflow. This is a fatal error which causes the process to terminate.

In the HotSpot implementation, Java methods share stack frames with C/C++ native code, namely user native code and the virtual machine itself. Java methods generate code that checks that stack space is available a fixed distance towards the end of the stack so that the native code can be called without exceeding the stack space. This distance towards the end of the stack is called “Shadow Pages.” The size of the shadow pages is between 3 and 20 pages, depending on the platform. This distance is tunable, so that applications with native code needing more than the default distance can increase the shadow page size. The option to increase shadow pages is `-XX:StackShadowPages=`*n*, where *n* is greater than the default stack shadow pages for the platform.

If your application gets a segmentation fault without a core file or fatal error log file (see Appendix C, Fatal Error Log) or a `STACK_OVERFLOW_ERROR` on Windows or the message “An irrecoverable stack overflow has occurred,” this indicates that the value of `StackShadowPages` was exceeded and more space is needed.

If you increase the value of `StackShadowPages`, you might also need to increase the default thread stack size using the `-Xss`parameter. Increasing the default thread stack size might decrease the number of threads that can be created, so be careful in choosing a value for the thread stack size. The thread stack size varies by platform from 256k to 1024k.

The following is a fragment from a fatal error log, on a Windows system, where a thread has provoked a stack overflow in native code.

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
```

```
#  EXCEPTION_STACK_OVERFLOW (0xc00000fd) at pc=0x10001011, pid=296, tid=2940
#
# Java VM: Java HotSpot(TM) Client VM (1.6-internal mixed mode, sharing)
# Problematic frame:
# C  [App.dll+0x1011]
#

---------------  T H R E A D  ---------------

Current thread (0x000367c0):  JavaThread "main" [_thread_in_native, id=2940]
:
Stack: [0x00040000,0x00080000),  sp=0x00041000,  free space=4k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
C  [App.dll+0x1011]
C  [App.dll+0x1020]
C  [App.dll+0x1020]
:
C  [App.dll+0x1020]
C  [App.dll+0x1020]
...<more frames>...

Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
j  Test.foo()V+0
j  Test.main([Ljava/lang/String;)V+0
v  ~StubRoutines::call_stub
```

Note the following information in the above output:

- The exception is `EXCEPTION_STACK_OVERFLOW`.
- The thread state is `_thread_in_native`, which means that the thread is executing native or JNI code.
- In the stack information the free space is only 4k (a single page on a Windows system). In addition, the stack pointer (`sp`) is at 0x00041000, which is close to the end of the stack (0x00040000).
- The printout of the native frames shows that a recursive native function is the issue in this case.
- The output notation `...<more frames>...` indicates that additional frames exist but were not printed. The output is limited to 100 frames.

4.1.4 Crash in the HotSpot Compiler Thread

If the fatal error log output shows that the `Current thread` is a `JavaThread` named `CompilerThread0`, `CompilerThread1`, or `AdapterCompiler`, then it is possible that you have encountered a compiler bug. In this case it might be necessary to temporarily work around the issue by switching the compiler (for example, by using the HotSpot Client VM instead of the HotSpot Server VM, or visa versa), or by excluding from compilation the method that provoked the crash. This is discussed in 4.2.1 Crash in HotSpot Compiler Thread or Compiled Code.

4.1.5 Crash in Compiled Code

If the crash occurred in compiled code, then it is possible that you have encountered a compiler bug that has resulted in incorrect code generation. You can recognize a crash in compiled code if the problematic frame is marked with the code `J` (meaning a compiled Java frame). Below is an example of a such a crash:

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
#  SIGSEGV (0xb) at pc=0x0000002a99eb0c10, pid=6106, tid=278546
#
# Java VM: Java HotSpot(TM) 64-Bit Server VM (1.6.0-beta-b51 mixed mode)
# Problematic frame:
# J  org.foobar.Scanner.body()V
#
:
Stack: [0x0000002aea560000,0x0000002aea660000),  sp=0x0000002aea65ddf0,
  free space=1015k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
J  org.foobar.Scanner.body()V

[error occurred during error reporting, step 120, id 0xb]
```

Note that a complete thread stack is not available. The output line "`error occurred during error reporting`" means that a problem arose trying to obtain the stack trace (perhaps stack corruption in this example).

It might be possible to temporarily work around the issue by switching the compiler (for example, by using the HotSpot Client VM instead of the HotSpot Server VM, or visa versa) or by excluding from compilation the method that provoked the crash. In this specific example it might not be possible to switch the compiler as it was taken from the 64-bit Server VM and hence it might not be feasible to switch to the 32-bit Client VM.

4.1.6 Crash in `VMThread`

If the fatal log output shows that the `Current thread` is the `VMThread`, then look for the line containing `VM_Operation` in the `THREAD` section. The `VMThread` is a special thread in the HotSpot VM. It performs special tasks in the VM such as garbage collection (GC). If the `VM_Operation` suggests that the operation is a garbage collection, then it is possible that you have encountered an issue such as heap corruption.

The crash might also be a GC issue, but it could equally be something else (such as a compiler or runtime bug) that leaves object references in the heap in an inconsistent or incorrect state. In this case, collect as much information as possible about the environment and try possible workarounds. If the issue is GC-related you might be able to temporarily work around the issue by changing the GC configuration. This is discussed in 4.2.2 Crash During Garbage Collection.

4.2 Finding a Workaround

If a crash occurs with a critical application, and the crash appears to be caused by a bug in the HotSpot VM, then it might be desirable to quickly find a temporary workaround. The purpose of this section is to suggest some possible workarounds. If the crash occurs with an application that is deployed with the most recent release of the JDK, then the crash should always be reported to Oracle.

NOTE - EVEN IF A WORKAROUND IN THIS SECTION SUCCESSFULLY ELIMINATES A CRASH, THE WORKAROUND IS NOT A FIX FOR THE PROBLEM, BUT MERELY A TEMPORARY SOLUTION. SUBMIT A SUPPORT CALL OR BUG REPORT WITH THE ORIGINAL CONFIGURATION THAT DEMONSTRATED THE ISSUE.

4.2.1 Crash in HotSpot Compiler Thread or Compiled Code

If the fatal error log indicates that the crash occurred in a compiler thread, then it is possible (but not always the case) that you have encountered a compiler bug. Similarly, if the crash is in compiled code then it is possible that the compiler has generated incorrect code.

In the case of the HotSpot Client VM (`-client` option), the compiler thread appears in the error log as `CompilerThread0`. With the HotSpot Server VM there are multiple compiler threads and these appear in the error log file as `CompilerThread0`, `CompilerThread1`, and `AdapterThread`.

Below is a fragment of an error log for a compiler bug that was encountered and fixed during the development of J2SE 5.0. The log file shows that the HotSpot Server VM is used and the crash occurred in `CompilerThread1`. In addition, the log file shows that the `Current CompileTask` was the compilation of the`java.lang.Thread.setPriority` method.

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
:
# Java VM: Java HotSpot(TM) Server VM (1.5-internal-debug mixed mode)
:
---------------  T H R E A D  ---------------

Current thread (0x001e9350): JavaThread "CompilerThread1" daemon [_thread_in_vm, id=20]

Stack: [0xb2500000,0xb2580000),  sp=0xb257e500,  free space=505k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [libjvm.so+0xc3b13c]
:

Current CompileTask:
opto: 11      java.lang.Thread.setPriority(I)V (53 bytes)
```

```
--------------  P R O C E S S  ---------------

Java Threads: ( => current thread )
  0x00229930 JavaThread "Low Memory Detector" daemon [_thread_blocked, id=21]
=>0x001e9350 JavaThread "CompilerThread1" daemon [_thread_in_vm, id=20]
 :
```

In this case there are two potential workarounds:

- The brute force approach: change the configuration so that the application is run with the `-client` option to specify the HotSpot Client VM.
- Assume that the bug only occurs during the compilation of the `setPriority` method and exclude this method from compilation.

The first approach (to use the `-client` option) might be trivial to configure in some environments. In others, it might be more difficult if the configuration is complex or if the command line to configure the VM is not readily accessible. In general, switching from the HotSpot Server VM to the HotSpot Client VM also reduces the peak performance of an application. Depending on the environment, this might be acceptable until the actual issue is diagnosed and fixed.

The second approach (exclude the method from compilation) requires creating the file `.hotspot_compiler` in the working directory of the application. Below is an example of this file:

```
exclude    java/lang/Thread    setPriority
```

In general the format of this file is `exclude CLASS METHOD`, where `CLASS` is the class (fully qualified with the package name) and `METHOD` is the name of the method. Constructor methods are specified as `<init>` and static initializers are specified as `<clinit>`.

NOTE - THE `.HOTSPOT_COMPILER` FILE IS AN UNSUPPORTED INTERFACE. IT IS DOCUMENTED HERE SOLELY FOR THE PURPOSES OF TROUBLESHOOTING AND FINDING A TEMPORARY WORKAROUND.

Once the application is restarted, the compiler will not attempt to compile any of the methods listed as excluded in the `.hotspot_compiler` file. In some cases this can provide temporary relief until the root cause of the crash is diagnosed and the bug is fixed.

In order to verify that the HotSpot VM correctly located and processed the `.hotspot_compiler` file that is shown in the example above, look for the following log information at runtime. Note that the file name separator is a dot, not a slash.

```
### Excluding compile:    java.lang.Thread::setPriority
```

4.2.2 Crash During Garbage Collection

If a crash occurs during garbage collection (GC), then the fatal error log reports that a `VM_Operation` is in progress. For the purposes of this discussion, assume that the mostly concurrent GC (`-XX:+UseConcMarkSweep`) is not in use.
The `VM_Operation` is shown in the `THREAD` section of the log and indicates one of the following situations:

- Generation collection for allocation
- Full generation collection
- Parallel gc failed allocation
- Parallel gc failed permanent allocation
- Parallel gc system gc

Most likely the current thread reported in the log is the `VMThread`. This is the special thread used to execute special tasks in the HotSpot VM. The following fragment of the fatal error log shows an example of a crash in the serial garbage collector:

```
---------------  T H R E A D  ---------------

Current thread (0x002cb720):  VMThread [id=3252]

siginfo: ExceptionCode=0xc0000005, reading address 0x00000000

Registers:
EAX=0x0000000a, EBX=0x00000001, ECX=0x00289530, EDX=0x00000000
ESP=0x02aefc2c, EBP=0x02aefc44, ESI=0x00289530, EDI=0x00289530
EIP=0x0806d17a, EFLAGS=0x00010246

Top of Stack: (sp=0x02aefc2c)
0x02aefc2c:   00289530 081641e8 00000001 0806e4b8
0x02aefc3c:   00000001 00000000 02aefc9c 0806e4c5
0x02aefc4c:   081641e8 081641c8 00000001 00289530
0x02aefc5c:   00000000 00000000 00000001 00000001
0x02aefc6c:   00000000 00000000 00000000 08072a9e
0x02aefc7c:   00000000 00000000 00000000 00035378
0x02aefc8c:   00035378 00280d88 00280d88 147fee00
0x02aefc9c:   02aefce8 0806e0f5 00000001 00289530
Instructions: (pc=0x0806d17a)
0x0806d16a:   15 08 83 3d c0 be 15 08 05 53 56 57 8b f1 75 0f
0x0806d17a:   0f be 05 00 00 00 00 83 c0 05 a3 c0 be 15 08 8b

Stack: [0x02ab0000,0x02af0000),  sp=0x02aefc2c,  free space=255k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [jvm.dll+0x6d17a]
V  [jvm.dll+0x6e4c5]
V  [jvm.dll+0x6e0f5]
V  [jvm.dll+0x71771]
V  [jvm.dll+0xfd1d3]
V  [jvm.dll+0x6cd99]
V  [jvm.dll+0x504bf]
V  [jvm.dll+0x6cf4b]
V  [jvm.dll+0x1175d5]
V  [jvm.dll+0x1170a0]
V  [jvm.dll+0x11728f]
V  [jvm.dll+0x116fd5]
C  [MSVCRT.dll+0x27fb8]
C  [kernel32.dll+0x1d33b]

VM_Operation (0x0373f71c): generation collection for allocation, mode:
 safepoint, requested by thread 0x02db7108
```

NOTE - A CRASH DURING GARBAGE COLLECTION DOES NOT IMPLY A BUG IN THE GARBAGE COLLECTION IMPLEMENTATION. IT COULD ALSO INDICATE A COMPILER OR RUNTIME BUG OR SOME OTHER ISSUE.

You can try the following workarounds if you get a repeated crash during garbage collection:

- Switch GC configuration. For example, if you are using the serial collector, try the throughput collector, or visa versa.
- If you are using the HotSpot Server VM, try the HotSpot Client VM.

If you are not sure which garbage collector is in use, you can use the `jmap` utility on Solaris OS and Linux (see 2.7 `jmap` Utility) to obtain the heap information from the core file, if the core file is available. In general if the GC configuration is not specified on the command line, then the serial collector will be used on Windows. On Solaris OS and Linux it depends on the machine configuration. If the machine has at least 2GB of memory and has at least 2 processors, then the throughput collector (Parallel GC) will be used. For smaller machines the serial collector is the default. The option to select the serial collector is `-XX:+UseSerialGC` and the option to select the throughput collector is `-XX:+UseParallelGC`. If, as a workaround, you switch from the throughput collector to the serial collector, then you might experience some performance degradation on multi-processor systems. This might be acceptable until the root issue is diagnosed and resolved.

4.2.3 Class Data Sharing

Class data sharing was a new feature in J2SE 5.0. When the JRE is installed on 32-bit platforms using the Sun-provided installer, the installer loads a set of classes from the system JAR file into a private internal representation and dumps that representation to a file called a shared archive. When the VM is started, the shared archive is memory-mapped in. This saves on class loading and allows much of the metadata associated with the classes to be shared across multiple VM instances. In J2SE 5.0, class data sharing is enabled only when the HotSpot Client VM is used. In addition, sharing is supported only with the serial garbage collector.

The fatal error log prints the version string in the header of the log. If sharing is enabled, it is indicated by the text `sharing`, as shown in the following example:

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
#  EXCEPTION_ACCESS_VIOLATION (0xc0000005) at pc=0x08083d77, pid=3572, tid=784
#
# Java VM: Java HotSpot(TM) Client VM (1.5-internal mixed mode, sharing)
# Problematic frame:
# V  [jvm.dll+0x83d77]
```

Sharing can be disabled by providing the `-Xshare:off` option on the command line. If the crash cannot be duplicated with sharing disabled but can be duplicated with sharing enabled, then it is possible that you have encountered a bug in this feature. In that case gather as much information as possible and submit a bug report.

4.3 Microsoft Visual C++ Version Considerations

The JDK 7 software is built on Windows using Microsoft Visual Studio 2010 Professional for both 32-bit and 64-bit platforms. If you experience a crash with a Java application and if you have native or JNI libraries that are compiled with a different release of the compiler, then you must consider compatibility issues between the runtimes. Specifically, your environment is supported only if you follow the Microsoft guidelines when dealing with multiple runtimes. For example, if you allocate memory using one runtime, then you must release it using the same runtime. Unpredictable behavior or crashes can arise if you release a resource using a different library than the one that allocated the resource.

Troubleshooting Hanging or Looping Processes

This chapter provides information and guidance on some specific procedures for troubleshooting hanging or looping processes.

Problems can occur that involve hanging or looping processes. A hang can occur for many reasons but often stems from a deadlock in application code, API code, or library code. A hang can even be due to a bug in the HotSpot virtual machine.

Sometimes an apparent hang turns out to be, in fact, a loop. For example, a bug in a VM process that causes one or more threads to go into an infinite loop can consume all available CPU cycles.

An initial step when diagnosing a hang is to find out if the VM process is idle or consuming all available CPU cycles. To do this requires using an operating system utility. If the process appears to be busy and is consuming all available CPU cycles then it is likely that the issue is a looping thread rather than a deadlock. On Solaris OS, for example, the command `prstat -L -p <pid>` can be used to report the statistics for all LWPs in the target process and thus will identify the threads that are consuming a lot of CPU cycles.

5.1 Diagnosing a Looping Process

If a VM process appears to be looping, the first step is to try to get a thread dump. If a thread dump can be obtained, it will often be clear which thread is looping. If the looping thread can be identified, then the trace stack in the thread dump can provide direction on where (and maybe why) the thread is looping.

If the application console (standard input/output) is available, then press the Ctrl-\ key combination (on Solaris OS or Linux) or the Ctrl-Break key combination (on Windows) to cause the HotSpot VM to print a thread dump, including thread state. On Solaris OS and Linux the thread dump can also be obtained by sending a `SIGQUIT` to the process (command `kill -QUIT <pid>`). In this case the thread dump is printed to the standard output of the target process. The output might be directed to a file, depending on how the process was started.

If the Java process is started with the `-XX:+PrintClassHistogram` command-line option, then the Ctrl-Break handler will produce a heap histogram.

If a thread dump can be obtained, then a good place to start is the thread stacks of the threads that are in the runnable state. See 2.15.1 Thread Dump for information on the format of the thread dump, as well as a table of the possible thread states in the thread dump. In some cases it might be necessary to get a sequence of thread dumps in order to determine which threads appear to be continuously busy.

If the application console is not available (process is running as a background process, or the VM output is directed to an unknown location), then the `jstack` utility can be used to obtain the stack thread. Use the `jstack -F` *pid* option to force a stack dump of the looping process. See 2.11 `jstack` Utility for information on the output of this utility. The `jstack` utility should also be used if the thread dump does not provide any evidence that a Java thread is looping.

When reviewing the output of the `jstack` utility, focus initially on the threads that are in the RUNNABLE state. This is the most likely state for threads that are busy and possibly looping. It might be necessary to execute `jstack` a number of times to get a more complete picture of which threads are looping. If a thread appears to be always in theRUNNABLE state, then the `-m` option can be used to print the native frames and can provide a further hint on what the thread is doing. If a thread appears to be looping continuously while in the RUNNABLE state, this situation can indicate a potential HotSpot VM bug that needs further investigation.

If the VM does not respond to a Ctrl-\ this could indicate a VM bug rather than an issue with application or library code. In this case use `jstack` with the `-m` option (in addition to the `-F` option) to get a thread stack for all threads. The output will include the thread stacks for VM internal threads. In this stack trace, identify threads that do not appear to be waiting. For example, on Solaris OS you identify the threads that are not in functions such as `__lwp_cond_wait`, `__lwp_park`, `___pollsys`, or other blocking functions. If it appears that the looping is caused by a VM bug, then collect as much data as possible and submit a bug report. See Chapter 7, Submitting Bug Reports for more details on data collection.

5.2 Diagnosing a Hung Process

If the application appears to be hung and the process appears to be idle, then the first step is to try to obtain a thread dump. If the application console is available, then press the Ctrl-\ keys (on Solaris OS or Linux) or the Ctrl-Break keys (on Windows) to cause the HotSpot VM to print a thread dump. On Solaris OS and Linux the thread dump can also be obtained by sending a `SIGQUIT` to the process (command `kill -QUIT <pid>`).

5.2.1 Deadlock Detected

If the hung process is capable of generating a thread dump, then the output is printed to the standard output of the target process. After printing the thread dump, the HotSpot VM executes a deadlock detection algorithm. If a deadlock is detected it will be printed along with the stack trace of the threads involved in the deadlock. Below is an example of this output.

```
Found one Java-level deadlock:
============================
"AWT-EventQueue-0":
  waiting to lock monitor 0x000ffbf8 (object 0xf0c30560, a java.awt.Component$AWTTreeLock),
  which is held by "main"
"main":
  waiting to lock monitor 0x000ffe38 (object 0xf0c41ec8, a java.util.Vector),
  which is held by "AWT-EventQueue-0"

Java stack information for the threads listed above:
===================================================
"AWT-EventQueue-0":
        at java.awt.Container.removeNotify(Container.java:2503)
        - waiting to lock <0xf0c30560> (a java.awt.Component$AWTTreeLock)
        at java.awt.Window$1DisposeAction.run(Window.java:604)
        at java.awt.Window.doDispose(Window.java:617)
        at java.awt.Dialog.doDispose(Dialog.java:625)
        at java.awt.Window.dispose(Window.java:574)
        at java.awt.Window.disposeImpl(Window.java:584)
        at java.awt.Window$1DisposeAction.run(Window.java:598)
        - locked <0xf0c41ec8> (a java.util.Vector)
        at java.awt.Window.doDispose(Window.java:617)
        at java.awt.Window.dispose(Window.java:574)
        at javax.swing.SwingUtilities$SharedOwnerFrame.dispose(SwingUtilities.java:1743)
        at javax.swing.SwingUtilities$SharedOwnerFrame.windowClosed(SwingUtilities.java:1722)
        at java.awt.Window.processWindowEvent(Window.java:1173)
        at javax.swing.JDialog.processWindowEvent(JDialog.java:407)
        at java.awt.Window.processEvent(Window.java:1128)
        at java.awt.Component.dispatchEventImpl(Component.java:3922)
        at java.awt.Container.dispatchEventImpl(Container.java:2009)
        at java.awt.Window.dispatchEventImpl(Window.java:1746)
        at java.awt.Component.dispatchEvent(Component.java:3770)
        at java.awt.EventQueue.dispatchEvent(EventQueue.java:463)
        at java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.java:214)
        at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:163)
        at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:157)
        at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:149)
        at java.awt.EventDispatchThread.run(EventDispatchThread.java:110)
"main":
        at java.awt.Window.getOwnedWindows(Window.java:844)
        - waiting to lock <0xf0c41ec8> (a java.util.Vector)
        at
javax.swing.SwingUtilities$SharedOwnerFrame.installListeners(SwingUtilities.java:1697)
        at javax.swing.SwingUtilities$SharedOwnerFrame.addNotify(SwingUtilities.java:1690)
        at java.awt.Dialog.addNotify(Dialog.java:370)
        - locked <0xf0c30560> (a java.awt.Component$AWTTreeLock)
        at java.awt.Dialog.conditionalShow(Dialog.java:441)
        - locked <0xf0c30560> (a java.awt.Component$AWTTreeLock)
        at java.awt.Dialog.show(Dialog.java:499)
        at java.awt.Component.show(Component.java:1287)
        at java.awt.Component.setVisible(Component.java:1242)
        at test01.main(test01.java:10)

Found 1 deadlock.
```

The default deadlock detection works with locks that are obtained using the `synchronized` keyword, as well as with locks that are obtained using the`java.util.concurrent` package. If the Java VM flag `-XX:+PrintConcurrentLocks` is set, then the stack trace also shows a list of lock owners.

If deadlock is detected, then you must examine the output in more detail in order to understand the deadlock. In the above example the thread `main` is locking object`<0xf0c30560>` and is waiting to enter `0xf0c41ec8`, which is locked by thread `AWT-EventQueue-0`. However, thread `AWT-EventQueue-0` is waiting to enter `0xf0c30560`, which is locked by `main`.

The detail in the stack traces provides information to help find the deadlock.

5.2.2 Deadlock Not Detected

If the thread dump is printed and no deadlocks are found, then the issue might be a bug in which a thread waiting on a monitor that is never notified. This could be a timing issue or a general logic bug.

To find out more about the issue, examine each of the threads in the thread dump and each thread that is blocked in `Object.wait()`. The caller frame in the stack trace indicates the class and method that is invoking the `wait()` method. If the code was compiled with line number information (the default), then this provides direction as to the code to examine. In most cases you must have some knowledge of the application logic or library in order to diagnose this issue further. In general you must understand how the synchronization works in the application and in particular the details and conditions for when and where monitors are notified.

5.2.3 No Thread Dump

If the VM does not respond to a Ctrl-\ or Ctrl-Break, then it is possible that the VM is deadlocked or hung for some other reason. In that case use the `jstack` utility (see 2.11`jstack` Utility) to obtain a thread dump. Use the `jstack -F` *pid* option to force a stack dump of the hung process. This also applies in the case where the application is not accessible or the output is directed to an unknown location.

In the `jstack` output, examine each of the threads in the `BLOCKED` state. The top frame can sometimes indicate why the thread is blocked, for example, `Object.wait` or`Thread.sleep`. The rest of the stack will give an indication of what the thread is doing. This is particularly true when the source has been compiled with line number information (the default) and you can cross reference the source code.

If a thread is in the `BLOCKED` state and the reason is not clear, then use the `-m` option to get a mixed stack. With the mixed stack output, it should be possible to identify why the thread is blocked. If a thread is blocked trying to enter a synchronized method or block, then you will see frames such as `ObjectMonitor::enter` near the top of the stack. Below is an example.

```
----------------- t@13 -----------------
0xff31e8b8      ___lwp_cond_wait + 0x4
0xfea8c810      void ObjectMonitor::EnterI(Thread*) + 0x2b8
0xfeac86b8      void ObjectMonitor::enter2(Thread*) + 0x250
:
```

Threads in the `RUNNABLE` state might also be blocked. The top frames in the mixed stack should indicate what the thread is doing.

One specific thread to check is `VMThread`. This is the special thread used to execute operations like garbage collection. It can be identified as the thread that is executing`VMThread::run()` in its initial frames. On Solaris OS it is typically `t@4`. On Linux it should be identifiable using the C++ mangled name `_ZN8VMThread4loopEv`.

In general the VM thread is in one of three states: waiting to execute a VM operation, synchronizing all threads in preparation for a VM operation, or executing a VM operation. If you suspect that a hang is a HotSpot VM bug rather than an application or class library deadlock, then pay special attention to the VM thread.

If the VM thread appears to be stuck in `SafepointSynchronize::begin`, then this could indicate an issue bringing the VM to a safepoint. A safepoint indicates that all threads executing in the VM are blocked and waiting for a special operation, such as garbage collection, to complete.

If the VM thread appears to be stuck in `function`, where `function` ends in `doit`, then this could also indicate a VM problem.

In general, if you can execute the application from the command line, and you get to a state where the VM does not respond to a Ctrl-\ or Ctrl-Break, it is more likely that you have uncovered a VM bug, a thread library issue, or a bug in another library. If this occurs, obtain a crash dump (see 7.4 Collecting Core Dumps for instructions on how to do this), gather as much information as possible, and submit a bug report or support call.

One other tool to mention in the context of hung processes is the `pstack` utility on Solaris OS. On Solaris 8 and 9 OS, this utility prints the thread stacks for LWPs in the target process. On Solaris 10 OS and starting with the JDK 5.0 release, the output of `pstack` is similar, though not identical, to the output from `jstack -m`. As with `jstack`, the Solaris 10 OS implementation of `pstack` prints the fully qualified class name, method name, and bci. It will also print line numbers for the cases where the source was compiled with line number information (the default). This is useful for developers and administrators who are familiar with the other utilities on Solaris OS that exercise features of the `/proc` file system.

The equivalent tool of `pstack` on Linux is `lsstack`. This utility is included in some distributions and otherwise obtained from the `sourceforge.net` web site. At the time of this writing, `lsstack` reported native frames only.

5.3 Solaris 8 OS Thread Library

The default thread library on Solaris 8 OS is often referred to as the T1 library. This thread library implemented the *m*:*n* threading model, where *m* user threads are mapped to *n* kernel-level threads (LWPs). Solaris 8 OS also shipped with an alternative and newer thread library in `/usr/lib/lwp`. The alternative thread library is often referred to as the T2 library, and it became the default thread library in Solaris 9 and 10 OS. In older releases of J2SE (pre-1.4.0 in particular) there were a number of issues with the default thread library, for example, bugs in the thread library, LWP synchronization problems, or LWP starvation. LWP starvation is a scenario in which there are user threads in the runnable state but there are no kernel level threads available.

Although the issues cited are historical, it should be noted that when the JDK software is deployed on Solaris 8 OS, it still uses the T1 library by default. LWP starvation type issues do not arise because the JDK release uses “bound threads” so that each user thread is bound to a kernel thread. However in the event that you encounter an issue, such as a hang, which you believe is a thread library issue, then you can instruct the HotSpot VM to use the T2 library by adding `/usr/lib/lwp` to the `LD_LIBRARY_PATH`. To check if the T2 library is in use, issue the command `pldd <pid>` to list the libraries loaded by the specified process.

Integrating Signal and Exception Handling

Sometimes developers have to integrate Java applications with code that uses signal or exception handlers. This chapter provides information on how signals are handled in the HotSpot Virtual Machine. It also describes the signal chaining facility that facilitates writing applications that need to install their own signal handlers. The signal chaining facility is available on Solaris OS and Linux.

6.1 Signal Handling on Solaris OS and Linux

The HotSpot Virtual Machine installs signal handlers to implement various features and to handle fatal error conditions. For example, in an optimization to avoid explicit null checks in cases where `java.lang.NullPointerException` will be thrown rarely, the `SIGSEGV` signal is caught and handled, and the `NullPointerException` is thrown.

In general there are two categories of situations where signal/traps arise.

- Situations in which signals are expected and handled. Examples include the implicit null handling cited above. Another example is the safepoint polling mechanism, which protects a page in memory when a safepoint is required. Any thread that accesses that page causes a `SIGSEGV`, which results in the execution of a stub that brings the thread to a safepoint.
- Unexpected signals. This includes a `SIGSEGV` when executing in VM code, JNI code, or native code. In these cases the signal is unexpected, so fatal error handling is invoked to create the error log and terminate the process.

The following table lists the signals that are currently used on Solaris OS and Linux. The mention “optional” means that the signal is not necessary when the `-Xrs` option is specified, as explained in 6.1.1 Reducing Signal Usage. The mention “configurable” means that alternative signals may be specified, as explained in 6.1.2 Alternative Signals. See 6.1.3 Signal Chaining for detailed information about signal chaining.

Signal	Description
`SIGSEGV`, `SIGBUS`, `SIGFPE`, `SIGPIPE`, `SIGILL`	Used in the implementation for implicit null check, and so forth.
`SIGQUIT`	Thread dump support: To dump Java stack traces at the standard error stream. (Optional.)
`SIGTERM`, `SIGINT`, `SIGHUP`	Used to support the shutdown hook mechanism (`java.lang.Runtime.addShutdownHook`) when the VM is terminated abnormally. (Optional.)
`SIGUSR1`	Used in the implementation of the `java.lang.Thread.interrupt` method. (Configurable.) Not used starting with Solaris 10 OS. Reserved on Linux.
`SIGUSR2`	Used internally. (Configurable.) Not used starting with Solaris 10 OS.
`SIGABRT`	The HotSpot VM does not handle this signal. Instead it calls the abort function after fatal error handling. If an application uses this signal then it should terminate the process to preserve the expected semantics.

6.1.1 Reducing Signal Usage

The `-Xrs` option instructs the HotSpot VM to reduce its signal usage. With this option fewer signals are used, although the VM installs its own signal handler for essential signals such as `SIGSEGV`. In the above table the signals tagged as optional are not used when the `-Xrs` option is specified. Specifying this option means that the shutdown hook mechanism will not execute if the process receives a `SIGQUIT`, `SIGTERM`, `SIGINT`, or `SIGHUP`. Shutdown hooks will execute, as expected, if the VM terminates normally (last non-daemon thread completes or the `System.exit` method is used).

6.1.2 Alternative Signals

On Solaris 8 and 9 OS, the `-XX:+UseAltSigs` option can be used to instruct the HotSpot VM to use alternative signals to `SIGUSR1` and `SIGUSR2`. Starting with Solaris 10 OS, this option is ignored, as the operating system reserves two additional signals (called `SIGJVM1` and `SIGJVM2`).

On Linux, the handler for `SIGUSR1` cannot be overridden. `SIGUSR2` is used to implement suspend and resume. However it is possible to specify an alternative signal to be used instead of `SIGUSR2`. This is done by specifying the `_JAVA_SR_SIGNUM` environment variable. If this environment variable is set, it must be set to a value larger than the maximum of `SIGSEGV` and `SIGBUS`.

6.1.3 Signal Chaining

If an application with native code requires its own signal handlers, then it might need to be used with the signal chaining facility. The signal chaining facility offers the following features:

- Support for pre-installed signal handlers when the HotSpot VM is created.

  When the VM is first created, existing signal handlers, that is, handlers for signals that are used by the VM, are saved. During execution, when any of these signals are raised and found not to be targeted at the Java HotSpot VM, the pre-installed handlers are invoked. In other words, pre-installed handlers are **chained** behind the VM handlers for these signals.

- Support for signal handler installation after the HotSpot VM is created, either inside JNI code or from another native thread.

  An application can link and load the `libjsig.so` shared library before `libc/libthread/libpthread`. This library ensures that calls such as `signal()`, `sigset()`, and `sigaction()` are intercepted so that they do not actually replace the Java HotSpot VM's signal handlers if the handlers conflict with those already installed by the Java HotSpot VM. Instead, these calls save the new signal handlers, or **chain** them behind the VM-installed handlers. During execution, when any of these signals are raised and found not to be targeted at the Java HotSpot VM, the pre-installed handlers are invoked.

  If support for signal handler installation after the creation of the VM is not required, then the `libjsig.so` shared library is not needed.

  Perform one of these two procedures to use the `libjsig.so` shared library.

  - Link it with the application that creates/embeds a HotSpot VM, for example:

    `cc -L` *libjvm.so-directory* **`-ljsig -ljvm java_application.c`**

  - Use the `LD_PRELOAD` environment variable, for example:

    `export LD_PRELOAD=`*libjvm.so-directory*`/libjsig.so; java_application(ksh)`**

    `setenv LD_PRELOAD` *libjvm.so-directory*`/libjsig.so; java_application(csh)`**

  The interposed `signal()`, `sigset()`, and `sigaction()` return the saved signal handlers, not the signal handlers installed by the Java HotSpot VM and which are seen by the operating system.

Note that `SIGUSR1` cannot be chained. If an application attempts to chain this signal on Solaris OS, then the HotSpot VM terminates with the following fatal error:

```
Signal chaining detected for VM interrupt signal, try -XX:+UseAltSigs
```

In addition, the `SIGQUIT`, `SIGTERM`, `SIGINT`, and `SIGHUP` signals cannot be chained. If the application needs to handle these signals, consider using the `-Xrs` option.

On Solaris OS, the `SIGUSR2` signal can be chained, but only for non-Java and non-VM threads; that is, it can only be used for native threads created by the application that do not attach to the VM.

6.2 Exception Handling on Windows

On Windows, an exception is an event that occurs during the execution of a program. There are two kinds of exceptions: hardware exceptions and software exceptions. Hardware exceptions are comparable to signals such as `SIGSEGV` and `SIGKILL` on Solaris OS and Linux. Software exceptions are initiated explicitly by applications or the operating system using the `RaiseException()` API.

On Windows, the mechanism for handling both hardware and software exceptions is called **structured exception handling (SEH)**. This is stack frame-based exception handling similar to the C++ and Java exception handling mechanism. In C++ the `__try` and `__except` keywords are used to guard a section of code that might result in an exception, as in the following example:

```
__try {
     // guarded body of code
 } __except (filter-expression) {
     // exception-handler block
 }
```

The `__except` block is filtered by a filter expression that uses an exception code (integer code returned by the `GetExceptionCode()` API), or exception information (`GetExceptionInformation()` API), or both.

The filter expression should evaluate to one of the following values:

- `EXCEPTION_CONTINUE_EXECUTION` = -1

  The filter expression has repaired the situation, and execution continues where the exception occurred. Unlike some exception schemes, SEH supports the **resumption model** as well. This is much like Unix signal handling in the sense that after the signal handler finishes, the execution continues where the program was interrupted. The difference is that the handler in this case is just the filter expression itself and not the `__except` block. However, the filter expression might also involve a function call.

- `EXCEPTION_CONTINUE_SEARCH` = 0

  The current handler cannot handle this exception. Continue the handler search for the next handler. This is similar to the catch block not matching an exception type in C++ and Java.

- `EXCEPTION_EXECUTE_HANDLER` = 1

  The current handler matches and can handle the exception. The `__except` block is executed.

The `__try` and `__finally` keywords are used to construct a termination handler as shown below.

```
__try {
    // guarded body of code
} __finally {
    // __finally block
}
```

When control leaves the `__try` block (after exception or without exception), the `__finally` block is executed. Inside the `__finally` block, the `AbnormalTermination()` API can be called to test whether control continued after the exception or not.

Windows programs can also install a top-level **unhandled exception filter** function to catch exceptions that are not handled in a `__try`/`__except` block. This function is installed on a process-wide basis using the `SetUnhandledExceptionFilter()` API. If there is no handler for an exception, then `UnhandledExceptionFilter()` is called, and this will call the top-level unhandled exception filter function, if any, to catch that exception. This function also shows a message box to notify the user about the unhandled exception.

Windows exceptions are comparable to Unix synchronous signals that are attributable to the current execution stream. In Windows, asynchronous events such as console events (for example, the user pressing Ctrl-C at the console) are handled by the console control handler registered using the `SetConsoleCtlHandler()` API.

If an application uses the `signal()` API on Windows, then the C runtime library (CRT) maps both Windows exceptions and console events to appropriate signals or C runtime errors. For example, CRT maps Ctrl-C to `SIGINT` and all other console events to `SIGBREAK`. Similarly, if you register the `SIGSEGV` handler, the C runtime library translates the corresponding exception to a signal. The CRT library startup code implements a `__try`/`__except` block around the `main()` function. The CRT's exception filter function (named `_XcptFilter`) maps the Win32 exceptions to signals and dispatches signals to their appropriate handlers. If a signal's handler is set to `SIG_DFL` (default handling), then `_XcptFilter` calls `UnhandledExceptionFilter`.

With Windows XP or Windows 2003, the **vectored exception handling** mechanism can also be used. Vectored handlers are not frame-based handlers. A program can register zero or more vectored exception handlers using the `AddVectoredExceptionHandler` API. Vectored handlers are invoked before structured exception handlers, if any, are invoked, regardless of where the exception occurred.

Vectored exception handler returns one of the following values:

- `EXCEPTION_CONTINUE_EXECUTION`: Skip next vectored and SEH handlers.
- `EXCEPTION_CONTINUE_SEARCH`: Continue next vectored or SEH handler.

Refer to the Microsoft web site at http://www.microsoft.com for further information on Windows exception handling.

6.2.1 Signal Handling in the HotSpot Virtual Machine

The HotSpot VM installs a top-level exception handler using the `SetUnhandledExceptionFilter` API (or the `AddVectoredExceptionHandler` API for 64-bit) during VM initialization.

It also installs win32 SEH using a `__try` /`__except` block in C++ around the thread (internal) start function call for each thread created.

Finally, it installs an exception handler around JNI functions.

If an application must handle structured exceptions in JNI code, it can use `__try` /`__except` statements in C++. However, if it must use the vectored exception handler in JNI code then the handler must return `EXCEPTION_CONTINUE_SEARCH` to continue to the VMs exception handler.

In general, there are two categories of situations in which exceptions arise:

- Situations where signals are expected and handled. Examples include the implict null handling cited above where accessing a null causes an`EXCEPTION_ACCESS_VIOLATION`, which is handled.
- Unexpected exceptions. An example is `EXCEPTION_ACCESS_VIOLATION` when executing in VM code or in JNI or native code. In these cases the signal is unexpected, and fatal error handling is invoked to create the error log and terminate the process.

6.2.2 Console Handlers

The HotSpot Virtual Machine registers console events as shown in the following table.

Console Event	Signal	Usage
`CTRL_C_EVENT`	`SIGINT`	Terminate process. (optional)
`CTRL_CLOSE_EVENT` `CTRL_LOGOFF_EVENT``CTRL_SHUTDOWN_EVENT`	`SIGTERM`	Used by the shutdown hook mechanism when the VM is terminated abnormally. (optional)
`CTRL_BREAK_EVENT`	`SIGBREAK`	Thread dump support. To dump Java stack traces at the standard error stream. (optional)

If an application must register its own console handler, then the `-Xrs` option can be used. With this option, shutdown hooks are not run on `SIGTERM` (with above mapping of events) and thread dump support is not available on `SIGBREAK` (with above mapping Ctrl-Break event).

Submitting Bug Reports

This chapter provides guidance on how to submit a bug report. It includes suggestions about what to try before submitting a report and what data to collect for the report.

7.1 Checking for Existing Fixes in Update Releases

The current version is JDK 7. Regularly scheduled updates to this release contain fixes for a set of critical bugs identified since the initial release of the platform. When an update release becomes available, it becomes the default download at the Java SE Downloads site.

The download site includes release notes that list the bug fixes in the release. Each bug in the list is linked to the bug description in the bug database. The release notes also include the list of fixes in previous update releases. If you encounter an issue, or suspect a bug, then, as an early step in the diagnosis, check the list of fixes that are available in the most recent update release.

Sometimes it is not obvious if an issue is a duplicate of a bug that is already fixed. Therefore, where possible, test with the latest update release to see if the problem persists.

7.2 Preparing to Submit a Bug Report

Before submitting a big report, consider the following recommendations:

- Collect as much relevant data as possible. For example, generate a thread-dump in the case of a deadlock, or locate the core file (where applicable) and `hs_err` file in the case of a crash. In all cases it is important to document the environment and the actions performed just before the problem is encountered.
- Where applicable, try to restore the original state and reproduce the problem using the documented steps. This helps to determine if the problem is reproducible or an intermittent issue.
- If the issue is reproducible, try to narrow down the problem. In some cases, a bug can be demonstrated with a small standalone test case. Bugs that are demonstrated by small test cases will typically be easy to diagnose when compared to test cases that consist of a large complex application.
- Search the bug database to see if this bug or a similar bug has been reported. If the bug has already been reported, the bug report might have further information, such as the following:
  - If the bug has already been fixed, the release in which it was fixed.
  - A workaround for the problem.
  - Comments in the evaluation that explain, in further detail, the circumstances that cause the bug to arise.

  The bug database is located at http://bugs.sun.com/bugdatabase/index.jsp.

- If you conclude that the bug has not already been reported, submit a new bug.

Before submitting a bug, verify that the environment where the problem arises is a supported configuration. See the Supported System Configurations site.

In addition to the system configurations, check the list of supported locales. See the Supported Locales web page.

In the case of the Solaris Operating System, check the recommended patch cluster for the operating system release to ensure that the recommended patches are installed.

7.3 Collecting Data for a Bug Report

In general it is recommended to collect as much relevant data as possible when you create a bug report or submit a support call. This section suggests the data to collect and, where applicable, it provides recommendations for the commands or general procedure for obtaining the data.

The following data can be collected prior to submitting a bug report:

- Hardware details
- Operating system details
- JDK version information
- Command-line options
- Environment variables

- Fatal error log (in the case of a crash)
- Core or crash dump (in the case of a crash and possibly a hang)
- Detailed description of the problem, including test case (where possible)
- Logs or trace information (where applicable)
- Results from troubleshooting steps

The following sections present more detail for each type of data.

7.3.1 Hardware Details

Sometimes a bug arises or can be reproduced only on certain hardware configurations. If a fatal error occurs, the error log might contain the hardware details. If an error log is not available, document in the bug report the number and the type of processors in the machine, the clock speed, and, where applicable and if known, some details on the features of that processor. For example, in the case of Intel processors, it might be relevant that hyper-threading is available.

7.3.2 Operating System

On the Solaris Operating System, the `showrev -a` command prints the operating system version and patch information.

On Linux, it is important to know which distribution and version is used. Sometimes the `/etc/*release` file indicates the release information, but as components and packages can be upgraded independently, it is not always a reliable indication of the configuration. Therefore, in addition to the information from the `*release` file, collect the following information:

- The kernel version. This can be obtained using the `uname -a` command.
- The `glibc` version. The `rpm -q glibc` command indicates the patch level of `glibc`.
- The thread library. There are two thread libraries for Linux, namely `LinuxThreads` and `NPTL`.
  The `LinuxThreads` library is used on 2.4 and older kernels and has “fixed stack” and “floating stack” variants. The Native POSIX Thread Library (`NTPL`) is used on the 2.6 kernel. Some Linux releases (such as RHEL3) include backports of`NPTL`to the 2.4 kernel. Use the command `getconf GNU_LIBPTHREAD_VERSION` to determine which thread library is used. If the `getconf` command returns an error to say that the variable does not exist, then it is likely that you are using an old kernel with the `LinuxThreads` library.

7.3.3 JDK Version

The JDK version string can be obtained using the `java -version` command.

Multiple versions of the JDK may be installed on the same machine. Therefore, ensure that you use the appropriate version of the `java` command by verifying that the installation `bin` directory appears in your `PATH` environment variable before other installations.

7.3.4 Command-Line Options

If the bug report does not include a fatal error log, it is important to document the full command line and all its options. This includes any options that specify heap settings (for example, the `-mx` option) or any `-XX` options that specify HotSpot specific options.

One of the features in the JDK is garbage collector ergonomics. On server-class machines the `java` command launches the HotSpot Server VM and a parallel garbage collector. A machine is considered to be a server machine if it has at least two processors and 2GB or more of memory.

The `-XX:+PrintCommandLineFlags` option can be used to verify the command-line options. This option prints all command-line flags to the VM. The command-line options can also be obtained for a running VM or core file using the `jmap` utility.

7.3.5 Environment Variables

Sometimes problems arise due to environment variable settings. When creating the bug report, indicate the values of the following Java environment variables (if set).

- `JAVA_HOME`
- `JRE_HOME`
- `JAVA_TOOL_OPTIONS`
- `_JAVA_OPTIONS`

- CLASSPATH
- JAVA_COMPILER
- PATH
- USERNAME

In addition, collect the following operating-system-specific environment variables.

- On Solaris OS and Linux, collect the values of the following environment variables.
  - LD_LIBRARY_PATH
  - LD_PRELOAD
  - SHELL
  - DISPLAY
  - HOSTTYPE
  - OSTYPE
  - ARCH
  - MACHTYPE
- On the Linux operating system, collect the values of the following environment variables.
  - LD_ASSUME_KERNEL
  - _JAVA_SR_SIGNUM
- On the Windows operating system, collect the values of the following environment variables.
  - OS
  - PROCESSOR_IDENTIFIER
  - _ALT_JAVA_HOME_DIR

7.3.6 Fatal Error Log

When a fatal error occurs, an error log is created. See Appendix C, Fatal Error Log for detailed information about this file.

The error log contains much information obtained at the time of the fatal error, such as version and environment information, details on the threads that provoked the crash, and so forth.

If the fatal error log is generated, be sure to include it in the bug report or support call.

7.3.7 Core or Crash Dump

Core and crash dumps can be very useful when trying to diagnose a system crash or hung process. The procedure for generating a dump is described in 7.4 Collecting Core Dumps.

7.3.8 Detailed Description of the Problem

When creating a problem description, try to include as much relevant information as possible. Describe the application, the environment, and most importantly the events leading up to the time when the problem was encountered.

- If the problem is reproducible, list the stepsthat are required to demonstrate the problem.
- If the problem can be demonstrated with a small test case, include the test case and the commands to compile and execute the test case.
- If the test case or problem requires third-party code (for example, a commercial or open source library or package), provide details on where and how to obtain the library.

Sometimes the problem can be reproduced only in a complex application environment. In this case, the description, coupled with logs, core file, and other relevant information, might be the sole means to diagnose the issue. In these situations the description should indicate if the submitter is willing to run further diagnosis or run test binaries on the system where the issue arises.

7.3.9 Logs and Traces

In some cases, log or trace output can help to quickly determine the cause of a problem.

For example, in the case of a performance issue the output of the `-verbose:gc` option can help in to diagnosing the problem. (This is the option to enable output from the garbage collector.)

In other cases the output from the `jstat` command can be used to capture statistical information over the time period leading up to the problem.

In the case of a deadlock or a hung VM (for example, due to a loop) the thread stacks can help diagnose the problem. The thread stacks are obtained using Ctrl-\ on Solaris OS and Linux and Ctrl-Break on the Windows operating system.

In general, include all relevant logs, traces and other output in the bug report or support call.

7.3.10 Results from Troubleshooting Steps

Before submitting the bug report, be sure to document any troubleshooting steps that were performed.

For example, if the problem is a crash and the application has native libraries, you might have already run the application with the `-Xcheck:jni` option to reduce the likelihood that the bug is in the native code. Another case could be a crash that occurs with the HotSpot Server VM (`-server` option). If you have also tested with the HotSpot Client VM (`-client` option) and the problem does not occur, this gives an indication that the bug might be specific to the HotSpot Server VM.

In general, include in the bug report all troubleshooting steps and results that have already occurred. This type of information can often reduce the time that is required to diagnose an issue.

7.4 Collecting Core Dumps

This section explains how to generate and collect core dumps (also known as crash dumps). A core dump or a crash dump is a memory snapshot of a running process. A core dump can be automatically created by the operating system when a fatal or unhandled error (for example, signal or system exception) occurs. Alternatively, a core dump can be forced by means of system-provided command-line utilities. Sometimes a core dump is useful when diagnosing a process that appears to be hung; the core dump may reveal information about the cause of the hang.

When collecting a core dump, be sure to gather other information about the environment so that the core file can be analyzed (for example, OS version, patch information, and the fatal error log).

Core dumps do not usually contain all the memory pages of the crashed or hung process. With each of the operating systems discussed here, the text (or code) pages of the process are not included in core dumps. But to be useful, a core dump must consist of pages of heap and stack as a minimum. Collecting non-truncated good core dump files is essential for postmortem analysis of the crash.

7.4.1 Collecting Core Dumps on Solaris OS

With the Solaris Operating System, unhandled signals such as a segmentation violation, illegal instruction, and so forth, result in a core dump. By default, the core dump is created in the current working directory of the process and the name of the core dump file is `core`. The user can configure the location and name of the core dump using the core file administration utility, `coreadm`. This procedure is fully described in the man page for the `coreadm` utility.

The `ulimit` utility is used to get or set the limitations on the system resources available to the current shell and its descendants. Use the `ulimit -c` command to check or set the core file size limit. Make sure that the limit is set to `unlimited`; otherwise the core file could be truncated. Note that `ulimit` is a Bash shell built-in command; on a C shell, use the `limit` command.

Ensure that any scripts that are used to launch the VM or your application do not disable core dump creation.

The `gcore` utility can be used to get a core image of running processes. This utility accepts a process id (pid) of the process for which you want to force core dump.

To get the list of Java processes running on the machine, you can use any of the following commands:

- `ps -ef | grep java`
- `pgrep java`
- `jps` command. The `jps` command-line utility does not perform name matching (that is, looking for “java” in the process command name) and so it can list Java VM embedded processes as well as the Java processes.

7.4.1.1 Using the `ShowMessageBoxOnError` Option on Solaris OS

A Java process can be started with the `-XX:+ShowMessageBoxOnError` command-line option. When a fatal error is encountered, the process prints a message to standard error and waits for a `yes` or `no` response from standard input. Below is an example of output when an unexpected signal occurs.

```
=======================================================================
Unexpected Error
-----------------------------------------------------------------------
SIGSEGV (0xb) at pc=0xfeba31ac, pid=8677, tid=2
Do you want to debug the problem?
To debug, run 'dbx - 8677'; then switch to thread 2
Enter 'yes' to launch dbx automatically (PATH must include dbx)
Otherwise, press RETURN to abort...
=======================================================================
```

Before answering `yes` or pressing RETURN, use the `gcore` utility to force a core dump. Then you can type `yes` to launch the `dbx` debugger.

7.4.1.2 Suspending a Process using `truss`

In situations where it is not possible to specify the `-XX:+ShowMessageBoxOnError` option, you might be able to use the `truss` utility. This Solaris OS utility is used to trace system calls and signals. You can use this utility to suspend the process when it reaches a specific function or system call.

The following command shows how to use the `truss` utility to suspend a process when the exit system call is executed (in other words, the process is about to exit).

```
$ truss -t \!all -s \!all -T exit -p pid
```

When the process calls `exit`, it will be suspended. At this point, you can attach the debugger to the process or call `gcore` to force a core dump.

7.4.2 Collecting Core Dumps on Linux

On the Linux operating system, unhandled signals such as segmentation violation, illegal instruction, and so forth, result in a core dump. By default, the core dump is created in the current working directory of the process and the name of the core dump file is `core.`*pid*, where *pid* is the process id of the crashed Java process.

The `ulimit` utility is used to get or set the limitations on the system resources available to the current shell and its descendants. Use the `ulimit -c` command to check or set the core file size limit. Make sure that the limit is set to `unlimited`; otherwise the core file could be truncated. Note that `ulimit` is a Bash shell built-in command; on a C shell, use the `limit` command.

Ensure that any scripts that are used to launch the VM or your application do not disable core dump creation.

You can use the `gcore` command in the `gdb` (GNU Debugger) interface to get a core image of a running process. This utility accepts the pid of the process for which you want to force the core dump.

To get the list of Java processes running on the machine, you can use any of the following commands:

- `ps -ef | grep java`
- `pgrep java`
- `jps` command. The `jps` command-line utility does not perform name matching (that is, looking for “java” in the process command name) and so it can list Java VM embedded processes as well as the Java processes.

7.4.2.1 Using the `ShowMessageBoxOnError` Option on Linux

A Java process can be started with the `-XX:+ShowMessageBoxOnError` command-line option. When a fatal error is encountered, the process prints a message to standard error and waits for a `yes` or `no` response from standard input. Below is an example of output when an unexpected signal occurs.

```
=======================================================================
Unexpected Error
-----------------------------------------------------------------------
SIGSEGV (0xb) at pc=0x06232e5f, pid=11185, tid=8194
Do you want to debug the problem?
To debug, run 'gdb /proc/11185/exe 11185'; then switch to thread 8194
```

```
Enter 'yes' to launch gdb automatically (PATH must include gdb)
Otherwise, press RETURN to abort...
================================================================
```

Type `yes` to launch the `gdb` (GNU Debugger) interface, as suggested by the error report shown above. In the `gdb` prompt, you can give the `gcore` command. This command creates a core dump of the debugged process with the name `core.`*pid*, where *pid* is the process ID of the crashed process. Make sure that the `gdb gcore` command is supported in your versions of `gdb`. Look for `help gcore` in the `gdb` command prompt.

7.4.3 Reasons for Not Getting a Core File

The following list explains the major reasons that a core file might not be generated. This list pertains to both Solaris OS and Linux, unless specified otherwise.

- The current user does not have permission to write in the current working directory of the process.
- The current user has write permission on the current working directory, but there is already a file named `core` that has read-only permission.
- The current directory does not have enough space or there is no space left.
- The current directory has a subdirectory named `core`.
- The current working directory is remote. It might be mapped by NFS (Network File System), and NFS failed just at the time the core dump was about to be created.
- Solaris OS only: The `coreadm` tool has been used to configure the directory and name of the core file, but any of the above reasons apply for the configured directory or filename.
- The core file size limit is too low. Check your core file limit using the `ulimit -c` command (Bash shell) or the `limit -c` command (C shell). If the output from this command is not `unlimited`, the core dump file size might not be large enough. If this is the case, you will get truncated core dumps or no core dump at all. In addition, ensure that any scripts that are used to launch the VM or your application do not disable core dump creation.
- The process is running a `setuid` program and therefore the operating system will not dump core unless it is configured explicitly.
- Java specific: If the process received `SIGSEGV` or `SIGILL` but no core dump, it is possible that the process handled it. For example, HotSpot VM uses the `SIGSEGV` signal for legitimate purposes, such as throwing `NullPointerException`, deoptimization, and so forth. The signal is unhandled by the Java VM only if the current instruction (PC) falls outside Java VM generated code. These are the only cases in which HotSpot dumps core.
- Java specific: The JNI Invocation API was used to create the VM. The standard Java launcher was not used. The custom Java launcher program handled the signal by just consuming it and produced the log entry silently. This situation has occurred with certain Application Servers and Web Servers. These Java VM embedding programs transparently attempt to restart (fail over) the system after an abnormal termination. In this case, the fact that a core dump is not produced is a feature and not a bug.

7.4.4 Collecting Crash Dumps on Windows

On the Windows operating system there are three types of crash dumps.

- Dr. Watson logfile, which is a text error log file that includes faulting stack trace and a few other details.
- User minidump, which can be considered a “partial” core dump. It is not a complete core dump, because it does not contain all the useful memory pages of the process.
- Dr. Watson full-dump, which is equivalent to a Unix core dump. This dump contains most memory pages of the process (except for code pages).

When an unexpected exception occurs on Windows, the action taken depends on two values in the following registry key.

```
\\HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\AeDebug
```

The two values are named `Debugger` and `Auto`. The `Auto` value indicates if the debugger specified in the value of the `Debugger` entry starts automatically when an application error occurs.

- A value of 0 for `Auto` means that the system displays a message box notifying the user when an application error occurs.
- A value of 1 for `Auto` means that the debugger starts automatically.

The value of `Debugger` is the debugger command that is to be used to debug program errors.

When a program error occurs, Windows examines the `Auto` value and if the value is 0 it executes the command in the `Debugger` value. If the value for `Debugger` is a valid command, a message box is created with two buttons: OK and Cancel. If the user clicks OK, the program is terminated. If the user clicks Cancel, the specified debugger is started. If the value for the `Auto` entry is set to 1 and the value for the `Debugger` entry specifies the command for a valid debugger, the system automatically starts the debugger and does not generate a message box.

7.4.4.1 Configuring Dr. Watson

The Dr. Watson debugger is used to create crash dump files. By default, the Dr. Watson debugger (`drwtsn32.exe`) is installed into the Windows system folder (`%SystemRoot%\System32`).

To install Dr. Watson as the postmortem debugger, run the following command.

```
drwtsn32 -i
```

To configure name and location of crash dump files, run `drwtsn32` without any options.

```
drwtsn32
```

In the Dr. Watson GUI window, make sure that the Create Crash Dump File checkbox is set and that the crash dump file path and log file path are configured in their respective text fields.

Dr. Watson may be configured to create a full dump using the registry. The registry key is as follows.

```
System Key: [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DrWatson]
Entry Name: CreateCrashDump
Value: (0 = disabled, 1 = enabled)
```

Note that if the application handles the exception, then the registry-configured debugger is not invoked. In that case it might be appropriate to use the `-XX:+ShowMessageBoxOnError` command-line option to force the process to wait for user intervention on fatal error conditions.

7.4.4.2 Forcing a Crash Dump

On the Windows operating system, the `userdump` command-line utility can be used to force a Dr. Watson dump of a running process. The `userdump` utility does not ship with Windows but instead is released as a component of the OEM Support Tools package.

An alternative way to force a crash dump is to use the `windbg` debugger. The main advantage of using `windbg` is that it can attach to process in a non-invasive manner (that is, read-only). Normally Windows terminates a process after a crash dump is obtained but with the non-invasive attach it is possible to obtain a crash dump and let the process continue. To attach the debugger non-invasively requires selecting the Attach to Process option and clicking the Noninvasive checkbox.

When the debugger is attached, a crash dump can be obtained using the following command.

```
.dump /f crash.dmp
```

The `windbg` debugger is included in the “Debugging Tools for Windows” download.

An additional utility in this download is the `dumpchk.exe` utility, which can verify that a memory dump file has been created correctly.

Both `userdump.exe` and `windbg` require the process id (pid) of the process. The `userdump -p` command lists the process and program for all processes. This is useful if you know that the application is started with the `java.exe` launcher. However, if a custom launcher is used (embedded VM), it might be difficult to recognize the process. In that case you can use the`jps` command line utility as it lists the pids of the Java processes only.

As with Solaris OS and Linux, you can also use the `-XX:+ShowMessageBoxOnError` command-line option on Windows. When a fatal error is encountered, the process shows a message box and waits for a yes or no response from the user.

Before clicking Yes or No, you can use the `userdump.exe` utility to generate the Dr. Watson dump for the Java process. This utility can also be used for the case where the process appears to be hung.

Appendix A

Environment Variables and System Properties

This section describes environment variables and system properties that can be useful in troubleshooting situations.

See also 7.3.5 Environment Variables in 7.3 Collecting Data for a Bug Report.

A.1 JAVA_HOME Environment Variable

The JAVA_HOME environment variable indicates the directory where the JDK software is installed.

A.2 JAVA_TOOL_OPTIONS Environment Variable

In many environments the command line to start the application is not readily accessible. This often arises with applications that use embedded VMs (meaning they use the JNI Invocation API to start the VM), or where the startup is deeply nested in scripts. In these environments the JAVA_TOOL_OPTIONS environment variable can be useful to augment a command line.

When this environment variable is set, the JNI_CreateJavaVM function (in the JNI Invocation API) prepends the value of the environment variable to the options supplied in its JavaVMInitArgs argument. In some cases this option is disabled for security reasons, for example, on Solaris OS the option is disabled when the effective user or group ID differs from the real ID.

This environment variable allows you to specify the initialization of tools, specifically the launching of native or Java programming language agents using the -agentlib or -javaagent options. In the following example the environment variable is set so that the HPROF profiler is launched when the application is started.

```
$ export JAVA_TOOL_OPTIONS="-agentlib:hprof"
```

This variable can also be used to augment the command line with other options for diagnostic purposes. For example, you can supply the -XX:OnError option to specify a script or command to be executed when a fatal error occurs.

Since this environment variable is examined at the time that JNI_CreateJavaVM is called, it cannot be used to augment the command line with options that would normally be handled by the launcher, for example, VM selection using the -client or the -server option.

The JAVA_TOOL_OPTIONS environment variable is fully described in the JAVA_TOOL_OPTIONS section of the JVM Tool Interface documentation.

A.3 java.security.debug System Property

The java.security.debug system property controls whether the security system of the JRE prints trace messages during execution. This option can be useful when diagnosing an issue involving a security manager when a SecurityException is thrown.

The property can have the following values:

- access - print all checkPermission results
- jar - print jar verification information
- policy - print policy information
- scl - print permissions that SecureClassLoader assigns

The following sub-options can be used with the access option:

- stack - include stack trace
- domain - dump all domains in context
- failure - before throwing exception, dump the stack and domain that did not have permission

ZC

For example, to print all `checkPermission` results and trace all domains in context, set the `java.security.debug` property to `access,stack`. To trace access failures, set the property to `access,failure`.

The following example shows the output of a `checkPermission` failure.

```
$ java -Djava.security.debug="access,failure" Application
access denied (java.net.SocketPermission server.foobar.com resolve
)
java.lang.Exception: Stack trace
    at java.lang.Thread.dumpStack(Thread.java:1158)
    at java.security.AccessControlContext.checkPermission
                        (AccessControlContext.java:253)
    at java.security.AccessController.checkPermission(AccessController.java:427)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:532)
    at java.lang.SecurityManager.checkConnect(SecurityManager.java:1031)
    at java.net.InetAddress.getAllByName0(InetAddress.java:1117)
    at java.net.InetAddress.getAllByName0(InetAddress.java:1098)
    at java.net.InetAddress.getAllByName(InetAddress.java:1061)
    at java.net.InetAddress.getByName(InetAddress.java:958)
    at java.net.InetSocketAddress.<init>(InetSocketAddress.java:124)
    at java.net.Socket.<init>(Socket.java:178)
    at Test.main(Test.java:7)
```

For more information on the `java.security.debug` system property, refer to the Security Tutorial.

Command-Line Options

This section describes some command line options that can be useful in diagnosing problems.

B.1 HotSpot VM Command-Line Options

Command-line options that are prefixed with `-XX` are specific to the Java HotSpot Virtual Machine. Many of these options are important for performance tuning and diagnostic purposes, and are therefore described in this appendix. All the `-XX` options are described at http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html.

B.1.1 Dynamic Changing of Flag Values

With the `jinfo -flag` command (2.6 `jinfo` Utility) and with the JConsole utility (2.3 JConsole Utility), you can dynamically set, unset, or change the value of certain Java VM flags for a specified Java process.

For the complete list of these flags, use the MBeans tab of the JConsole utility. See the list of values for the `DiagnosticOptions` attribute of the `HotSpotDiagnostic`MBean, which is in the `com.sun.management` domain. These flags are the following:

- `HeapDumpOnOutOfMemoryError`
- `HeapDumpPath`
- `PrintGC`
- `PrintGCDetails`
- `PrintGCTimeStamps`
- `PrintClassHistogram`
- `PrintConcurrentLocks`

B.1.2 `-XX:+HeapDumpOnOutOfMemoryError` Option

The `-XX:+HeapDumpOnOutOfMemoryError` command-line option tells the HotSpot VM to generate a heap dump when an allocation from the Java heap or the permanent generation cannot be satisfied. There is no overhead in running with this option, and so it can be useful for production systems where `OutOfMemoryError` takes a long time to surface.

You can also specify this option at runtime with the MBeans tab in the `jconsole` utility.

The heap dump is in HPROF binary format, and so it can be analyzed using any tools that can import this format. For example, the `jhat` tool can be used to do rudimentary analysis of the dump. See 2.5 `jhat` Utility.

The following example shows the result of running out of memory with this flag set.

```
$ java -XX:+HeapDumpOnOutOfMemoryError -mn256m -mx512m ConsumeHeap
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid2262.hprof ...
Heap dump file created [531535128 bytes in 14.691 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at ConsumeHeap$BigObject.(ConsumeHeap.java:22)
        at ConsumeHeap.main(ConsumeHeap.java:32)
```

The `ConsumeHeap` fills up the Java heap and runs out of memory. When `java.lang.OutOfMemoryError` is thrown, a heap dump file is created. In this case the file is 507MB and is created as `java_pid2262.hprof` in the current directory.

By default the heap dump is created in a file called `java_pid`*pid*`.hprof` in the working directory of the VM, as in the example above. You can specify an alternative file name or directory with the `-XX:HeapDumpPath=` option. For example `-XX:HeapDumpPath=/disk2/dumps` will cause the heap dump to be generated in the `/disk2/dumps`directory.

B.1.3 `-XX:OnError=` Option

When a fatal error occurs, the HotSpot Virtual Machine can optionally execute a user-supplied script or command. The script or command is specified using the `-XX:OnError=`*string* command line option, where *string* is a single command, or a list of commands separated by a semicolon. Within *string*, all occurrences of `%p` are replaced with the current process ID (pid), and all occurrences of `%%` are replaced by a single `%`. The following examples demonstrate how this option can be used.

On Solaris OS the `pmap` command displays information about the address space of a process. In the following example, if a fatal error occurs, the `pmap` command is executed to display the address space of the process.

```
java -XX:OnError="pmap %p" MyApplication
```

The following example shows how the fatal error report can be mailed to a support alias when a fatal error is encountered

```
java -XX:OnError="cat hs_err_pid%p.log|mail support@acme.com" \ MyApplication
```

On Solaris OS the `gcore` command creates a core image of the specified process, and the `dbx` command launches the debugger. In the following example, the `gcore`command is executed to create the core image, and the debugger is started to attach to the process.

```
java -XX:OnError="gcore %p;dbx - %p" MyApplication
```

In the following Linux example, the `gdb` debugger is launched when an unexpected error is encountered. Once launched, `gdb` will attach to the VM process

```
java -XX:OnError="gdb - %p" MyApplication
```

On Windows the Dr. Watson debugger can be configured as the post-mortem debugger so that a crash dump is created when an unexpected error is encountered. See 7.4.4 Collecting Crash Dumps on Windows for other details.

An alternate approach to obtaining a crash dump on Windows it to use the `-XX:OnError` option to execute the `userdump.exe` utility, as follows.

```
java -XX:OnError="userdump.exe %p" MyApplication
```

The `userdump` utility is part of the Microsoft OEM Support Tools package, which can be downloaded from the Microsoft site. See 7.4.4 Collecting Crash Dumps on Windowsfor further details.

The above example assumes that the path to the `userdump.exe` utility is defined in the `PATH` variable.

B.1.4 `-XX:+ShowMessageBoxOnError` Option

When the option `-XX:+ShowMessageBoxOnError` is set and a fatal error is encountered, the HotSpot VM will display information about the fatal error and prompt the user to specify whether the native debugger is to be launched. In the case of Solaris OS and Linux, the output and prompt are sent to the application console (standard input and standard output). In the case of Windows, a Windows message box pops up.

Below is an example from a fatal error encountered on a Linux system.

```
========================================================================
Unexpected Error
------------------------------------------------------------------------
SIGSEGV (0xb) at pc=0x2000000001164db1, pid=10791, tid=1026

Do you want to debug the problem?

To debug, run 'gdb /proc/10791/exe 10791'; then switch to thread 1026
```

```
Enter 'yes' to launch gdb automatically (PATH must include gdb)
Otherwise, press RETURN to abort...
================================================================================
```

In this case a SIGSEGV error has occurred and the user is prompted to specify whether the gdb debugger is to be launched to attach to the process. If the user enters y or yes,gdb will be launched (assuming it is on the PATH variable).

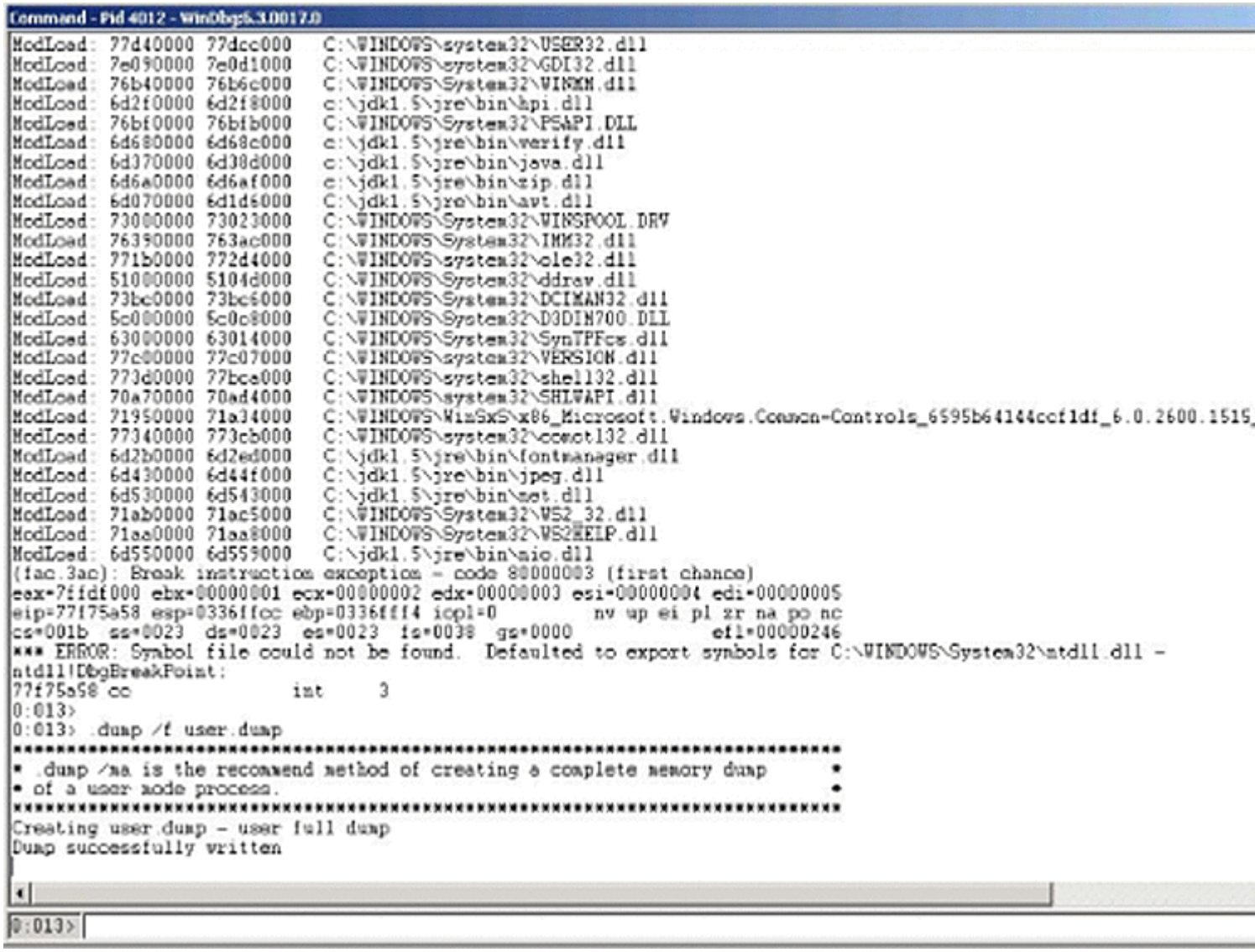
On Solaris OS the message is similar to the above except that the user is prompted to start the dbx debugger. On Windows a message box is displayed. If the user presses the YES button, the VM will attempt to start the default debugger. This debugger is configured by a registry setting; see 7.4.4 Collecting Crash Dumps on Windows for further details. If Microsoft Visual Studio is installed, the default debugger is typically configured to be msdev.exe.

In the above example the output includes the process ID (10791 in this case) and also the thread ID (1026 in this case). If the debugger is launched, one of the initial steps in the debugger might be to select the thread and obtain its stack trace.

As the process is waiting for a response it is possible to use other tools to obtain a crash dump or query the state of the process. On Solaris OS, for example, a core dump can be obtained using the gcore utility.

On Windows a Dr. Watson crash dump can be obtained using the userdump or windbg programs. The windbg program is included in Microsoft's Debugging Tools for Windows. See 7.4.4 Collecting Crash Dumps on Windows for further information on windbg and the link to the download location. In windbg select the Attach to a Process menu option, which displays the list of processes and prompts for the process ID. The HotSpot VM displays a message box, which includes the process ID. Once selected the .dump /f command can be used to force a crash dump. In the following example a crash dump is created in file crash.dump.

Creating Crash Dump with windbg



In general the `-XX:+ShowMessageBoxOnError` option is more useful in a development environment where debugger tools are available. The `-XX:OnError` option is more suitable for production environments where a fixed sequence of commands or scripts are executed when a fatal error is encountered.

B.1.5 Other -xx Options

Several other `-XX` command-line options can be useful in troubleshooting.

- `-XX:OnOutOfMemoryError=`*string* is used to specify a command or script to execute when an `OutOfMemoryError` is first thrown.
- `-XX:ErrorFile=`*filename* is used to specify a location for the fatal error log file. See C.1 Location of Fatal Error Log.
- `-XX:HeapDumpPath=`*path* is used to specify a location for a heap dump. See B.1.2 - `XX:+HeapDumpOnOutOfMemoryError` Option.
- `-XX:MaxPermSize=`*size* is used to specify the size of the permanent generation memory. See 3.1.2 Detail Message: `PermGen space`.
- `-XX:+PrintCommandLineFlags` is used to print all the VM command-line flags. See 7.3 Collecting Data for a Bug Report.
- `-XX:+PrintConcurrentLocks` will cause the Ctrl-Break handler to print a list of concurrent locks owned by each thread.
- `-XX:+PrintClassHistogram` will cause the Ctrl-Break handler to print a heap histogram.
- `-XX:+PrintGCDetails` and `-XX:+PrintGCTimeStamps` are used to print detailed information about garbage collection. See B.2.3 `-verbose:gc` Option.
- `-XX:+UseAltSigs` is used (on Solaris 8 and 9 OS) to instruct the HotSpot VM to use alternate signals to `SIGUSR1` and `SIGUSR2`. See 6.1 Signal Handling on Solaris OS and Linux.
- `-XX:+UseConcMarkSweepGC`, `-XX:+UseSerialGC`, and `-XX:+UseParallelGC` specify the garbage collection policy to be used. See 4.2.2 Crash During Garbage Collection.

B.2 Other Command-Line Options

In addition to the `-XX` options, many other command-line options can provide troubleshooting information. This section describes a few of these options.

B.2.1 `-Xcheck:jni` Option

The `-Xcheck:jni` option is useful in diagnosing problems with applications that use the Java Native Interface (JNI). Sometimes bugs in the native code can cause the HotSpot VM to crash or behave incorrectly.

The `-Xcheck:jni` option is added to the command line that starts the application, as in the following example.

```
java -Xcheck:jni MyApplication
```

The `-Xcheck:jni` option causes the VM to do additional validation on the arguments passed to JNI functions. Note that the option is not guaranteed to find all invalid arguments or diagnose logic bugs in the application code, but it can help diagnose a large number of such problems.

When an invalid argument is detected, the VM prints a message to the application console or to standard output, prints the stack trace of the offending thread, and aborts the VM.

In the following example, a NULL value was incorrectly passed to a JNI function that does not allow a NULL value.

```
FATAL ERROR in native method: Null object passed to JNI
    at java.net.PlainSocketImpl.socketAccept(Native Method)
    at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:343)
    - locked <0x450b9f70> (a java.net.PlainSocketImpl)
    at java.net.ServerSocket.implAccept(ServerSocket.java:439)
    at java.net.ServerSocket.accept(ServerSocket.java:410)
    at org.apache.tomcat.service.PoolTcpEndpoint.acceptSocket
                        (PoolTcpEndpoint.java:286)
    at org.apache.tomcat.service.TcpWorkerThread.runIt
                        (PoolTcpEndpoint.java:402)
    at org.apache.tomcat.util.ThreadPool$ControlRunnable.run
                        (ThreadPool.java:498)
    at java.lang.Thread.run(Thread.java:536)
```

In the following example, an incorrect argument was provided to a JNI function that expects a `jfieldID` argument.

```
FATAL ERROR in native method: Instance field not found in JNI get/set
                        field operations
        at java.net.PlainSocketImpl.socketBind(Native Method)
        at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:359)
        - locked <0xf082f290> (a java.net.PlainSocketImpl)
        at java.net.ServerSocket.bind(ServerSocket.java:318)
        at java.net.ServerSocket.<init>(ServerSocket.java:185)
        at jvm003a.<init>(jvm003.java:190)
        at jvm003a.<init>(jvm003.java:151)
        at jvm003.run(jvm003.java:51)
        at jvm003.main(jvm003.java:30)
```

The following list presents examples of other problems that the `-Xcheck:jni` option can help diagnose.

- Cases where the JNI environment for the wrong thread is used
- Cases where an invalid JNI reference is used
- Cases where a reference to a non-array type is provided to a function that requires an array type
- Cases where a non-static field ID is provided to a function that expects a static field ID
- Cases where a JNI call is made with an exception pending

In general, all errors detected by the `-Xcheck:jni` option are fatal errors, that is, the error is printed and the VM is aborted. There is one exception to this behavior. When a JNI call is made within a JNI critical region, the following non-fatal warning message is printed.

```
Warning: Calling other JNI functions in the scope of
Get/ReleasePrimitiveArrayCritical or Get/ReleaseStringCritical
```

A JNI critical region is created when native code uses the JNI functions `GetPrimitiveArrayCritical` or `GetStringCritical` to obtain a reference to an array or string in the Java heap. The reference is held until the native code calls the corresponding release function. The code between the get and release is called a JNI critical section and during that time the HotSpot VM cannot bring the VM to a state that allows garbage collection to occur. The general recommendation is not to use other JNI functions within a JNI critical section, and in particular any JNI function that could potentially cause a deadlock. The warning printed above by the `-Xcheck:jni` option is thus an indication of a potential issue; it does not always indicate an application bug.

For more information on JNI, refer to the Java Native Interface documentation web site.

B.2.2 `-verbose:class` Option

The `-verbose:class` option enables logging of class loading and unloading.

B.2.3 `-verbose:gc` Option

The `-verbose:gc` option enables logging of garbage collection (GC) information. It can be combined with other HotSpot VM specific options such as `-XX:+PrintGCDetails` and `-XX:+PrintGCTimeStamps` to get further information about the GC. The information output includes the size of the generations before and after each GC, total size of the heap, the size of objects promoted, and the time taken.

These options, together with detailed information about GC analysis and tuning, are described at the GC Portal site.

The `-verbose:gc` option can be dynamically enabled at runtime using the management API or JVM TI. See section 2.17 Developing Diagnostic Tools for further information on these APIs.

The `jconsole` monitoring and management tool can also enable or disable the option when the tool is attached to a management VM. See 2.3 JConsole Utility.

B.2.4 `-verbose:jni` Option

The `-verbose:jni` option enables logging of JNI. When a JNI or native method is resolved, the HotSpot VM prints a trace message to the application console (standard output). It also prints a trace message when a native method is registered using the JNI `RegisterNative` function. The `-verbose:jni` option can be useful in diagnosing issues with applications that use native libraries.

Fatal Error Log

When a fatal error occurs, an error log is created with information and the state obtained at the time of the fatal error.

Note that the format of this file can change slightly in update releases.

This appendix contains the following sections.

C.1 Location of Fatal Error Log

The product flag `-XX:ErrorFile=`*file* can be used to specify where the file will be created, where *file* represents the full path for the file location. The substring %% in the *file*variable is converted to %, and the substring %p is converted to the process ID of the process.

In the following example, the error log file will be written to the directory `/var/log/java` and will be named `java_error`*pid*`.log`.

```
java -XX:ErrorFile=/var/log/java/java_error%p.log
```

If the `-XX:ErrorFile=`*file* flag is not specified, by default the file name is `hs_err_pid`*pid*`.log`, where *pid* is the process ID of the process.

In addition, if the `-XX:ErrorFile=`*file* flag is not specified, the system attempts to create the file in the working directory of the process. In the event that the file cannot be created in the working directory (insufficient space, permission problem, or other issue), the file is created in the temporary directory for the operating system. On Solaris OS and Linux the temporary directory is `/tmp`. On Windows the temporary directory is specified by the value of the `TMP` environment variable; if that environment variable is not defined, the value of the `TEMP` environment variable is used.

C.2 Description of Fatal Error Log

The error log contains information obtained at the time of the fatal error, including the following information, where possible.

- The operating exception or signal that provoked the fatal error
- Version and configuration information
- Details on the thread that provoked the fatal error and thread's stack trace
- The list of running threads and their state
- Summary information about the heap
- The list of native libraries loaded
- Command line arguments
- Environment variables
- Details about the operating system and CPU

NOTE - IN SOME CASES ONLY A SUBSET OF THIS INFORMATION IS OUTPUT TO THE ERROR LOG. THIS CAN HAPPEN WHEN A FATAL ERROR IS OF SUCH SEVERITY THAT THE ERROR HANDLER IS UNABLE TO RECOVER AND REPORT ALL DETAILS.

The error log is a text file consisting of the following sections:

- A header that provides a brief description of the crash. See C.3 Header Format.
- A section with thread information. See C.4 Thread Section Format.
- A section with process information. See C.5 Process Section Format.
- A section with system information. See C.6 System Section Format.

NOTE - NOTE THAT THE FORMAT OF THE FATAL ERROR LOG DESCRIBED HERE IS BASED ON JDK 7. THE FORMAT MIGHT BE DIFFERENT WITH OTHER RELEASES.

C.3 Header Format

The header section at the beginning of every fatal error log file contains a brief description of the problem. The header is also printed to standard output and may show up in the application's output log.

The header includes a link to the HotSpot Virtual Machine Error Reporting Page, where the user can submit a bug report.

The following is a sample header from a crash.

```
#
# An unexpected error has been detected by Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x417789d7, pid=21139, tid=1024
#
# Java VM: Java HotSpot(TM) Client VM (1.6.0-rc-b63 mixed mode, sharing)
# Problematic frame:
# C  [libNativeSEGV.so+0x9d7]

#
# If you would like to submit a bug report, please visit:
#   http://java.sun.com/webapps/bugreport/crash.jsp
#
```

This example shows that the VM crashed on an unexpected signal. The next line describes the signal type, program counter (pc) that caused the signal, process ID and thread ID, as follows.

```
#  SIGSEGV (0xb) at pc=0x417789d7, pid=21139, tid=1024
      |      |           |             |         +--- thread id
      |      |           |             +------------- process id
      |      |           +--------------------------- program counter
      |      |                                        (instruction pointer)
      |      +---------------------------------------- signal number
      +----------------------------------------------- signal name
```

The next line contains the VM version (Client VM or Server VM), an indication whether the application was run in mixed or interpreted mode, and an indication whether class file sharing was enabled.

```
# Java VM: Java HotSpot(TM) Client VM (1.6.0-rc-b63 mixed mode, sharing)
```

The next information is the function frame that caused the crash, as follows.

```
# Problematic frame:
# C  [libNativeSEGV.so+0x9d7]
  |              +-- Same as pc, but represented as library name and offset.
  |                  For position-independent libraries (JVM and most shared
  |                  libraries), it is possible to inspect the instructions
  |                  that caused the crash without a debugger or core file
  |                  by using a disassembler to dump instructions near the
  |                  offset.
```

```
  +----------------- Frame type
```

In this example, the “C” frame type indicates a native C frame. The following table shows the possible frame types.

Thread Types

Frame Type	Description
C	Native C frame
j	Interpreted Java frame
V	VM frame
v	VM generated stub frame
J	Other frame types, including compiled Java frames

Internal errors will cause the VM error handler to generate a similar error dump. However, the header format is different. Examples of internal errors are `guarantee()` failure, assertion failure, `ShouldNotReachHere()`, and so forth. Here is an example of how the header looks for an internal error.

```
#
# An unexpected error has been detected by HotSpot Virtual Machine:
#
# Internal Error (4F533F4C494E55583F491418160E43505000F5), pid=10226, tid=16384
#
# Java VM: Java HotSpot(TM) Client VM (1.6.0-rc-b63 mixed mode)
```

In the above header, there is no signal name or signal number. Instead the second line now contains the text “Internal Error” and a long hexadecimal string. This hexadecimal string encodes the source module and line number where the error was detected. In general this “error string” is useful only to engineers working on the HotSpot Virtual Machine.

The error string encodes a line number and therefore it changes with each code change and release. A crash with a given error string in one release (for example 1.6.0) might not correspond to the same crash in an update release (for example 1.6.0_01), even if the strings match.

NOTE - DO NOT ASSUME THAT A WORKAROUND OR SOLUTION THAT WORKED IN ONE SITUATION ASSOCIATED WITH A GIVEN ERROR STRING WILL WORK IN ANOTHER SITUATION ASSOCIATED WITH THAT SAME ERROR STRING. NOTE THE FOLLOWING FACTS:

- **ERRORS WITH THE SAME ROOT CAUSE MIGHT HAVE DIFFERENT ERROR STRINGS.**
- **ERRORS WITH THE SAME ERROR STRING MIGHT HAVE COMPLETELY DIFFERENT ROOT CAUSES.**

THEREFORE, THE ERROR STRING SHOULD NOT BE USED AS THE SOLE CRITERION WHEN TROUBLESHOOTING BUGS.

C.4 Thread Section Format

This section contains information about the thread that just crashed. If multiple threads crash at the same time, only one thread is printed.

C.4.1 Thread Information

The first part of the thread section shows the thread that provoked the fatal error, as follows.

```
Current thread (0x0805ac88):  JavaThread "main" [_thread_in_native, id=21139]
                    |             |         |            |          +-- ID
                    |             |         |            +------------- state
                    |             |         +-------------------------- name
                    |             +-------------------------------------- type
                    +---------------------------------------------------- pointer
```

The thread pointer is the pointer to the Java VM internal thread structure. It is generally of no interest unless you are debugging a live Java VM or core file.

The following list shows possible thread types.

- `JavaThread`
- `VMThread`
- `CompilerThread`
- `GCTaskThread`
- `WatcherThread`
- `ConcurrentMarkSweepThread`

The following table shows the important thread states.

Thread States

Thread State	Description
`_thread_uninitialized`	Thread is not created. This occurs only in the case of memory corruption.
`_thread_new`	Thread has been created but it has not yet started.
`_thread_in_native`	Thread is running native code. The error is probably a bug in native code.
`_thread_in_vm`	Thread is running VM code.
`_thread_in_Java`	Thread is running either interpreted or compiled Java code.
`_thread_blocked`	Thread is blocked.
`..._trans`	If any of the above states is followed by the string `_trans`, that means that the thread is changing to a different state.

The thread ID in the output is the native thread identifier.

If a Java thread is a daemon thread, then the string `daemon` is printed before the thread state.

C.4.2 Signal Information

The next information in the error log describes the unexpected signal that caused the VM to terminate. On a Windows system the output appears as follows.

```
siginfo: ExceptionCode=0xc0000005, reading address 0xd8ffecf1
```

In the above example, the exception code is `0xc0000005` (ACCESS_VIOLATION), and the exception occurred when the thread attempted to read address `0xd8ffecf1`.

On Solaris OS and Linux systems the signal number (`si_signo`) and signal code (`si_code`) are used to identify the exception, as follows.

```
siginfo:si_signo=11, si_errno=0, si_code=1, si_addr=0x00004321
```

C.4.3 Register Context

The next information in the error log shows the register context at the time of the fatal error. The exact format of this output is processor-dependent. The following example shows output for the Intel (IA32) processor.

```
Registers:
EAX=0x00004321, EBX=0x41779dc0, ECX=0x080b8d28, EDX=0x00000000
ESP=0xbfffc1e0, EBP=0xbfffc1f8, ESI=0x4a6b9278, EDI=0x0805ac88
EIP=0x417789d7, CR2=0x00004321, EFLAGS=0x00010216
```

The register values might be useful when combined with instructions, as described below.

C.4.4 Machine Instructions

After the register values, the error log contains the top of stack followed by 32 bytes of instructions (opcodes) near the program counter (PC) when the system crashed. These opcodes can be decoded with a disassembler to produce the instructions around the location of the crash. Note that IA32 and AMD64 instructions are variable in length, and so it is not always possible to reliably decode instructions before the crash PC.

```
Top of Stack: (sp=0xbfffc1e0)
0xbfffc1e0:   00000000 00000000 0818d068 00000000
0xbfffc1f0:   00000044 4a6b9278 bfffd208 41778a10
0xbfffc200:   00004321 00000000 00000cd8 0818d328
0xbfffc210:   00000000 00000000 00000004 00000003
0xbfffc220:   00000000 4000c78c 00000004 00000000
0xbfffc230:   00000000 00000000 00180003 00000000
0xbfffc240:   42010322 417786ec 00000000 00000000
0xbfffc250:   4177864c 40045250 400131e8 00000000
Instructions: (pc=0x417789d7)
0x417789c7:   ec 14 e8 72 ff ff ff 81 c3 f2 13 00 00 8b 45 08
0x417789d7:   0f b6 00 88 45 fb 8d 83 6f ee ff ff 89 04 24 e8
```

C.4.5 Thread Stack

Where possible, the next output in the error log is the thread stack. This includes the addresses of the base and the top of the stack, the current stack pointer, and the amount of unused stack available to the thread. This is followed, where possible, by the stack frames, and up to 100 frames are printed. For C/C++ frames the library name may also be printed. It is important to note that in some fatal error conditions the stack may be corrupt, and in this case this detail may not be available.

```
Stack: [0x00040000,0x00080000),  sp=0x0007f9f8,  free space=254k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [jvm.dll+0x83d77]
C  [App.dll+0x1047]
j  Test.foo()V+0
j  Test.main([Ljava/lang/String;)V+0
v  ~StubRoutines::call_stub
V  [jvm.dll+0x80f13]
V  [jvm.dll+0xd3842]
V  [jvm.dll+0x80de4]
C  [java.exe+0x14c0]
C  [java.exe+0x64cd]
C  [kernel32.dll+0x214c7]

Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
j  Test.foo()V+0
j  Test.main([Ljava/lang/String;)V+0
v  ~StubRoutines::call_stub
```

The log contains two thread stacks.

- The first thread stack is **Native frames**, which prints the native thread showing all function calls. However this thread stack does not take into account the Java methods that are inlined by the runtime compiler; if methods are inlined they appear to be part of the parent's stack frame.

  The information in the thread stack for native frames provides important information about the cause of the crash. By analyzing the libraries in the list from the top down, you can generally determine which library might have caused the problem and report it to the appropriate organization responsible for that library.

- The second thread stack is **Java frames**, which prints the Java frames including the inlined methods, skipping the native frames. Depending on the crash it might not be possible to print the native thread stack but it might be possible to print the Java frames.

C.4.6 Further Details

If the error occurred in the VM thread or in a compiler thread, then further details may be printed. For example, in the case of the VM thread, the VM operation is printed if the VM thread is executing a VM operation at the time of the fatal error. In the following output example, the compiler thread provoked the fatal error. The task is a compiler task and the HotSpot Client VM is compiling method `hs101t004Thread.ackermann`.

```
Current CompileTask:
HotSpot Client Compiler:754   b
nsk.jvmti.scenarios.hotswap.HS101.hs101t004Thread.ackermann(IJ)J (42 bytes)
```

For the HotSpot Server VM the output for the compiler task is slightly different but will also include the full class name and method.

C.5 Process Section Format

The process section is printed after the thread section. It contains information about the whole process, including thread list and memory usage of the process.

C.5.1 Thread List

The thread list includes the threads that the VM is aware of. This includes all Java threads and some VM internal threads, but does not include any native threads created by the user application that have not attached to the VM. The output format follows.

```
=>0x0805ac88 JavaThread "main" [_thread_in_native, id=21139]
|      |         |        |             |             +----- ID
|      |         |        |             +------------------- state
|      |         |        |                                  (JavaThread only)
|      |         |        +----------------------------------- name
|      |         +-------------------------------------------- type
|      +------------------------------------------------------ pointer
+------------------------------------------------------------- "=>" current thread
```

An example of this output follows.

```
Java Threads: ( => current thread )
  0x080c8da0 JavaThread "Low Memory Detector" daemon [_thread_blocked, id=21147]
  0x080c7988 JavaThread "CompilerThread0" daemon [_thread_blocked, id=21146]
  0x080c6a48 JavaThread "Signal Dispatcher" daemon [_thread_blocked, id=21145]
  0x080bb5f8 JavaThread "Finalizer" daemon [_thread_blocked, id=21144]
  0x080ba940 JavaThread "Reference Handler" daemon [_thread_blocked, id=21143]
=>0x0805ac88 JavaThread "main" [_thread_in_native, id=21139]

Other Threads:
  0x080b6070 VMThread [id=21142]
  0x080ca088 WatcherThread [id=21148]
```

The thread type and thread state are described in C.4 Thread Section Format.

C.5.2 VM State

The next information is the VM state, which indicates the overall state of the virtual machine. The following table describes the general states.

VM States

General VM State	Description
`not at a safepoint`	Normal execution.
`at safepoint`	All threads are blocked in the VM waiting for a special VM operation to complete.
`synchronizing`	A special VM operation is required and the VM is waiting for all threads in the VM to block.

The VM state output is a single line in the error log, as follows.

```
VM state:not at safepoint (normal execution)
```

C.5.3 Mutexes and Monitors

The next information in the error log is a list of mutexes and monitors that are currently owned by a thread. These mutexes are VM internal locks rather than monitors associated with Java objects. Below is an example to show how the output might look when a crash happens when VM locks are held. For each lock the log contains the name of the lock, its owner, and the addresses of a VM internal mutex structure and its OS lock. In general this information is useful only to those who are very familiar with the HotSpot VM. The owner thread can be cross-referenced to the thread list.

```
VM Mutex/Monitor currently owned by a thread:
([mutex/lock_event])[0x007357b0/0x0000031c] Threads_lock - owner thread: 0x00996318
[0x00735978/0x000002e0] Heap_lock - owner thread: 0x00736218
```

C.5.4 Heap Summary

The next information is a summary of the heap. The output depends on the garbage collection (GC) configuration. In this example the serial collector is used, class data sharing is disabled, and the tenured generation is empty. This probably indicates that the fatal error occurred early or during start-up and a GC has not yet promoted any objects into the tenured generation. An example of this output follows.

```
Heap
 def new generation   total 576K, used 161K [0x46570000, 0x46610000, 0x46a50000)
  eden space 512K,  31% used [0x46570000, 0x46598768, 0x465f0000)
  from space 64K,   0% used [0x465f0000, 0x465f0000, 0x46600000)
  to   space 64K,   0% used [0x46600000, 0x46600000, 0x46610000)
 tenured generation   total 1408K, used 0K [0x46a50000, 0x46bb0000, 0x4a570000)
   the space 1408K,   0% used [0x46a50000, 0x46a50000, 0x46a50200, 0x46bb0000)
 compacting perm gen  total 8192K, used 1319K [0x4a570000, 0x4ad70000, 0x4e570000)
   the space 8192K,  16% used [0x4a570000, 0x4a6b9d48, 0x4a6b9e00, 0x4ad70000)
No shared spaces configured.
```

C.5.5 Memory Map

The next information in the log is a list of virtual memory regions at the time of the crash. This list can be long in the case of large applications. The memory map can be very useful when debugging some crashes, as it can tell you what libraries are actually being used, their location in memory, as well as the location of heap, stack, and guard pages.

The format of the memory map is operating-system-specific. On the Solaris Operating System, the base address and library name are printed. On the Linux system the process memory map (`/proc/`*pid*`/maps`) is printed. On the Windows system, the base and end addresses of each library are printed. The following example output was generated on Linux/x86. Note that most of the lines have been omitted from the example for the sake of brevity.

```
Dynamic libraries:
08048000-08056000 r-xp 00000000 03:05 259171     /h/jdk6/bin/java
08056000-08058000 rw-p 0000d000 03:05 259171     /h/jdk6/bin/java
08058000-0818e000 rwxp 00000000 00:00 0
40000000-40013000 r-xp 00000000 03:0a 400046     /lib/ld-2.2.5.so
40013000-40014000 rw-p 00013000 03:0a 400046     /lib/ld-2.2.5.so
40014000-40015000 r--p 00000000 00:00 0
Lines omitted.
4123d000-4125a000 rwxp 00001000 00:00 0
4125a000-4125f000 rwxp 00000000 00:00 0
4125f000-4127b000 rwxp 00023000 00:00 0
4127b000-4127e000 ---p 00003000 00:00 0
4127e000-412fb000 rwxp 00006000 00:00 0
412fb000-412fe000 ---p 00083000 00:00 0
412fe000-4137b000 rwxp 00086000 00:00 0
Lines omitted.
44600000-46570000 rwxp 00090000 00:00 0
46570000-46610000 rwxp 00000000 00:00 0
46610000-46a50000 rwxp 020a0000 00:00 0
46a50000-46bb0000 rwxp 00000000 00:00 0
46bb0000-4a570000 rwxp 02640000 00:00 0
Lines omitted.
```

The format of a line in the above memory map is as follows.

```
40049000-4035c000 r-xp 00000000 03:05 824473  /jdk1.5/jre/lib/i386/client/libjvm.so
|<------------>|  ^      ^       ^     ^        |<----------------------------------->|
  Memory region    |      |       |     |                        |
                   |      |       |     |                        |
  Permission   --- +      |       |     |                        |
    r: read               |       |     |                        |
    w: write              |       |     |                        |
    x: execute            |       |     |                        |
    p: private            |       |     |                        |
    s: share              |       |     |                        |
                          |       |     |                        |
  File offset   ----------+       |     |                        |
                                  |     |                        |
  Major ID and minor ID of -------+     |                        |
  the device where the file             |                        |
  is located (i.e. /dev/hda5)           |                        |
                                        |                        |
  inode number  ------------------------+                        |
                                                                 |
  File name  ----------------------------------------------------+
```

In the memory map output, each library has two virtual memory regions: one for code and one for data. The permission for the code segment is marked with `r-xp` (readable, executable, private), and the permission for the data segment is `rw-p` (readable, writable, private).

The Java heap is already included in the heap summary earlier in the output, but it can be useful to verify that the actual memory regions reserved for heap match the values in the heap summary and that the attributes are set to `rwxp`.

Thread stacks usually show up in the memory map as two back-to-back regions, one with permission `---p` (guard page) and one with permission `rwxp` (actual stack space). In addition, it is useful to know the guard page size or stack size. For example, in this memory map, the stack is located from `4127b000` to `412fb000`.

On a Windows system, the memory map output is the load and end address of each loaded module, as in the example below.

```
Dynamic libraries:
0x00400000 - 0x0040c000     c:\jdk6\bin\java.exe
0x77f50000 - 0x77ff7000     C:\WINDOWS\System32\ntdll.dll
0x77e60000 - 0x77f46000     C:\WINDOWS\system32\kernel32.dll
0x77dd0000 - 0x77e5d000     C:\WINDOWS\system32\ADVAPI32.dll
0x78000000 - 0x78087000     C:\WINDOWS\system32\RPCRT4.dll
0x77c10000 - 0x77c63000     C:\WINDOWS\system32\MSVCRT.dll
```

```
0x08000000 - 0x08183000     c:\jdk6\jre\bin\client\jvm.dll
0x77d40000 - 0x77dcc000     C:\WINDOWS\system32\USER32.dll
0x7e090000 - 0x7e0d1000     C:\WINDOWS\system32\GDI32.dll
0x76b40000 - 0x76b6c000     C:\WINDOWS\System32\WINMM.dll
0x6d2f0000 - 0x6d2f8000     c:\jdk6\jre\bin\hpi.dll
0x76bf0000 - 0x76bfb000     C:\WINDOWS\System32\PSAPI.DLL
0x6d680000 - 0x6d68c000     c:\jdk6\jre\bin\verify.dll
0x6d370000 - 0x6d38d000     c:\jdk6\jre\bin\java.dll
0x6d6a0000 - 0x6d6af000     c:\jdk6\jre\bin\zip.dll
0x10000000 - 0x10032000     C:\bugs\crash2\App.dll
```

C.5.6 VM Arguments and Environment Variables

The next information in the error log is a list of VM arguments, followed by a list of environment variables. An example follows.

```
VM Arguments:
java_command: NativeSEGV 2

Environment Variables:
JAVA_HOME=/h/jdk
PATH=/h/jdk/bin:.:/h/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:
     /usr/dist/local/exe:/usr/dist/exe:/bin:/usr/sbin:/usr/ccs/bin:
     /usr/ucb:/usr/bsd:/usr/etc:/etc:/usr/dt/bin:/usr/openwin/bin:
     /usr/sbin:/sbin:/h:/net/prt-web/prt/bin
USERNAME=user
LD_LIBRARY_PATH=/h/jdk6/jre/lib/i386/client:/h/jdk6/jre/lib/i386:
     /h/jdk6/jre/../lib/i386:/h/bugs/NativeSEGV
SHELL=/bin/tcsh
DISPLAY=:0.0
HOSTTYPE=i386-linux
OSTYPE=linux
ARCH=Linux
MACHTYPE=i386
```

Note that the list of environment variables is not the full list but rather a subset of the environment variables that are applicable to the Java VM.

C.5.7 Signal Handlers

On Solaris OS and Linux, the next information in the error log is the list of signal handlers.

```
Signal Handlers:
SIGSEGV: [libjvm.so+0x3aea90], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGBUS: [libjvm.so+0x3aea90], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGFPE: [libjvm.so+0x304e70], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGPIPE: [libjvm.so+0x304e70], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGILL: [libjvm.so+0x304e70], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGUSR1: SIG_DFL, sa_mask[0]=0x00000000, sa_flags=0x00000000
SIGUSR2: [libjvm.so+0x306e80], sa_mask[0]=0x80000000, sa_flags=0x10000004
SIGHUP: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGINT: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGQUIT: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGTERM: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGUSR2: [libjvm.so+0x306e80], sa_mask[0]=0x80000000, sa_flags=0x10000004
```

C.6 System Section Format

The final section in the error log is the system information. The output is operating-system-specific but in general includes the operating system version, CPU information, and summary information about the memory configuration.

The following example shows output on a Solaris 9 OS system.

```
---------------  S Y S T E M  ---------------
```

```
OS:                       Solaris 9 12/05 s9s_u5wos_08b SPARC
          Copyright 2005 Sun Microsystems, Inc.  All Rights Reserved.
                      Use is subject to license terms.
                         Assembled 21 November 2005

uname:SunOS 5.9 Generic_112233-10 sun4u  (T2 libthread)
rlimit: STACK 8192k, CORE infinity, NOFILE 65536, AS infinity
load average:0.41 0.14 0.09

CPU:total 2 has_v8, has_v9, has_vis1, has_vis2, is_ultra3

Memory: 8k page, physical 2097152k(1394472k free)

vm_info: Java HotSpot(TM) Client VM (1.5-internal) for solaris-sparc,
built on Aug 12 2005 10:22:32 by unknown with unknown Workshop:0x550
```

On Solaris OS and Linux, the operating system information is contained in the file `/etc/*release`. This file describes the kind of system the application is running on, and in some cases the information string might include the patch level. Some system upgrades are not reflected in the `/etc/*release` file. This is especially true on the Linux system, where the user can rebuild any part of the system.

On Solaris OS the `uname` system call is used to get the name for the kernel. The thread library (T1 or T2) is also printed.

On the Linux system the `uname` system call is also used to get the kernel name. The `libc` version and the thread library type are also printed. An example follows.

```
uname:Linux 2.4.18-3smp #1 SMP Thu Apr 18 07:27:31 EDT 2002 i686
libc:glibc 2.2.5 stable linuxthreads (floating stack)
    |<- glibc version ->|<--  pthread type       -->|
```

On Linux there are three possible thread types, namely `linuxthreads (fixed stack)`, `linuxthreads (floating stack)`, and `NPTL`. They are normally installed in`/lib`, `/lib/i686`, and `/lib/tls`.

It is useful to know the thread type. For example, if the crash appears to be related to `pthread`, then you might be able to work around an issue by selecting a different`pthread` library. A different `pthread` library (and `libc`) can be selected by setting `LD_LIBRARY_PATH` or `LD_ASSUME_KERNEL`.

The `glibc` version usually does not include the patch level. The command `rpm -q glibc` might provide more detailed version information.

On Solaris OS and Linux, the next information is the `rlimit` information. Note that the default stack size of the VM is usually smaller than the system limit. An example follows.

```
rlimit: STACK 8192k, CORE 0k, NPROC 4092, NOFILE 1024, AS infinity
             |          |         |           |           virtual memory (-v)
             |          |         |           +--- max open files (ulimit -n)
             |          |         +------------ max user processes (ulimit -u)
             |          +-------------------------- core dump size (ulimit -c)
             +---------------------------------------- stack size (ulimit -s)
load average:0.04 0.05 0.02
```

The next information specifies the CPU architecture and capabilities identified by the VM at start-up, as in the following example.

CPU:total 2 family 6, cmov, cx8, fxsr, mmx, sse | | |<----- CPU features ---->| | | | +--- processor family (IA32 only): | 3 - i386 | 4 - i486 | 5 - Pentium | 6 - PentiumPro, PII, PIII | 15 - Pentium 4 +------------ Total number of CPUs

The following table shows the possible CPU features on a SPARC system.

SPARC Features

SPARC Feature	Description
has_v8	Supports v8 instructions.
has_v9	Supports v9 instructions.
has_vis1	Supports visualization instructions.
has_vis2	Supports visualization instructions.
is_ultra3	UltraSparc III.
no-muldiv	No hardware integer multiply and divide.
no-fsmuld	No multiply-add and multiply-subtract instructions.

The following table shows the possible CPU features on an Intel/IA32 system.

Intel/IA32 Features

Intel/IA32 Feature	Description
cmov	Supports cmov instruction.
cx8	Supports cmpxchg8b instruction.
fxsr	Supports fxsave and fxrstor.
mmx	Supports MMX.
sse	Supports SSE extensions.
sse2	Supports SSE2 extensions.
ht	Supports Hyper-Threading Technology.

The following table shows the possible CPU features on an AMD64/EM64T system.

AMD64/EM64T Features

AMD64/EM64T Feature	Description
amd64	AMD Opteron, Athlon64, and so forth.
em64t	Intel EM64T processor.
3dnow	Supports 3DNow extension.
ht	Supports Hyper-Threading Technology.

The next information in the error log is memory information, as follows.

```
                                        unused swap space
                 total amount of swap space      |
unused physical memory         |          |
```

```
  total amount of physical memory     |              |          |
       page size            |         |              |          |
           v                v         v              v          v
Memory: 4k page, physical 513604k(11228k free), swap 530104k(497504k free)
```

Some systems require swap space to be at lease twice the size of real physical memory, whereas other systems do not have any such requirements. As a general rule, if both physical memory and swap space are almost full, there is good reason to suspect that the crash was due to insufficient memory.

On Linux systems the kernel may convert most of unused physical memory to file cache. When there is a need for more memory, the Linux kernel will give the cache memory back to the application. This is handled transparently by the kernel, but it does mean the amount of unused physical memory reported by fatal error handler could be close to zero when there is still sufficient physical memory available.

The final information in the SYSTEM section of the error log is `vm_info`, which is a version string embedded in `libjvm.so/jvm.dll`. Every Java VM has its own unique`vm_info` string. If you are in doubt about whether the fatal error log was generated by a particular Java VM, check the version string.

Summary of Tools in This Release

This appendix presents a summary of tools available in the current release of the JDK, as well as the changes since the previous release.

D.1 Troubleshooting Tools Available in JDK 7

This section lists the troubleshooting tools available by platform: Unix (Solaris OS and Linux) and Windows.

D.1.1 Solaris OS and Linux

All the JDK troubleshooting tools that are described in this document are available in JDK 7 on both Solaris OS and Linux.

D.1.2 Windows Operating System

The following JDK troubleshooting tools are also available in JDK 7 on the Windows operating system.

- HPROF profiler
- JConsole utility
- `jdb` utility
- `jhat` utility
- `jinfo` utility
- `jmap` utility
- `jps` utility (not currently available on Windows 98 or Windows ME)
- `jrunscript` utility
- `jstack` utility
- `jstat` utility (not currently available on Windows 98 or Windows ME)
- `jstatd` daemon (not currently available on Windows 98 or Windows ME)
- `visualgc` tool (not currently available on Windows 98 or Windows ME)

D.2 Changes to Troubleshooting Tools in JDK 7

This is a list of changes to the JDK troubleshooting tools and options from JDK 1.5 to JDK 7.

- The Java VisualVM tool is included in JDK releases starting with release 6 update 7.
- The Heap Analysis Tool (HAT) has been replaced by the new `jhat` command–line tool. This new tool has the same functionality as HAT, with the following additional enhancements:
  - `jhat` can parse incomplete and truncated heap dumps.
  - `jhat` can read heap dumps generated on 64–bit systems.
  - `jhat` supports Object Query Language (OQL), with which you can create your own queries on the heap dump.
- The JConsole tool has the following changes.
  - A new user interface (button) is provided for deadlock detection, including `java.util.concurrent` locks.
  - The Connection dialog is new.
  - The Overview tab is new.
  - You no longer need to specify the `-Dcom.sun.management.jmxremote` command-line option when starting the application to be monitored.
  - You can connect to a VM using the attach mechanism.
  - You can pass a flag on the command line to the VM that is running JConsole.
  - The `com.sun.management.HotSpotDiagnostic` MBean property is new. In the MBean tab, you can use this property to dump heap, get and set VM options, and change management options dynamically.
- The `jdb` tool has the following changes:
  - Shows return values in method and exit traces.
  - Trace method entry/exit without stopping.
  - The JPDA `ProcessAttachingConnector` is new.
- The `jinfo` tool has the following changes:
  - The new `jinfo -flag` option allows you to dynamically set, unset, and change the values of certain Java VM flags for a specified Java process.
  - The `jinfo` command is new to Windows, but only the option `jinfo -flag` *pid*.
- The `jmap` command has the following changes:

- The `jmap -finalizerinfo` option is new. With this option, the command prints information on objects awaiting finalization.
- The `jmap -permstat` option has been updated to print also the number and size of internalized `String` instances.
- The `jmap -dump:format=b,file=`*filename* option is new. With this option, `jmap` obtains a heap dump from a running process or from a core file and writes it in binary HPROF format to a specified file. This file can then be analyzed with the `jhat` tool.
- The `jmap -F` option is new and is for Solaris OS and Linux only. This option forces the use of the Serviceability Agent in case the process does not respond.
- The `jmap -J` option is new and is for Solaris OS and Linux only. This option passes flags to the VM on which `jmap` is running.
- The `live` suboption is new for the `histo` option. With this suboption only live objects are counted.
- The `jmap` command is new to Windows, but only the `jmap -dump:`*dump-options pid* option and the `jmap -histo[:live]` *pid* option.

- The `jrunscript` tool is new.
- The `jstack` command has the following changes:
  - The `jstack` *pid* option has been changed to work like remote Ctrl-Break, that is, the output of the thread dump is slightly different, and there is more information on deadlocks and JNI Global References.
  - The `jstack -F` option is new and is for Solaris OS and Linux only. This option forces a thread dump in case the VM is hung.
  - The `jstack -l` option is new. This option prints information about ownable synchronizers (locks) in the heap.
  - The `jstack` command is new to Windows, but only the `jstack` *pid* option and the `jstack -l` *pid* option.
- The `-XX:OnOutOfMemoryError=`*string* option allows you to specify a command or script to be run when an `OutOfMemoryError` is first thrown. This is useful for data capture, for example, with `jmap`.
- The `-XX:+HeapDumpOnOutOfMemoryError` command-line option is new as of JDK release 5 update 7. If this option is set and the VM detects a native out–of–memory error, a heap dump is generated. See B.1.2 - `XX:+HeapDumpOnOutOfMemoryError` Option.
- Other new environment variables to help in troubleshooting out-of-memory errors are the following:
  - `-XX:HeapDumpPath=`*pathname*
  - `-XX:SegmentedHeapDumpThreshold=`*threshold*
  - `-XX:HeapDumpSegmentSize=`*size*
- The fatal error log has the following changes:
  - You now have the ability to specify a location for the fatal error log. See C.1 Location of Fatal Error Log.
  - There is now a URL at the beginning of fatal error log for reporting incidents. See C.3 Header Format.
- The VM now contains two built-in DTrace probe providers: `hotspot` and `hotspot_jni`. (Solaris 10 OS only)
- The `fastdebug` builds can be useful for testing, diagnosing, and isolating problems. However, they should not be used in production environments.
- The Ctrl-Break handler has the following changes:
  - Ctrl-Break now shows the thread state.
  - Ctrl-Break now shows the sizes of the heap areas.
  - With `-XX:+PrintConcurrentLocks`, Ctrl-Break will also print the list of concurrent locks owned by each thread, as well as detect deadlocks involving both monitor locks and concurrent locks.
- The `HPROF format=b` option now includes primitive type instance fields and primitive array content in heap dumps.
- When the `java.lang.OutOfMemoryError` error is thrown, a stack trace is now printed also. In addition, when the specific message indicates that the system is almost out of swap space (`Out of swap space?`), a fatal error log is now generated.