Java™ Troubleshooting Guide for HP-UX Systems

HP Part Number: 5991-7463
Published: February 2007
Edition: 2

Table of Contents

List of Figures

List of Tables

About This Document

The information in this document will help application developers and support engineers debug their Java applications on HP-UX systems.

Intended Audience

This document is intended for application developers and support engineers who are debugging Java applications on HP-UX systems.

New and Changed Information in This Edition

This is the second version of this document. It contains updated information about `HPjmeter`, new information about the jvmstat and `visualgc` tools, and corrections to the first version of this document.

Document Organization

This document contains three chapters:

Chapter 1: Diagnostic and Monitoring Tools and Options—This chapter provides information on tools and options useful for Java troubleshooting on HP-UX.

Chapter 2: Useful System Tools for Java Troubleshooting—This chapter provides information about HP-UX system tools to aide in Java troubleshooting.

Chapter 3: Getting Help from Hewlett-Packard—This chapter contains information about collecting necessary data before opening a Java-related support call.

Typographic Conventions

This document uses the following typographical conventions:

`%`, `$`, or `#`	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells. A number sign represents the superuser prompt.
audit(5)	A manpage. The manpage name is *audit*, and it is located in Section 5.
`Command`	A command name or qualified command phrase.
`Computer output`	Text displayed by the computer.
Ctrl+x	A key sequence. A sequence such as **Ctrl+x** indicates that you must hold down the key labeled **Ctrl** while you press another key or mouse button.
`ENVIRONMENT VARIABLE`	The name of an environment variable, for example, `PATH`.
[ERROR NAME]	The name of an error, usually returned in the `errno` variable.
Key	The name of a keyboard key. **Return** and **Enter** both refer to the same key.
Term	The defined use of an important word or phrase.
`User input`	Commands and other text that you type.
`Variable`	The name of a placeholder in a command, function, or other syntax display that you replace with an actual value.
[]	The contents are optional in syntax. If the contents are a list separated by |, you must choose one of the items.
{}	The contents are required in syntax. If the contents are a list separated by |, you must choose one of the items.

...	The previous element can be repeated an arbitrary number of times.
◎	Indicates the continuation of a code example.
|	Separates items in a list of choices.
WARNING	A warning calls attention to important information that if not understood or followed will result in personal injury or nonrecoverable system problems.
CAUTION	A caution calls attention to important information that if not understood or followed will result in data loss, data corruption, or damage to hardware or software.
IMPORTANT	This alert provides essential information to explain a concept or to complete a task.
NOTE	A note contains additional information to emphasize or supplement important points of the main text.

Related Information

This document contains information specific to troubleshooting Java problems on HP-UX systems.

More information can also be found in the HP-UX Programmer's Guide for Java™ 2. In addition, the *Trouble-Shooting and Diagnostic Guide for Java 2 Platform, Standard Edition 5.0* from Sun Microsystems also contains some information that may be useful.

Publishing History

The document printing date and part number indicate the document's current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The document part number will change when extensive changes are made. Document updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details. The latest version of this document is available online at:

http://www.docs.hp.com

Manufacturing Part Number	Supported Operating Systems	Supported Versions	Edition Number	Publication Date
5991-7463	*HP-UX 11i*	*Versions 1 and 2*	*Edition 2*	*February 2007*

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to:

feedback@fc.hp.com

Include the document title, manufacturing part number, and any comment, error found, or suggestion for improvement you have concerning this document.

1 Diagnostic and Monitoring Tools and Options

This chapter describes the tools and options available for postmortem diagnostics, analysis of hung/deadlocked processes, monitoring memory usage, and performance monitoring.

The tools and options are listed in tables by their respective functions in the first section of this chapter. Many of them are listed in multiple tables since they can be used for multiple functions.

The tools and options are described in detail with examples, where applicable, in the remaining sections of this chapter. All the tools and options described in this chapter are either included in the Java 2 Platform Standard Edition Development Kit (JDK 1.5), are included with Hewlett-Packard's Java product, or are available for download at the Go Java! website:

http://www.hp.com/products1/unix/java

1.1 HP-UX Java Tools and Options Tables

The tools and options are categorized into the following table groupings:

- Crash Analysis Tools
- Hung and Deadlocked Processes
- Fatal Error Handling
- Monitoring Memory Use
- Performance Monitoring Tools
- Miscellaneous Tools and Options
- JDK Tools Not Available on HP-UX

1.1.1 Crash Analysis Tools

Several of the options and tools described in this chapter are designed for postmortem diagnostics. These are the options and tools that can be used to obtain additional information if an application crashes. This analysis may either be done at the time of the crash or at a later time using information from the core file. In addition to these tools, many other tools have features useful for crash analysis.

Table 1-1 Tools and Options for Crash Analysis

Tool or Option	Description and Usage
wdb/gdb	An HP-supported implementation of the gdb debugger that has Java support. For simplicity, this document will refer to wdb/gdb as gdb from this point forward. gdb can be used to attach to a running process.
Fatal Error Log (hs_err_pid<pid>.log)	Contains information obtained at the time of the crash. Often one of the first pieces of data to examine when a crash occurs.
-XX:OnError	Specify a sequence of user-supplied scripts or commands to be executed when a crash occurs.
-XX:+ShowMessageBoxOnError	Suspend the process when a crash occurs. Depending on the user response, it can launch the gdbgdb debugger to attach to the Java VM.
jdb	Java language debugger.

1.1.2 Hung and Deadlocked Processes

The following options and tools can help you debug a hung or deadlocked process:

Table 1-2 Tools and Options for Debugging Hung and Deadlocked Processes

Tool or Option	Description and Usage
wdb/gdb	An HP-supported implementation of the gdb debugger that has Java support. For simplicity, this document refers to wdb/gdb as gdb from this point forward. gdb can be used to attach to a running process.
HPjmeter	Used to identify and diagnose performance problems in Java applications running on HP-UX. It can also be used to debug thread and heap issues.
Ctrl-Break Handler	Used to retrieve thread dump information. It also executes a deadlock detection algorithm and reports any deadlocks detected involving synchronized code. Heap dumps are also generated beginning with JDK 1.5.0.05 and SDK 1.4.2.11 when the -XX:+HeapDumpOnCtrlBreak option is specified.
-XX:+HeapDump and _JAVA_HEAPDUMP Environment Variable, starting with JDK 1.5.0.03 and SDK 1.4.2.10	Used to observe memory allocation in a running Java application by taking snapshots of the heap over time. It can be set by providing the `-XX:+HeapDump` option or setting the `_JAVA_HEAPDUMP` environment variable.
jdb	Java language debugger.

1.1.3 Fatal Error Handling

The following options are useful for retrieving more information when fatal errors occur:

Table 1-3 Options for Fatal Error Handling

Option	Description and Usage
-XX:OnError	Used to specify a sequence of user-supplied scripts or commands to be executed when a crash occurs.
-XX:+ShowMessageBoxOnError	Used to suspend the process when a crash occurs. After the process is suspended, the user can use gdb to attach to the Java VM.
-XX:+HeapDumpOnOutOfMemoryError, starting with SDK 1.4.2.11 and JDK 1.5.0.04	Enables dumping of the heap when an out of memory error condition occurs in the Java VM.

1.1.4 Monitoring Memory Use

The following options and tools are useful for monitoring memory usage of running applications:

Table 1-4 Tools and Options for Monitoring Memory Use

Tool	Description and Usage
HPjmeter	Used to identify and diagnose performance problems in Java applications by examining and monitoring the heap and threads.
HPjtune	HP's garbage collection (GC) visualization tool for analyzing garbage collection activity in a Java program.
-XX:+HeapDump and _JAVA_HEAPDUMP Environment Variable, starting with JDK 1.5.0.03 and SDK 1.4.2.10	Used to observe memory allocation in a running Java application by taking snapshots of the heap over time. It can be set by providing the `-XX:+HeapDump` option or setting the `_JAVA_HEAPDUMP` environment variable.
-Xverbosegc (HP only) and -verbose:gc	Used to enable logging of garbage collection information. The HP-only `-Xverbosegc` option generates additional GC information that is used by HPjtune. It is preferable to use `-Xverbosegc` instead of `-verbose:gc`.

Table 1-4 Tools and Options for Monitoring Memory Use *(continued)*

Tool	Description and Usage
hat	This third-party tool may be used to perform Java heap analysis.
jconsole (1.5 only)	Used to monitor and manage an application launched with a management agent on a local or remote machine.

1.1.5 Performance Monitoring Tools

The following tools are useful for monitoring system and application performance:

Table 1-5 Performance Monitoring Tools

Tool	Description and Usage
HPjmeter	Used to identify and diagnose performance problems in Java applications.
HPjtune	HP's GC visualization tool for analyzing garbage collection activity in a Java program.
jstat (1.5 only)	Attaches to the Java VM and collects and logs performance statistics.
jconsole (1.5 only)	Launches a simple console tool enabling you to monitor and manage an application launched with a management agent on a local or remote machine.
hprof	Simple profiler agent used for heap and CPU profiling.

1.1.6 Miscellaneous Tools and Options

The following tools and options do not fall into any of the previous categories:

Table 1-6 Miscellaneous Tools and Options

Tool or Option	Description and Usage
JAVA_TOOL_OPTIONS Environment Variable	Used to augment the options specified in the Java command line.
jvmstat Tools	Tools include `jps`, `jstat`, and `jstatd`. These tools are included with JDK 1.5.
visualgc	Uses jvmstat technology to provide visualization of garbage collection activity in the Java VM.
-verbose:class	Enables logging of class loading and unloading.
-verbose:jni	Enables logging of JNI (Java Native Interface).
-Xcheck:jni	Performs additional validation on the arguments passed to JNI functions.

1.1.7 JDK Tools Not Available on HP-UX

Some JDK tools are not available on HP-UX, so they are not described in this document. They are provided in JavaSoft JDK as unsupported tools. Equivalent functionality is available via `gdb` Java support, `HPjmeter`, and the HeapDump options.

Table 1-7 JDK Tools Not Available on HP-UX

Tool	Description and HP-UX Alternative
`jinfo`	Prints Java configuration information for a given Java process, core file, or remote debug server.
`jmap`	Prints shared object memory maps or Java heap memory details of a given process, core file, or remote debug server. Use the HeapDump options or gdb heap dump functionality instead.
`jstack`	Prints a Java stack trace of Java threads for a given Java process, core file, or remote debug server. Use gdb stack trace back functionality instead.
Serviceability Agent (SA)	Not yet ported to HP-UX.

1.2 Ctrl-Break Handler

A thread dump is printed if the Java process receives a SIGQUIT signal. Therefore, issuing the command `kill -3 <pid>` causes the process with id <pid> to print a thread dump to its standard output. The application continues processing after the thread information is printed.

In addition to the thread stacks, the ctrl-break handler also executes a deadlock detection algorithm. If any deadlocks are detected, the ctrl-break handler also prints out additional information on each deadlocked thread. The SIGQUIT signal can also be used to print heap dump information when using the `-XX:+HeapDump` or `-XX:+HeapDumpOnCtrlBreak` options described further on in this chapter.

Following is an example of output generated when SIGQUIT is sent to a running Java process:

```
Full thread dump  [Thu Oct 12 14:00:56 PDT 2006] (Java HotSpot(TM) Server VM 1.5.0.03 jinteg:02.13.06-21:25 IA64 mixed mode):

"Thread-3" prio=10 tid=00a78480 nid=24 lwp_id=2669798 runnable [0bfc0000..0bfc0ae0]
 at java.lang.Math.log(Native Method)
 at spec.jbb.JBButil.negativeExpDistribution(JBButil.java:795)
 at spec.jbb.TransactionManager.go(TransactionManager.java:234)
 at spec.jbb.JBBmain.run(JBBmain.java:258)
 at java.lang.Thread.run(Thread.java:595)

"Thread-2" prio=2 tid=009fb7a0 nid=23 lwp_id=2669797 runnable [0c1c0000..0c1c0b60]
 at spec.jbb.Order.dateOrderlines(Order.java:341)
 - waiting to lock <444ba618> (a spec.jbb.Order)
 at spec.jbb.DeliveryTransaction.process(DeliveryTransaction.java:213)
 at spec.jbb.DeliveryHandler.handleDelivery(DeliveryHandler.java:103)
 at spec.jbb.DeliveryTransaction.queue(DeliveryTransaction.java:363)
 - locked <154927e8> (a spec.jbb.DeliveryTransaction)
 at spec.jbb.TransactionManager.go(TransactionManager.java:431)
 at spec.jbb.JBBmain.run(JBBmain.java:258)
 at java.lang.Thread.run(Thread.java:595)

"Thread-1" prio=10 tid=008ffa80 nid=22 lwp_id=2669796 runnable [0c3c0000..0c3c0de0]
 at spec.jbb.infra.Collections.longStaticBTree.get(longStaticBTree.java:1346)
 at spec.jbb.Warehouse.retrieveStock(Warehouse.java:307)
 at spec.jbb.Orderline.validateAndProcess(Orderline.java:341)
 - locked <48563610> (a spec.jbb.Orderline)
 at spec.jbb.Order.processLines(Order.java:289)
 - locked <48563128> (a spec.jbb.Order)
 at spec.jbb.NewOrderTransaction.process(NewOrderTransaction.java:282)
 at spec.jbb.TransactionManager.go(TransactionManager.java:278)
 at spec.jbb.JBBmain.run(JBBmain.java:258)
 at java.lang.Thread.run(Thread.java:595)

"Thread-0" prio=2 tid=00781240 nid=21 lwp_id=2669795 runnable [0c5c0000..0c5c0e60]
 at spec.jbb.infra.Util.DisplayScreen.privIntLeadingZeros(DisplayScreen.java:448)
 at spec.jbb.infra.Util.DisplayScreen.putDollars(DisplayScreen.java:1214)
 at spec.jbb.NewOrderTransaction.secondDisplay(NewOrderTransaction.java:416)
 - locked <154d4828> (a spec.jbb.NewOrderTransaction)
 at spec.jbb.TransactionManager.go(TransactionManager.java:279)
 at spec.jbb.JBBmain.run(JBBmain.java:258)
 at java.lang.Thread.run(Thread.java:595)

"Low Memory Detector" daemon prio=10 tid=00778b80 nid=19 lwp_id=2669774 runnable [00000000..00000000]

"CompilerThread1" daemon prio=10 tid=00772c30 nid=17 lwp_id=2669772 waiting on condition [00000000..0a7ff728]

"CompilerThread0" daemon prio=10 tid=007703f0 nid=16 lwp_id=2669771 waiting on condition [00000000..0afff5b8]

"AdapterThread" daemon prio=10 tid=0076c8d0 nid=15 lwp_id=2669770 waiting on condition [00000000..00000000]

"Signal Dispatcher" daemon prio=10 tid=0076a2e0 nid=14 lwp_id=2669769 waiting on condition [00000000..00000000]

"Finalizer" daemon prio=10 tid=00530a60 nid=13 lwp_id=2669768 in Object.wait() [750c0000..750c0e60]
 at java.lang.Object.wait(Native Method)
 - waiting on <11000100> (a java.lang.ref.ReferenceQueue$Lock)
 at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:133)
 - locked <11000100> (a java.lang.ref.ReferenceQueue$Lock)
 at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:149)
```

```
 at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:197)

"Reference Handler" daemon prio=10 tid=0052de80 nid=12 lwp_id=2669767 in Object.wait() [752c0000..752c0ce0]
 at java.lang.Object.wait(Native Method)
 - waiting on <11003dc8> (a java.lang.ref.Reference$Lock)
 at java.lang.Object.wait(Object.java:474)
 at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:123)
 - locked <11003dc8> (a java.lang.ref.Reference$Lock)

"main" prio=8 tid=0047dc90 nid=1 lwp_id=-1 waiting on condition [7fffd000..7fffe398]
 at java.lang.Thread.sleep(Native Method)
 at spec.jbb.JBButil.SecondsToSleep(JBButil.java:740)
 at spec.jbb.Company.displayResultTotals(Company.java:942)
 at spec.jbb.JBBmain.DoARun(JBBmain.java:387)
 at spec.jbb.JBBmain.DOIT(JBBmain.java:1137)
 at spec.jbb.JBBmain.main(JBBmain.java:1490)

"VM Thread" prio=10 tid=004ff510 nid=11 lwp_id=2669766 runnable

"GC task thread#0 (ParallelGC)" prio=10 tid=004d0520 nid=3 lwp_id=2669758 runnable

"GC task thread#1 (ParallelGC)" prio=10 tid=004d0600 nid=4 lwp_id=2669759 runnable

"GC task thread#2 (ParallelGC)" prio=10 tid=004d06e0 nid=5 lwp_id=2669760 runnable

"GC task thread#3 (ParallelGC)" prio=10 tid=004d07c0 nid=6 lwp_id=2669761 runnable

"GC task thread#4 (ParallelGC)" prio=10 tid=004d08a0 nid=7 lwp_id=2669762 runnable

"GC task thread#5 (ParallelGC)" prio=10 tid=004d0980 nid=8 lwp_id=2669763 runnable

"GC task thread#6 (ParallelGC)" prio=10 tid=004d0a60 nid=9 lwp_id=2669764 runnable

"GC task thread#7 (ParallelGC)" prio=10 tid=004d0b40 nid=10 lwp_id=2669765 runnable

"VM Periodic Task Thread" prio=8 tid=00500ad0 nid=18 lwp_id=2669773 waiting on condition
```

1.3 Fatal Error Log (hs_err_pid<pid>.log)

When a fatal error occurs, an error log is created in the file hs_err_pid<pid>.log, where <pid> is the process id of the process. The file is created in the working directory of the process, if possible. In the event that the file cannot be created in the working directory (for example, if there is insufficient space, a permission problem, or another issue), then the file is created in the temporary directory,/tmp. The error log contains information obtained at the time of the fatal error. This includes :

- Operating exception or signal that provoked the fatal error
- Version and configuration information
- Details on the thread that provoked the fatal error and its stack trace
- List of running threads and their states
- Summary information about the heap
- List of native libraries loaded
- Command line arguments
- Environment variables
- Details about the operating system and CPU

In some cases, only a subset of this information is output to the error log. This happens when a fatal error is so severe that the error handler is unable to recover and report all details.

1.4 gdb

Java stack unwind enhancements have been added to gdb to enable it to support unwinding across Java frames and provide an effective way to examine stack traces containing mixed language frames (Java and C/C++) of both live Java processes and core files. This has been implemented by adding subcommands for Java VM debugging to gdb.

The following table shows which Java versions on PA-RISC and Integrity systems have the stack unwind and the gdb Java subcommands features. These features are available in gdb version 4.5 and later versions.

Table 1-8 Java Version Information for `gdb` Java VM Debugging Features

Platform	Stack Unwind Enhancements	Java Subcommands	GDB Version
PA–RISC 32–bit	SDK 1.3.1.02+	SDK 1.4.1.05+	4.5+
PA–RISC 64–bit	SDK 1.4.1.01+	SDK 1.4.1.05+	4.5+
Integrity 32–bit	SDK 1.3.1.06+	SDK 1.4.1.05+	4.5–5.2
Integrity 64–bit	SDK 1.4.0.01+	SDK 1.4.1.05+	4.5–5.2
Integrity 32, 64–bit	SDK 1.4.2.10+	SDK 1.4.2.10+	*5.3+
Integrity 32, 64–bit	JDK 1.5.0.03+	JDK 1.5.0.03+	*5.3+

*`gdb` version 5.3 requires SDK 1.4.2.10 and later versions or JDK 1.5.0.03 and later versions in order to use the Java VM debugging features.

In order to use this functionality, the `GDB_JAVA_UNWINDLIB` environment variable must be set to the path name of the Java unwind library. The default location of the Java unwind library on various systems is shown following. The examples are for SDK 1.4; if you are using JDK 1.5, substitute `/opt/java1.5` for `/opt/java1.4`.

```
/opt/java1.4/jre/lib/PA_RISC/server/libjunwind.sl
/opt/java1.4/jre/lib/PA_RISC2.0/server/libjunwind.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/server/libjunwind.sl
/opt/java1.4/jre/lib/IA64N/server/libjunwind.so
/opt/java1.4/jre/lib/IA64W/server/libjunwind.so
```

Following are a few examples. If you are using `ksh` on a PA-RISC machine, this is how you set the environment variable for a 32–bit Java application:

```
export GDB_JAVA_UNWINDLIB=/opt/java1.4/jre/lib/PA_RISC2.0/server/libjunwind.sl
```

Additionally, this is how you set the environment variable on an Integrity machine for a 32–bit Java application:

```
export GDB_JAVA_UNWINDLIB=/opt/java1.4/jre/lib/IA64N/server/libjunwind.so
```

If the SDK is installed in a location other than the default, substitute the non-default location for `/opt/java1.4` in the previous commands.

1.4.1 Java Stack Unwind Features

The Java stack unwind features are useful for troubleshooting problems in the Java VM. Following is a list of the Java stack unwind features:

- View mixed language frames information, including Java frames and C/C++ native frames, in a `gdb` backtrace.
- Distinguish various Java frame types including interpreted, compiled, and adapter frames.
- View Java method name, signature, and class package name for Java method frames.

Additional stack unwind features are available starting with SDK 1.4.2. These features fall into three categories: Java stack unwind enhancements, Java heap support, and Java threads support.

These additional features are available as part of the Java stack unwind enhancements:

- View Java compiled frame inlined methods.
- View Java interpreted or compiled frame specific information.
- View Java interpreted or compiled frame arguments and local variables.
- Disassemble Java method bytecodes.
- Print out the Java unwind table.

These additional features are available as part of the Java heap support:

- View Java heap parameters.
- Dump Java object.
- Print Java heap histogram.
- Find all the instances of a given Java class.
- Find all the references to a given object in the Java heap.
- Find out the object OOP (object-oriented pointer) of the given field address.

These additional features are available as part of Java threads support:

- View Java threads state information.
- View current Java thread information.
- View Java interpreted frame monitors information.

1.4.2 `gdb` Subcommands for Java VM Debugging

To view the `gdb` commands that support Java VM debugging, type `help java` at the `gdb` prompt.

```
(gdb) help java
Java and JVM debugging commands.

List of java subcommands:

java args -- Show the current or specified Java frame arguments info
java bytecodes -- Disassemble the given Java method's bytecodes
java heap-histogram -- Show the Java heap object histogram
java instances -- Find all the instances of the given klassOop in the Java heap
java jvm-state -- Show Java virtual machine's current internal states
java locals -- Show the current or specified Java frame locals info
java mutex-info -- Print out details of the static mutexes
java object -- Print out the given Java object's fields info
java oop -- Find the Java object oop of the given Java heap address
java references -- Find all the references to the given Java object in the Java heap
java unwind-info -- Show the unwind info of the code where the given pc is located
java unwind-table -- Print out the dynamically generated Java Unwind Table

Type "help java" followed by java subcommand name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

The following two tables list Java VM debugging commands and Java subcommands:

Table 1-9 Java VM Debugging Commands

`backtrace`	Print backtrace of mixed Java and native frames
`info frame`	Print Java frame specific information if this is a Java frame
`info threads`	Print state information for all threads
`thread`	Print detailed state information for the current thread

Table 1-10 Java Subcommands

`java args`	Show the current or specified Java frame arguments information
`java bytecodes`	Disassemble the given Java method's bytecodes
`java heap-histogram`	Show the Java heap object histogram
`java instances`	Find all the instances of the given klassOop in the Java heap
`java jvm-state`	Show the current internal state of the Java VM
`java locals`	Show the current or specified Java frame locals information

Table 1-10 Java Subcommands *(continued)*

`java object`	Print the given Java object's fields information
`java oop`	Find the Java object OOP of the given Java heap address
`java references`	Find all the references to the given Java object in the Java heap
`java unwind-info`	Show the unwind information of the code where the given pc is located
`java unwind-table`	Print the dynamically generated Java unwind table

Type `help java` followed by the subcommand name for full documentation. Command name abbreviations are allowed if they are unambiguous.

Following are examples that illustrate the `gdb` command-line options for invoking `gdb` on a core file and on a hung process.

The first set of examples illustrate how to invoke `gdb` on a core file:

- Invoke `gdb` on a core file generated when running a 32-bit Java application on an Integrity system with `/opt/java1.4/bin/java`:

  ```
  $ gdb /opt/java1.4/bin/IA64N/java core.java
  ```

- Invoke `gdb` on a core file generated when running a 64-bit Java application on an Integrity system with `/opt/java1.4/bin/java -d64`:

  ```
  $ gdb /opt/java1.4/bin/IA64W/java core.java
  ```

- Invoke `gdb` on a core file generated when running a 32-bit Java application on PA-RISC using `/opt/java1.4/bin/java`:

  ```
  $ gdb /opt/java1.4/bin/PA_RISC2.0/java core.java
  ```

- Invoke `gdb` on a core file generated when running a 64-bit Java application on PA-RISC using `/opt/java1.4/bin/java`:

  ```
  $ gdb /opt/java1.4/bin/PA_RISC2.0W/java core.java
  ```

When debugging a core file, it is good practice to rename the file from `core` to another name to avoid accidentally overwriting it.

If the Java and system libraries used by the failed application reside in non-standard locations, then the `GDB_SHLIB_PATH` environment variable must be set to specify the location of the libraries.

The following example illustrate how to invoke `gdb` on a hung process:

- Determine the process id:

  ```
  $ ps -u user1 | grep java
      23989 pts/9     8:52 java
  ```

- Attach `gdb` to the running process:

  ```
  $ gdb -p 23989

    HP gdb 5.0 for HP Itanium (32 or 64 bit) and target HP-UX 11.2x.
    Copyright 1986 - 2001 Free Software Foundation, Inc.
    Hewlett-Packard Wildebeest 5.0 (based on GDB) is covered by the
    GNU General Public License.Type "show copying" to see the conditions to
    change it and/or distribute copies. Type "show warranty" for
    warranty/support.

    Reading symbols from /opt/java1.4/bin/IA64N/java...
  ```

```
(no debugging symbols found)...done.
Attaching to program: /opt/java1.4/bin/IA64N/java, process 23989
(no debugging symbols found)...
Reading symbols from /usr/lib/hpux32/libpthread.so.1...
(no debugging symbols found)...done.
Reading symbols from /usr/lib/hpux32/libdl.so.1...
...
```

NOTE: If the version of gdb on the system is older than version 4.5, it will be necessary to specify the full path of the Java executable in order to use the gdb subcommands. For example:

gdb /opt/java1.4/bin/PA_RISC2.0/java –p 23989

A tutorial on gdb may be found at the following website:

http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,1677,00.html

1.5 HPjconfig

HPjconfig is a configuration tool for tuning your HP-UX 11i system to match the characteristics of your application. It provides kernel parameter recommendations tailored to your HP-UX hardware platform and application characteristics. HPjconfig has features for saving and restoring configurations so you can distribute customized recommendations across your customer base.

HPjconfig can also be used to verify that your systems has all the necessary patches required for Java. The patches required for Java can be found at the following website:

http://www.hp.com/products1/unix/java/patches

HPjconfig runs on SDK 1.3.1 and later versions, SDK 1.4.x, and JDK 1.5.0.x. HP-UX 11.00 or later versions is required. All HP-UX 11i HP Integrity and HP 9000 PA-RISC systems are supported.

For more information about HPjconfig including the download, go to:

http://www.hp.com/products1/unix/java/java2/hpjconfig/index.html

HPjconfig can be run in either graphical user interface (GUI) mode or non-GUI (command-line) mode. In either mode, it generates a summary of the configuration information in the log file named HPjconfig_<hostname>_<date>_<timestamp>.log. This log file name can be specified using the -logfile option.

Following is usage information for the HPjconfig command:

```
usage:
    HPjconfig [ options ] -gui
    HPjconfig [ options ] <object> <action>

objects: -patches &| -tunables
actions: -listreq | -listmis | -listpres | -apply

options:
    -patches     operate on java-specific patches
    -tunables    operate on java-specific tunables
    -listreq     list all java required patches or tunables that are applicable to this system
    -listmis     list missing java-specific patches or tunables on the system
    -listpres    list applied (installed) java-specific patches or tunables on the system
    -apply       apply (install) missing java-specific patches or tunables on the system
    -javavers  s java versions for selecting patches e.g 1.2, 1.3, 1.4, 5.0
    -[no]gui     run in GUI mode
    -logfile   s name of log file
    -proxyhost s HTTP proxy host name for accessing live data
    -proxyport s HTTP proxy port for accessing live data
    -help        show help string and exit
    -version     show version string
```

Following are examples of invoking HPjconfig in GUI mode from the csh and the ksh:

```
(csh) $ setenv DISPLAY <Display's IP Address>:0.0
      $ setenv PATH $PATH:/usr/sbin
      $ java -jar HPjconfig.jar

(ksh) $ export DISPLAY=<Display's IP Address>:0.0
      $ export PATH=$PATH:/usr/sbin
      $ java -jar HPjconfig.jar
```

The following four figures show the System, Application, Patches, and Tunables tabs for the `HPjconfig` tool:

Figure 1-1 `HPjconfig` **- System Tab**

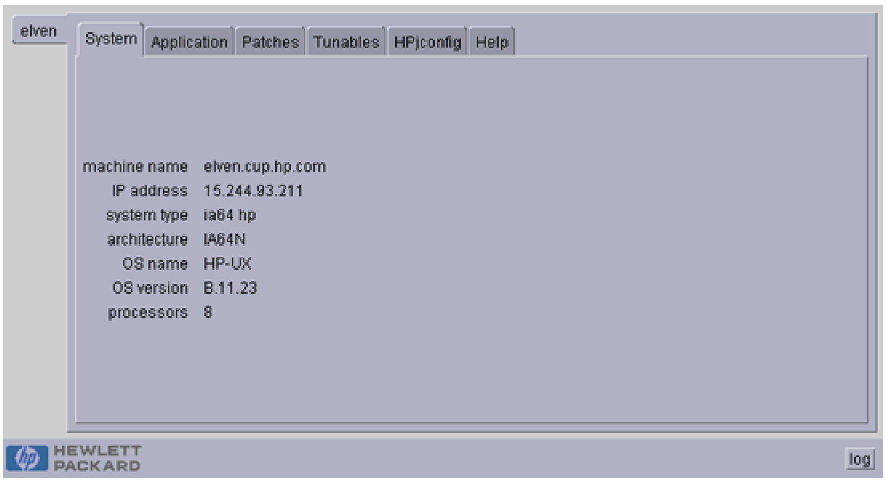


Figure 1-2 `HPjconfig` **- Application Tab**

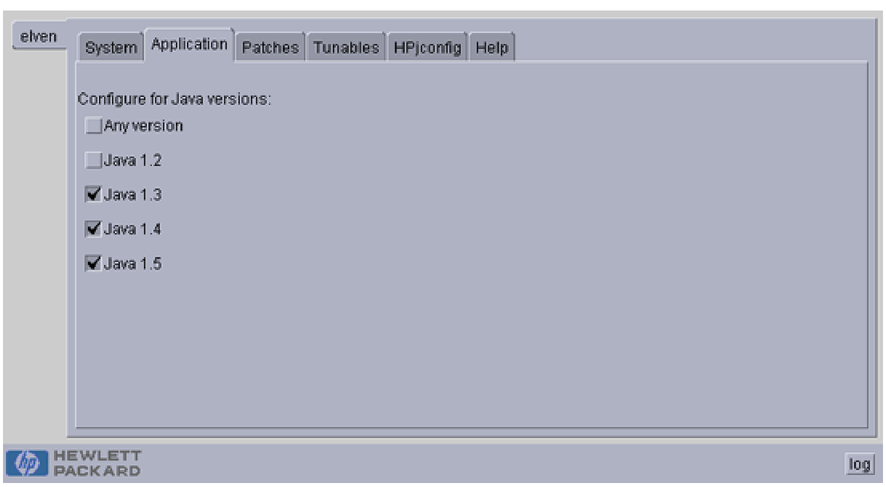


Figure 1-3 `HPjconfig` **- Patches Tab**

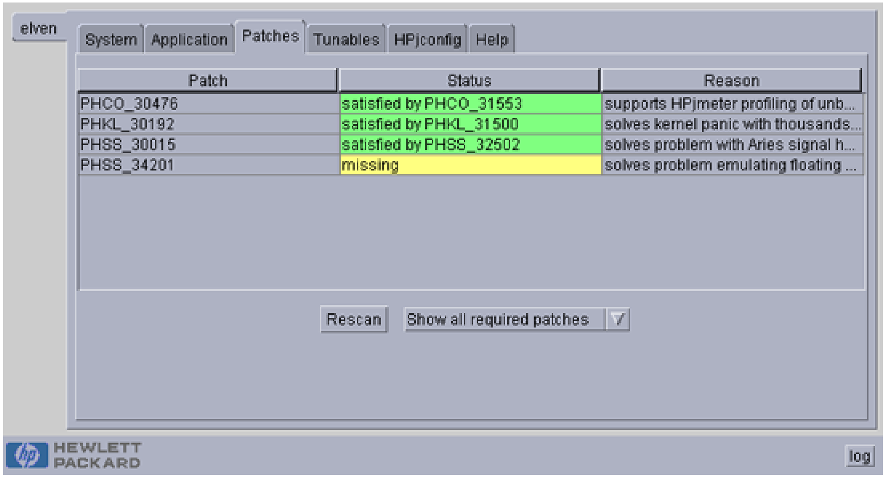
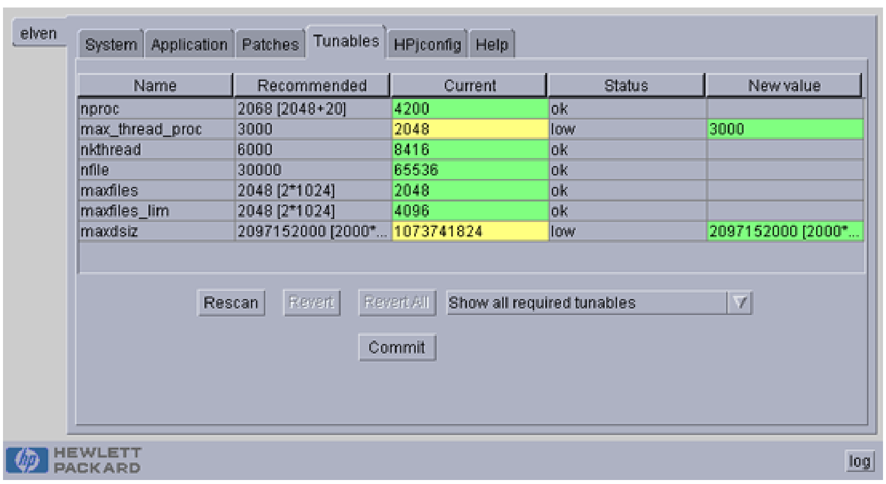


Figure 1-4 `HPjconfig` **- Tunables Tab**



Following are the commands for invoking `HPjconfig` in non-GUI mode. The `-help` option lists options you can use in this mode.

```
$ cd <hpjconfig_installation_dir>
$ java -jar ./HPjconfig.jar -nogui -help
```

Following is an example using `HPjconfig` in non-GUI mode to list missing patches for Java SDK 1.4:

```
$ java -jar HPjconfig.jar -nogui -patches -listmis -javavers 1.4
Log written to HPjconfig_mutant_20060915_040458.log
List of missing patches:
PHSS_34201      solves problem emulating floating point conversion when running
PA2.0 Java on an IPF system.. solves problem with Aries signal
handling that overlaps Java signal handling. solves problem emulating
floating point conversion when running PA2.0 Java on an IPF system..
solves problem with Aries signal handling that overlaps Java signal
handling.
```

Following is an example using `HPjconfig` to show the values for HP-UX tunables required by Java:

```
$ java -jar HPjconfig.jar -nogui -tunables -listreq
Log written to HPjconfig_mutant_20060915_040934.log
List of required tunables:
Name                            Recommended value
```

```
nproc                           2048+20
max_thread_proc                 3000
nkthread                        6000
nfile                           30000
maxfiles                        2*1024
maxfiles_lim                    2*1024
maxdsiz                         2000*1024*1024
```

Following is an example of using `HPjconfig` to display tunables that are set to values less than those recommended:

```
$ java -jar HPjconfig.jar -nogui -tunables -listmis
Log written to HPjconfig_mutant_20060915_040955.log
List of tunables whose values are less than the recommended values:
Name                            Recommended value
max_thread_proc                 3000
maxdsiz                         2000*1024*1024
```

Following is an example log file produced by `HPjconfig`:

```
$ more HPjconfig_server1_20060915_042600.log
   Fri Sep 15 16:26:00 PDT 2006

   HPjconfig 3.0.01 (Thu Jul 21 14:52:47 2005)

   Machine name:  server1
   IP address:    15.244.94.25
   System type:   ia64 hp server rx5670
   Architecture:  IA64N
   OS name:       HP-UX
   OS version:    B.11.23
   Processors:    4

   Java version:  1.4

   Reading required patches/tunables information from /tmp/HPjconfig.xml
   Read required patches/tunables information

   Reading patch list from system
   Read patch list from system

   List of required patches:
   PHCO_30476    supports HPjmeter profiling of unbound (MxN) threads.
   PHKL_30192    solves kernel panic with thousands of MxN threads.
   PHSS_30015    solves problem with Aries signal handling that overlaps Java sig
   nal handling.
   PHSS_34201    solves problem emulating floating point conversion when running
   PA2.0 Java on an IPF system.. solves problem with Aries signal handling that ove
   rlaps Java signal handling. solves problem emulating floating point conversion w
   hen running PA2.0 Java on an IPF system.. solves problem with Aries signal handl
   ing that overlaps Java signal handling.
```

1.6 HPjmeter

With the release of `HPjmeter` 3.0, all previous versions of `HPjmeter` (1.x, 2.x) are no longer available for download and are no longer supported by HP.

If you have an old version of `HPjmeter`, please download `HPjmeter` 3.0 from:

http://www.hp.com/products1/unix/java/hpjmeter/index.html

`HPjmeter` can be used to identify and diagnose performance problems in Java applications running on HP-UX. It can also monitor live Java applications and analyze profiling data generated by the `-Xrunhprof:heap=dump`, `-Xeprof`, `-Xverbosegc`, `-Xloggc`, and `–XX:+HeapDump` options. Additionally, when using JDK 1.5.04 or later releases, `HPjmeter` can capture profiling data with zero preparation (that is, without pre-planning).

The following table lists the features of HPjmeter 3.0:

Table 1-11 HPjmeter 3.0 Features

Integrated HPjtune functions with concurrent improvements in tool and help usability
Ability to examine Java Management Extension management beans (Mbeans) content and the Java VM internal memory configuration
Automatic problem detection and alerts • Memory leak detection alerts with leak rate • Thread deadlock detection • Abnormal thread termination detection • Expected out of memory error • Excessive method compilation
Dynamic real-time display of application behavior • Java heap size • Garbage collection events and percentage time spent in garbage collection • CPU usage per method for hottest methods
Object allocation percentage by method • Object allocation percentage by object type • Method compilation count in the Java VM dynamic compiler • Number of classes loaded by the Java VM • Thrown exception statistics • Multi-application, multi-node monitoring from a single console
Drill down into application profile metrics • Graphic display of profiling data • Call graphs with call count, or with CPU or clock time • Per thread display of time spent • Per thread or per process display

HPjmeter can display data generated by the following Java product versions, on the specified architectures, with the specified HP-UX operating system, as detailed in the following table:

Table 1-12 Java SDKs and JDKs Supported by HPjmeter 3.0

Java Version	Architecture	HP-UX Versions
SDK 1.4.2.02 or later	PA-RISC 1.1, PA-RISC 2.0	11.11, 11.23
JDK 1.5 or later	PA-RISC 2.0	11.11, 11.23
SDK 1.4.2.02 or later	Integrity	11.22, 11.23
JDK 1.5.x	Integrity	11.22, 11.23

The HPjmeter console can be run on:

- PA-RISC HP-UX 11.11, 11.23
- Integrity HP-UX 11.22, 11.23
- Windows XP/2000/NT
- Linux

The user's guide for HPjmeter may be found at:

http://www.hp.com/products1/unix/java/hpjmeter/infolibrary/user_guide.pdf

More information on HPjmeter may be found at:

http://www.hp.com/products1/unix/java/hpjmeter/index.html

1.6.1 Using HPjmeter to Monitor Applications

The following steps show how to start the monitoring agent when launching the HPjmeter console. For most Java installations, linkage to the appropriate libraries is completed automatically as part of the installation process, and, therefore, the first step is not needed. Begin with the second step if you have a standard installation of the Java Runtime Environment.

1. Set the SHLIB_PATH environment variable to include the location of the HPjmeter agent library as appropriate for 32 or 64-bit Java VM.

   Following are examples that show how to set this variable in both the csh and the ksh for the different libraries.

   To select the PA-RISC 32-bit library:

   ```
   (csh) setenv SHLIB_PATH /opt/hpjmeter/lib/PA_RISC2.0
   (ksh) export SHLIB_PATH=/opt/hpjmeter/lib/PA_RISC2.0
   ```

   To select the PA-RISC 64-bit library:

   ```
   (csh) setenv SHLIB_PATH /opt/hpjmeter/lib/PA_RISC2.0W
   (ksh) export SHLIB_PATH=/opt/hpjmeter/lib/PA_RISC2.0W
   ```

   To select the Integrity 32-bit library:

   ```
   (csh) setenv SHLIB_PATH /opt/hpjmeter/lib/IA64N
   (ksh) export SHLIB_PATH=/opt/hpjmeter/lib/IA64N
   ```

   To select the Integrity 64-bit library:

   ```
   (csh) setenv SHLIB_PATH /opt/hpjmeter/lib/IA64W
   (ksh) export SHLIB_PATH=/opt/hpjmeter/lib/IA64W
   ```

2. Confirm that the node agent is running. With a standard installation, the node agent should be running as a daemon on the system where it was installed. A node agent must be running before the console can connect to a managed node to discover applications and open monitoring sessions.

   To verify that the node agent is running, use the following ps command:

   ```
   % ps -ef | grep node
   ```

   The last output column (the *args* column) from ps should show the following:

   ```
   $JMETER_HOME/bin/nodeagent -daemon
   ```

   where JMETER_HOME=/opt/hpjmeter. The -daemon flag indicates that the node agent is running as a daemon.

   If the node agent is not running, follow these steps to enable it:

   a. Verify that you are logged in with root permissions.
   b. Check that the following files exist:
      - /sbin/init.d/HPjmeter_NodeAgent
      - /sbin/rc3.d/S999HPjmeter_NodeAgent
   c. Issue the following command to start the node agent daemon manually. Note: substitute start with stop to stop the node agent.

      ```
      $ /sbin/init.d/HPjmeter_NodeAgent start
      ```

   If you cannot use the node agent as a daemon or you need to set up access restrictions, start the node agent manually by issuing the following command (no root access needed):

```
$ /opt/hpjmeter/bin/nodeagent
```

By default, the node agent listens for console connections on port 9505. Use the `-port port_number` option to specify an alternate port number.

3. Start the Java application with the Java VM agent. For example, to start the `myapp` application on JDK 1.5 enter:

```
/opt/java1.5/bin/java -Xms256m -Xmx512m -agentlib:jmeter myapp
```

   On SDK 1.4.2 versions enter:

```
/opt/java1.4/bin/java -Xms256m -Xmx512m  \
 -Xbootclasspath/a:$JMETER_HOME/lib/agent.jar -Xrunjmeter myapp
```

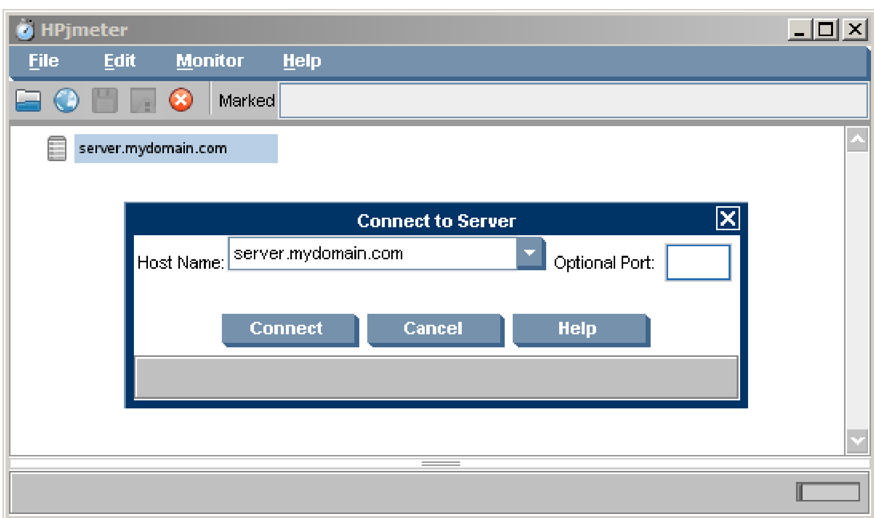
   This enables the `myapp` process to be dynamically monitored with the console.
4. Start the `HPjmeter` console by entering the command:

```
/opt/hpjmeter/bin/hpjmeter
```

1.6.2 Connect to the Node Agent From the `HPjmeter` Console

1. Choose Connect from the File Menu or select the Connect to Server icon [ ]. The following screen displays:

Figure 1-5 `HPjmeter` **- Connecting to Server**



2. In the Connect to Server dialog box, type the host name where the Java application and corresponding node agent are running.
3. If the node agent was started on a nonstandard port, specify the port number in the Optional Port box.
4. Select `Connect`. The running Java VM for each application should appear in the console main window pane marked with the symbol.

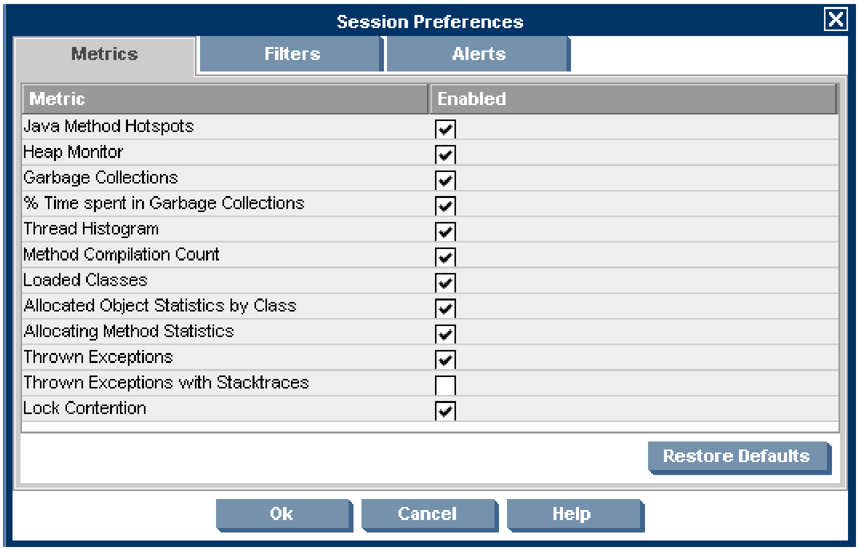
NOTE: If there is a connection failure, the symbol will not be displayed. Instead the symbol will be displayed next to the server name to indicate the server connection failure. If this happens, verify the node agent is running on the specified server.

5. If you want to connect to several node agents, repeat the previous steps.

1.6.3 Set Session Preferences

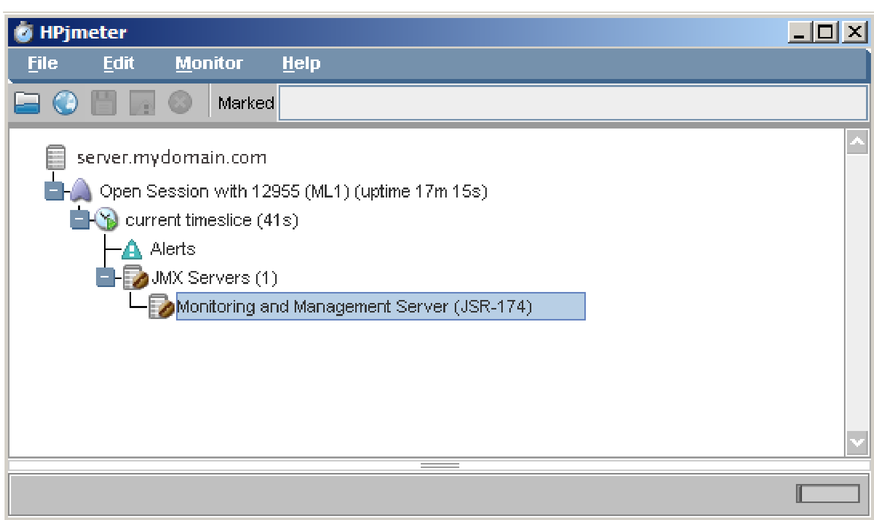
1. Double-click the Java VM icon in the data pane for the application that you want to monitor. This opens the Session Preferences dialog box shown in the following screen:

Figure 1-6 `HPjmeter` **- Setting Session Preferences**



2. Check the default settings for metrics, filters, and alerts, and enable the settings you want to activate.
3. Click OK. The Session Preferences window will close and the newly Open Session will be visible, marked by the icon. Refer to the following screen for an example:

Figure 1-7 `HPjmeter` **- Collecting Metrics**

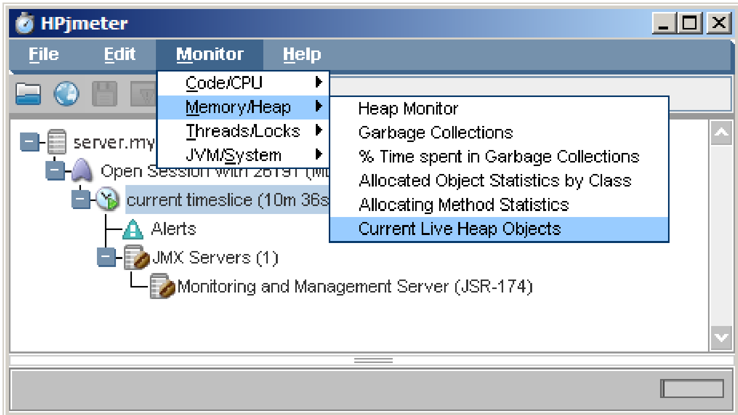


4. Wait for the console to collect metrics. The length of time depends on the application size, the load imposed on the application, and the selected preferences. Typically, the wait will be from 5 to 30 minutes. Longer collection time gives you greater accuracy in the results.

1.6.4 Viewing Monitoring Metrics During Your Open Session

1. Click the open session or time slice to highlight the data to be viewed.
2. Use the *Monitor* menu on the console main window to select the desired metrics. Refer to the following screen for an example:

Figure 1-8 `HPjmeter` **- Choosing Metrics to Monitor**



3. Select a metric. A metric visualizer displaying the chosen data will open. Refer to the `HPjmeter` User's Guide for details on individual metrics and how to interpret the data.

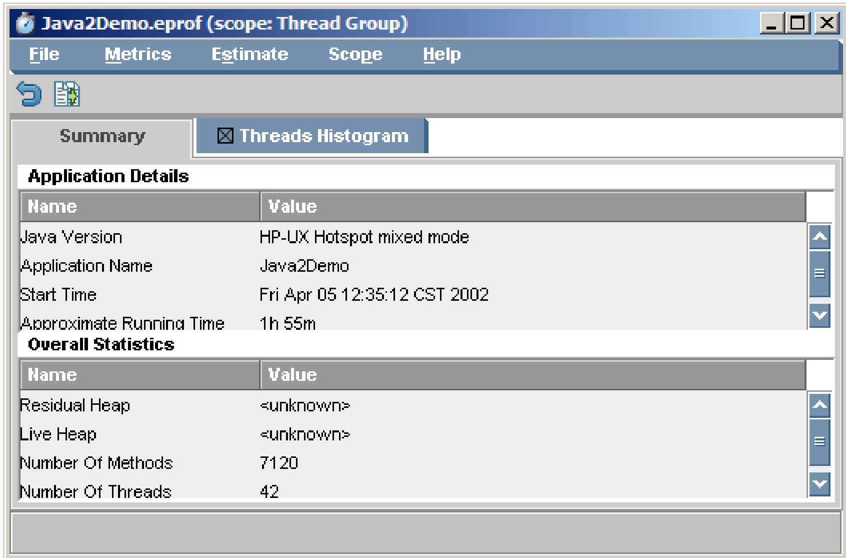
1.6.5 Using `HPjmeter` to Analyze Profiling Data

The following steps summarize how to use `HPjmeter` to save and view profiling information from your applications.

NOTE: If you are running JDK 1.5.0.04 or later, you can send a signal to the running Java VM to start and stop a profiling data collection period. This can be done without pre-planning and without interrupting your application. See *Profiling with Zero Preparation* in the `HPjmeter` User's Guide.

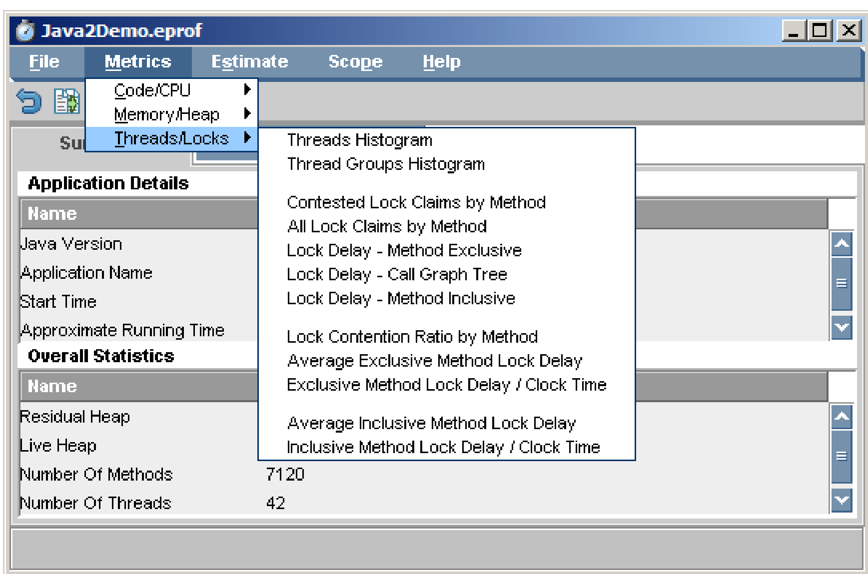
1. Change the command line of your Java application to use `-Xeprof` or `-agentlib:hprof` options to capture profiling data.
2. Run the application to create a data file.
3. Start the console from a local installation on your client workstation.
4. Click `File—>Open File` to browse for and open the data file.
5. A profile analysis window will open displaying a set of tabs containing summary and graphical metric data. The following screen shows an example:

Figure 1-9 `HPjmeter` **- Profile Data**



6. Click among the tabs to view available metrics. Use the Metrics or Estimate menus to select additional metrics to view. Each metric you select opens in a new tab. Mousing over each category in the cascading menu will reveal the relevant metrics for that category. The following screen shows the available metrics for the threads/locks category:

Figure 1-10 `HPjmeter` **- Threads/Locks Metrics**

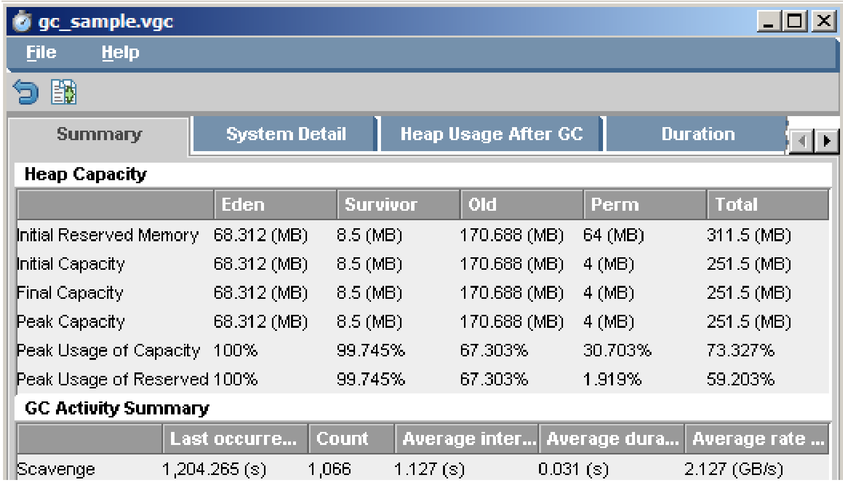


1.6.6 Using `HPjmeter` to Analyze Garbage Collection Data

The following steps summarize how to use `HPjmeter` to save and view garbage collection information from your applications:

1. Change the command line of your Java application to use `-Xverbosegc` or `-Xloggc` to capture garbage collection data.
2. Run the application to create a data file.
3. Start the console from a local installation on your client workstation.
4. Click `File—>Open File` to browse for and open the data file.
5. A GC viewer window opens and displays a set of tabs containing metric data. Following is an example garbage collection analysis screen:

Figure 1-11 `HPjmeter` - Garbage Collection Analysis



1.6.7 Running the `HPjmeter` Sample Programs

`HPjmeter` includes two sample applications you can run to see live examples of a memory leak and a thread deadlock situation. You can use the visualizers to examine data during the demonstration session.

Following are the general steps for running the sample applications:

1. Start the console.
2. Start the node agent if it is not running as a daemon.
3. Start the sample application from the command line:

   ```
   $ cd $JMETER_HOME/demo
   $ export LD_LIBRARY_PATH=$JMETER_HOME/lib
   $ java -agentlib:jmeter agent.jar -jar ML1.jar
   ```

   As a convenience, `HPjmeter` includes a script that sets up the library path and bootclasspath using the Java VM found at installation time. Following are instructions for using this script:

   ```
   $ cd $JMETER_HOME/demo
   $ ../bin/run_simple_jvmagent -jar sample_program
   ```

   Use the file name of the specific sample you want to run in place of `sample_program`.
4. In the console main window, select Connect and type in the host name of the machine running the sample application. If you specified a port number when starting the node agent, use the same port number. Otherwise, leave the port number box empty.
5. An icon representing the host appears in the main window. After a few moments, the console also shows the sample application as a child node of the host.
6. Double-click the application node to open a monitoring session with the application.
7. Click OK to accept the default settings for metrics, filters, and alerts.

1.6.7.1 Sample Memory Leak Application

This application demonstrates how memory leak alerts work in `HPjmeter`. It uses a simple program which allocates some objects. The program uses a `java.util.Vector` object to retain references to some of the objects. This application is configured to leak memory at the rate of about 10 MB per hour. It is available from the `HPjmeter` installation directory:

```
Source: $JMETER_HOME/demo/ML1.java
Binary: $JMETER_HOME/demo/ML1.jar
```

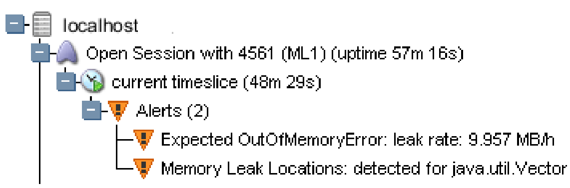
Use the class name `ML1` with the `run_simple_jvmagent` script to start the sample. When measuring the sample application, allow considerable time for the heap to mature and stabilize, and for the Java VM agent to collect memory leak data. Eventually, you will see the following two alerts:

- Expected OutOfMemory Error Alert with the leaking rate
- Memory Leak Locations Alert with the leak location

When using the default garbage collectors and heap size for SDK 1.4.2, the detection of a memory leak for this demonstration program occurs after about 20 minutes. This time may be substantially longer when using a different Java VM or nonstandard garbage collector or heap settings. In real situations, the detection time depends on the maximum heap size, the size of the leak, the application running time, and the application and load characteristics. Typically, the detection will occur in about one hour.

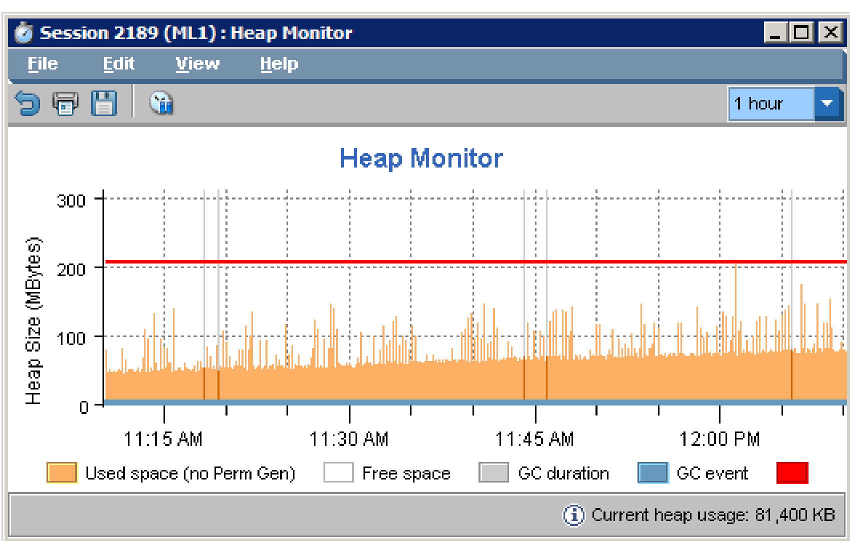
Following is a memory leak alert for the sample program:

Figure 1-12 `HPjmeter` **- Memory Leak Alert**



Following is the heap display:

Figure 1-13 `HPjmeter` **- Heap Monitor Display**



1.6.7.2 Sample Thread Deadlock Application

This application demonstrates how `HPjmeter` detects deadlocked threads. It creates pairs of threads every 30 seconds, stopping at 50 threads, which synchronize work using shared locks. Occasionally, the program reverses the order on which locks are taken, eventually causing a deadlock, which generates a Thread Deadlock Alert.

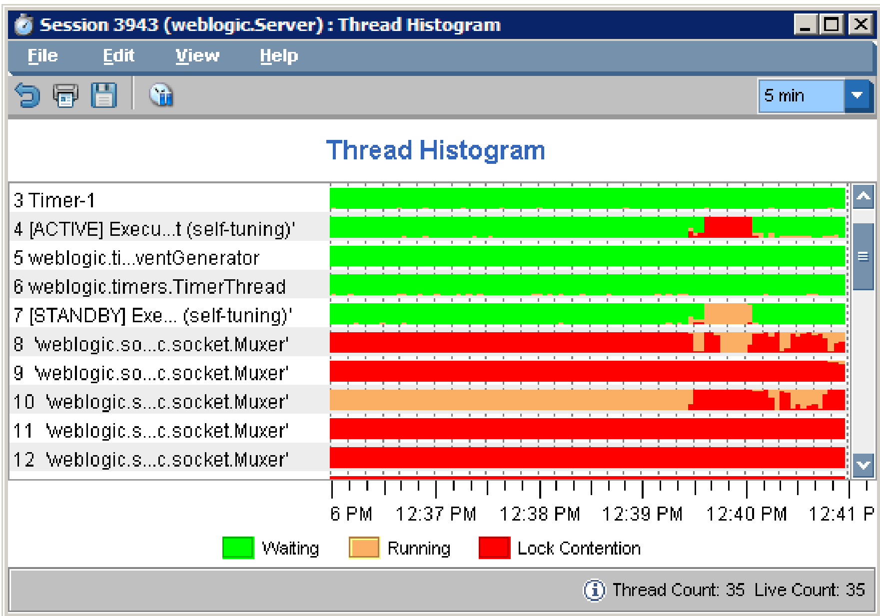
The sample application is available from the `HPjmeter` installation directory:

```
Source: $JMETER_HOME/demo/DL1.java
Binary: $JMETER_HOME/demo/DL1.jar
```

Use the class name `DL1` with the `run_simple_jvmagent` script to start the sample. Use the Thread Histogram display to view the thread activity. Deadlocked threads show a solid red bar.

Following is an example thread histogram display:

Figure 1-14 `HPjmeter` **- Thread Histogram**



1.7 `HPjtune`

NOTE: The `HPjtune` product has reached end of life. HP has integrated `HPjtune` functionality into `HPjmeter` 3.0 and recommends migrating to `HPjmeter` for the latest in bug fixes, enhancements, and support.

`HPjtune` is a garbage collection visualization tool for analyzing garbage collection activity in a Java program. Data files for `HPjtune` can be generated using -Xverbosegc or -verbose:gc. `HPjtune` lets you view this data in the following ways:

- Predefined graphs, which show the utilization of garbage collector resources and the impact of the garbage collector on application performance.
- User-configurable graphs, which access selected GC metrics.
- Other predefined graphs, which show GC behavior pertaining to threads.

`HPjtune` also includes a unique feature which allows you to use the data collected with the -Xverbosegc option to predict the effect of new garbage collector parameters on future application runs.

For more information about `HPjtune` and to download the tool, go to:

http://www.hp.com/products1/unix/java/java2/hpjtune/index.html

Following is an example of running Java with the -Xverbosegc option to generate a data file to be used by `HPjtune`:

```
$ /opt/java1.5/bin/java -Xverbosegc:file=java2d_gc.out -jar Java2Demo.jar
```

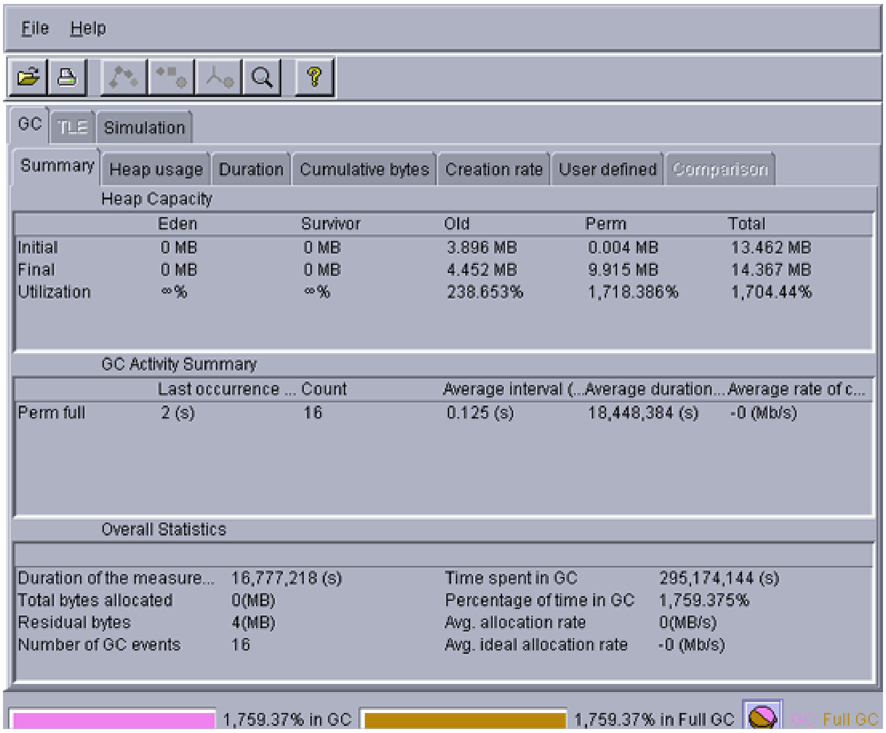
The -Xverbosegc option causes a data file containing garbage collection data to be generated into file `java2d_gc.out.<pid>`. This is how to invoke `HPjtune` on that file:

```
$ /opt/java1.5/bin/java -jar <HPjtune_insdir>/hpjtune/HPjtune.jar java2d_gc.out.15878
```

where <HPjtune_insdir> is the location of the `HPjtune` installation.

Following is an example screen shot to illustrate `HPjtune`'s output:

Figure 1-15 `HPjtune` **Screen**



1.8 hat

The `hat` tool is a third-party tool that can be used for heap analysis. It starts a web server on a binary-format heap dump file produced by one of the heap dump options such as -XX:+HeapDumpOnCtrlBreak or `-Xrunhprof:heap=dump,format=b`.

Following in an example using `hat`. The first command generates a binary heap dump file. The second command invokes `hat` on the binary heap profile.

```
$ java -Xrunhprof:heap=dump,format=b MyApp

$ hat -port=7002 java.hprof
```

The `hat` tool sets up an http server on the specified port. It can then be accessed by bringing up the default page in a web browser, for example, `http://<hostname.domain>:7002`. If you run `hat` on the same system as the browser, the server can be accessed by navigating to the URL `http://<hostname.domain>:7002`.

For more information on `hat`, refer to the following website:

https://hat.dev.java.net

For invocation details, refer to:

https://hat.dev.java.net/doc/README.html

NOTE: Beginning with Java SE 6, `hat` will be replaced with `jhat`, which will be included with the standard 6.0 distribution. For information on `jhat`, refer to the following website:

http://java.sun.com/javase/6/docs/technotes/tools/share/jhat.html

1.9 `hprof`

`hprof` is a simple profiler used for heap and CPU profiling. For more information, refer to:

http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html#hprof

1.10 `java.security.debug` System Property

The `java.security.debug` system property controls whether the security checks in the JRE (Java Runtime Environment) print trace messages during execution. This option can be useful when trying to determine why a `SecurityException` is thrown by a security manager. This system property can be set to one of the following values:

- `access` —print all `checkPermission` results
- `jar` —print jar verification information
- `policy` —print policy information
- `scl` —print permissions assigned by the `SecureClassLoader`

The `access` option has the following sub-options:

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

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

Following is an example showing 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)
```

1.11 `JAVA_TOOL_OPTIONS` Environment Variable

The command line used to start an application is not always readily accessible in many environments. This is especially true with applications that use embedded Java VMs or ones where the startup is deeply nested in scripts. In these environments, the `JAVA_TOOL_OPTIONS` environment variable may be useful to add options to the command line when the application is run. This environment variable is primarily intended to support the initialization of tools, specifically the launching of native or Java agents using the `-agentlib` or `-javaagent` options.

The `JAVA_TOOL_OPTIONS` environment variable is processed at the time of the invocation of the Java VM. When this environment variable is set, the `JNI_CreateJavaVM()` function prepends the value of the environment variable to the options supplied in its `JavaVMInitArgs` argument. For security reasons this option is disabled in setuid processes; that is, processes where the effective user or group ID differs from the real user or group ID.

In the following example, the environment variable is set to launch the `hprof` profiler when the application is started:

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

Although this environment variable is intended to support the initialization of tools, it is also useful for augmenting the command line with options for diagnostics purposes. For example, you could use it to add the -XX:OnError option to the command line when it would be helpful for a script or command to be executed when a fatal error occurred.

Since this environment variable is processed when `JNI_CreateJavaVM()` is called, it cannot be used to augment the Java launcher options. Some examples of these launcher options are the following VM selection options:

- `java -d64`
- `java -client`
- `java -server`

To pass arguments to the Java launcher, set the `JAVA_LAUNCHER_OPTIONS` environment variable to a string containing the desired arguments.

This environment variable is fully described in the JVMTI specification at:

http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html#tooloptions

1.12 `jconsole` (1.5 only)

The `jconsole` command launches a graphical console tool that enables you to monitor and manage Java applications on a local or remote machine.

`jconsole` can attach to any application that is started with the Java Management Extensions (JMX) agent. A system property defined on the command line enables the JMX agent. Once attached, `jconsole` can be used to display useful information such as thread usage, memory consumption, and details about class loading, runtime compilation, and the operating system.

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.

To use `jconsole`:

1. Start the application with the `-Dcom.sun.management.jmxremote` option. This option sets the `com.sun.management.jmxremote` system property, which enables the JMX agent.
2. Start `jconsole` with the `jconsole` command.
3. When `jconsole` starts, it shows a window listing the managed Java VMs on the machine. The process id (pid) and command line arguments for each Java VM are displayed. Select one of the Java VMs, and `jconsole` attaches to it.

Following is an example invocation of `jconsole`. First the Java application must be started with the JMX agent enabled:

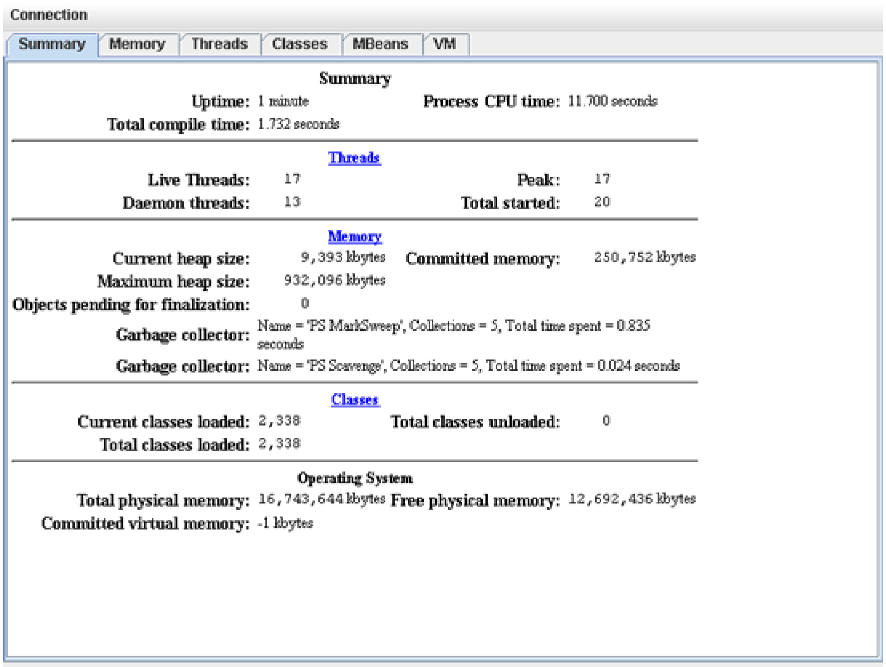
```
$ java -Dcom.sun.management.jmxremote -jar Java2Demo.jar &
 [1] 13028
```

Now the `jconsole` tool can be started on the managed Java VM:

```
$ /opt/java1.5/bin/jconsole 13028
```

The following figure shows a `jconsole` screen shot:

Figure 1-16 `jconsole` **Screen**



To learn more about `jconsole`, refer to the following website:

http://java.sun.com/j2se/1.5.0/docs/guide/management/jconsole.html

1.13 `jdb`

The SDK includes a command-line debugger, `jdb`, to help you find and fix bugs in Java programs running on a local or remote Java machine. Refer to the following website for more information:

http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/jdb.html

A `jdb` tutorial may be found at:

http://www.javaworld.com/javaworld/javaqa/2000-06/04-qa-0623-jdb.html

1.14 `jps` (1.5 only)

The `jps` tool lists the Java VMs on the target system. The tool is limited to reporting information on Java VMs that the user has access rights to, as determined by HP-UX specific access control mechanisms. For example, if a non-root user executes the `jps` command, a listing of all virtual machines started with that user's uid is given by the operating system.

Following is the usage information for the `jps` command:

```
Usage:  jps [-help]
        jps [-q] [-mlvV] [<hostname>[:<port>]]

Description of options:
  -q  Suppress the output of the class name, JAR file name, and arguments
      passed to the main method, producing only a list of local JVM pids
  -m  Show the arguments passed to the main method. This output may be null
      for embedded JVMs.
  -l  Show the full package name for the application's main class or the
      full path name of the application's JAR file.
  -v  Show the arguments passed to the JVM.
  -V  Show the arguments passed to the JVM through the flags file
      (the .hotspotrc file or the file specified by -XX:Flags=<filename>).

Note:  These options are subject to change or removal in the future.
```

Following is an example using `jps`:

```
$ /opt/java1.5/bin/jps -lmv
  16666 sun.tools.jps.Jps -lmv
  -Denv.class.path=.:/opt/java1.5/lib/classes.zip -Dapplication.home=/opt/java1.5 -Xms8m
  16665 MyObjectWaiterApp -Xverbosegc
  16641 spec.jbb.JBBmain -propfile S.pr.8 -Xmx1600m -Xms1600m -Xmn1500m
```

For more information about `jps`, refer to the following document:

http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jps.html

1.15 `jstat` (1.5 only)

The `jstat` utility is a statistics monitoring tool. It attaches to a Java VM and collects and logs performance statistics as specified by the command line options. The target Java VM is identified by its virtual machine identifier.

The `jstat` utility does not require the Java VM to be started with any special options. This utility is included in the JDK download.

The following table lists the `jstat` command options:

Table 1-13 Options to the `jstat` Command

-class	Prints statistics on the behavior of the class loader
-compiler	Prints statistics on the behavior of the Java compiler
-gc	Prints statistics on 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 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 their 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 Java compilation method statistics

A complete description of the `jstat` tool, including examples, can be found at:

http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstat.html

Following is an example `jstat` command which attaches to pid 27395 and takes five samples at 250 millisecond intervals. The `-gcnew` option specifies that statistics of the behavior of the new generation is output.

```
$ jstat -gcnew 27395 250 5
  S0C    S1C     S0U    S1U  TT  MTT   DSS     EC      EU     YGC    YGCT
 64.0   64.0    0.0   31.7  31   31   32.0   512.0   178.6   249   0.203
 64.0   64.0    0.0   31.7  31   31   32.0   512.0   355.5   249   0.203
 64.0   64.0   35.4    0.0   2   31   32.0   512.0    21.9   250   0.204
 64.0   64.0   35.4    0.0   2   31   32.0   512.0   245.9   250   0.204
 64.0   64.0   35.4    0.0   2   31   32.0   512.0   421.1   250   0.204
```

Following is a description of the column headings in the example:

Table 1-14 `jstat` — New Generation Statistics

Column	Description
S0C	Current survivor space 0 capacity (KB)
S1C	Current survivor space 1 capacity (KB)
S0U	Survivor space 0 utilization (KB)
S1U	Survivor space 1 utilization (KB)
TT	Tenuring threshold
MTT	Maximum tenuring threshold
DSS	Desired survivor size (KB)
EC	Current Eden space capacity (KB)
EU	Eden space utilization (KB)
YGC	Number of young generation GC events
YGCT	Young generation garbage collection time

1.16 `jstatd` (1.5 only)

The `jstatd` tool launches an RMI (remote method invocation) server that monitors the creation and termination of Java VMs and provides an interface to allow remote monitoring tools to attach to Java VMs running on the local host.

For more information, refer to the following website:

http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstatd.html

1.17 jvmstat Tools

The Java VM shipped with SDK 1.4.2 and later provides always-on instrumentation needed to support monitoring tools and utilities.

As of JDK 1.5, the following subset of jvmstat tools is included with the JDK: `jps` (formerly `jvmps`), `jstat` (formerly `jvmstat`), and `jstatd` (formerly `perfagent`). The `visualgc` tool is not included with JDK 1.5, but is instead provided in the unbundled jvmstat 3.0 distribution.

For more details, refer to the following website:

http://java.sun.com/performance/jvmstat

1.18 `-verbose:class`

The `-verbose:class` option displays information about each loaded class. It enables logging of class loading and unloading.

1.19 `-verbose:gc`

The `-verbose:gc` option enables logging of garbage collection (GC) information. It can be combined with other Java VM specific options such as `-XX:+PrintGCDetails` and `-XX:+PrintGCTimeStamps` to retrieve more 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 along with detailed information about GC analysis and tuning, are described at Sun's GC portal site:

http://java.sun.com/developer/technicalArticles/Programming/GCPortal

The `-verbose:gc` option can be dynamically enabled at runtime using the management API or JVMTI. The `jconsole` monitoring and management tool can also enable or disable this option when attached to a management Java VM.

For other GC logging options, see -Xverbosegc.

1.20 -verbose:jni

The `-verbose:jni` option enables logging of Java Native Interface (JNI). Specifically, when a JNI native method is resolved, the Java 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 may be useful when trying to diagnose issues with applications that use native libraries.

1.21 visualgc

The `visualgc` tool uses jvmstat technology to provide visualization of garbage collection activity in the Java VM. The Java VM shipped with JDK 1.4.2 and later releases provides the always-on instrumentation needed to support monitoring tools and utilities such as `visualgc`.

As of JDK 1.5, the following subset of the jvmstat tools is included with the Java VM: `jps` (formerly `jvmps`), `jstat` (formerly `jvmstat`), and `jstatd` (formerly `perfagent`). `visualgc` is not included in this set, but is instead provided in the unbundled jvmstat 3.0 distribution. The download for jvmstat 3.0 may be found at:

http://java.sun.com/performance/jvmstat

`visualgc` attaches to a running Java VM processs to collect and graphically display garbage collection, class loader, and Java compiler performance data.

The target Java VM is identified by its virtual machine identifier, or `vmid`. On HP-UX, the `vmid` is the process id of the running Java application.

For details on `visualgc` usage refer to:

http://java.sun.com/performance/jvmstat/visualgc.html

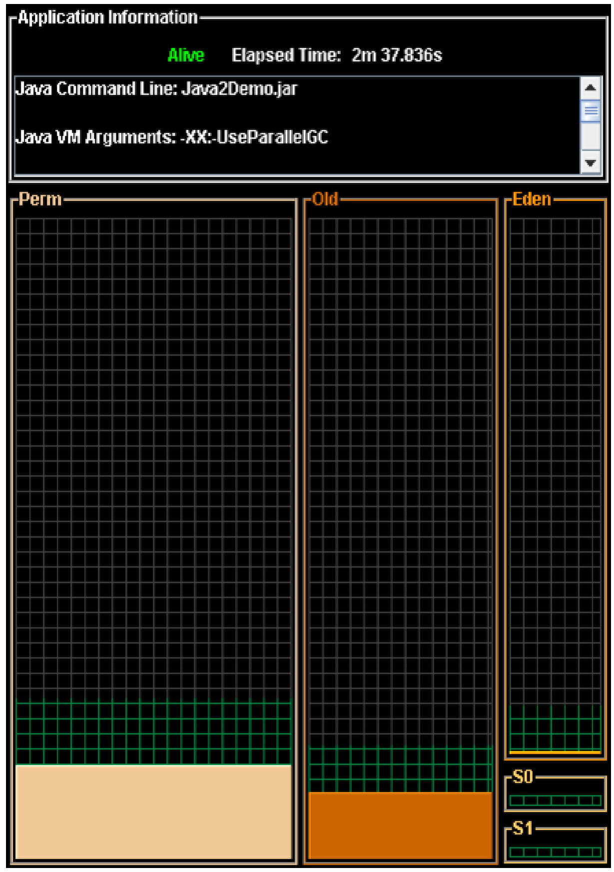
When `visualgc` is attached to a running Java VM it opens the following windows:

1. *Application Information* window
2. *Graph* window
3. *Survivor Age Histogram* window (optional)

The *Survivor Age Histogram* window is only available when Parallel Scavenge is in use (`-XX:+UseParallelGC` or `-XX:+AggressiveHeap` options).

Following is an example `visualgc` *Application Information* window and a description of the different window areas:

Figure 1-17 `visualgc` ***Application Information*** **Window**



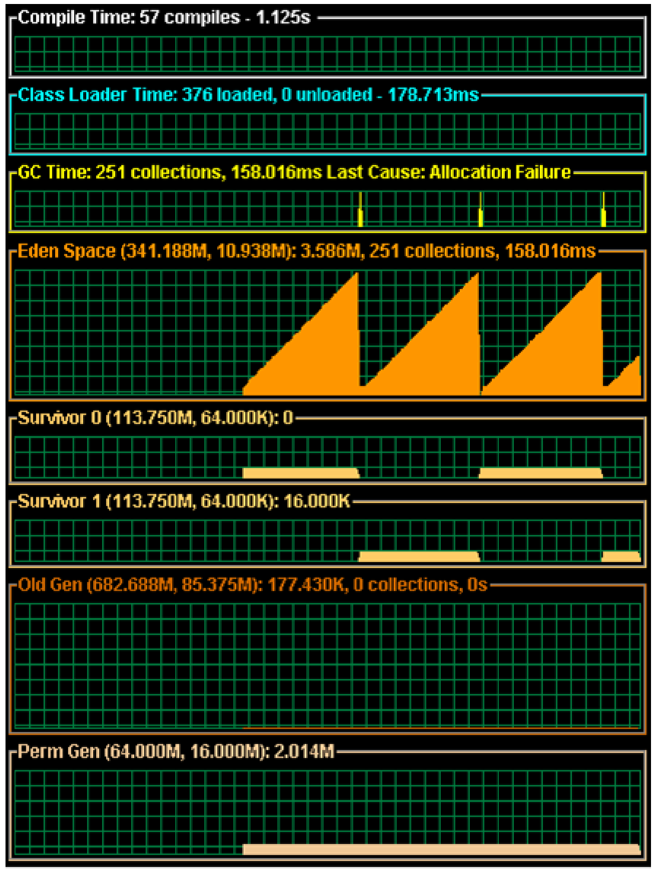
The top panel of this window is labelled *Application Information* . This panel has an Alive/Dead indicator and the elapsed time since the start of the Java application. Following this panel there is a scrollable text area that lists miscellaneous information about the configuration of the target Java application and the Java VM. This section includes main class or jar file name, the arguments to the class's `main` method, arguments passed to the Java VM, and the values of certain Java properties exported as instrumentation objects.

The bottom panel shows a graphical view of the spaces that make up the generational garbage collection system. This panel is divided into three vertical sections, one for each of the generations: the *Perm* generation, the *Old* (or Tenured) generation, and the *Young* generation. The *Young* generation is comprised of three separate spaces, the *Eden* space, and two Survivor spaces, *S0* and *S1*.

The screen areas representing the various spaces are sized in proportion to the maximum capacities of the spaces. The screen areas for the three GC generations are of fixed size and do not vary over time. Each space is filled with a unique color indicating the current utilization of the space relative to its maximum capacity. The unique color for each space is used consistently among this window and the other two `visualgc` windows (*Graph* and *Survivor Age Histogram*).

The *Graph* window displays the values of various statistics as a function of time. The resolution of the horizontal axis of the graph is determined by the `interval` command line argument, where each sample occupies two pixels of screen area. The height of each display depends on the metric being plotted. Following is an example *Graph* window:

Figure 1-18 `visualgc` ***Graph*** **Window**

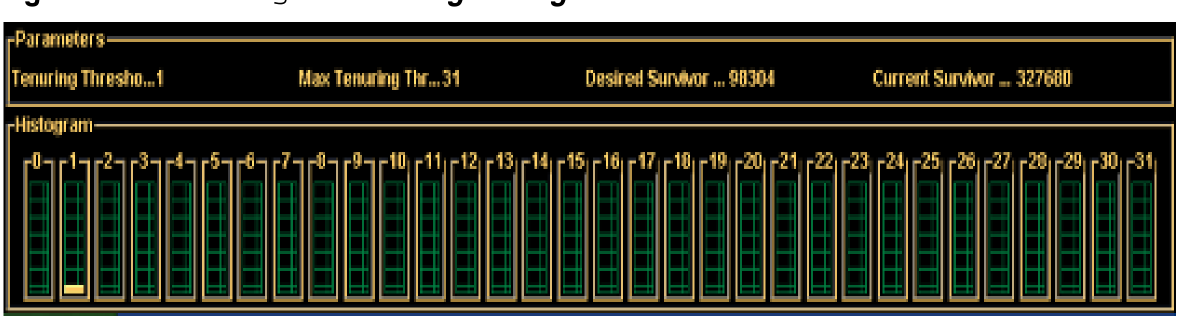


Each of the GC space graphs can be displayed in one of two modes: reserved mode or committed mode; committed mode is the default. In reserved mode, the data is scaled according to the maximum capacity of the space. The background grid is painted in dark gray to represent the uncommitted portion and in green to represent the committed portion of reserved memory. In committed mode, the data is scaled according to the current capacity of the space. The mode can be toggled by right-clicking over the space and checking or unchecking the "Show Reserved Space" check box.

The *Survivor Age Histogram* window consist of two panels, the *Parameters* panel and the *Histogram* panel. The *Parameters* panel displays the size of the survivor spaces and the parameters that control the promotion behavior of the young generation. The *Histogram* panel displays a snapshot of the age distribution of objects in the active survivor space after the last *Young* generation collection. The display is comprised of 32 identically sized regions, one for each possible object age. Each region represents 100% of the active Survivor Space and is filled with a colored area that indicates the percentage of the survivor space occupied by objects of the given age.

Following is an example *Survivor Age Histogram* window:

Figure 1-19 `visualgc` ***Survivor Age Histogram*** **Window**



When the Java VM is started with the Parallel Young GC option (`-XX:+UseParallelGC`), the *Survivor Age Histogram* window is not displayed because the Parallel Young collector does not maintain a survivor age histogram since it applies a different policy for maintaining objects in the survivor spaces.

1.22 `-Xcheck:jni`

The `-Xcheck:jni` option is useful when trying to diagnose problems with applications that use the Java Native Interface (JNI). Sometimes there are bugs in the native code that cause the Java VM to crash or behave incorrectly. Add the `-Xcheck:jni` option to the command line when starting the application. For example:

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

The `-Xcheck:jni` tells the Java VM to do additional validation on the arguments passed to JNI functions. This option may not find all invalid arguments or diagnose logic bugs in the application code; however, it can help diagnose these types of problems.

When an invalid argument is detected, the Java VM prints a message to the application console (standard output), prints the stack trace of the offending thread, and aborts the Java VM. Following is an example where a NULL is incorrectly passed to a JNI function that does not allow NULL:

```
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)
```

Following is another example of output that is displayed when something other than a jfieldID is provided to a JNI function that expects a jfieldID:

```
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)
```

Following are some types of problems that `-Xcheck:jni` can help diagnose:

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

In general, all errors detected by `-Xcheck:jni` are fatal; the error is printed and the Java VM is aborted. One exception to this is a non-fatal warning that is printed when a JNI call is made within a JNI critical region. This is the warning that is displayed when this happens:

```
Warning: Calling other JNI functions in the scope of
Get/ReleasePrimitiveArrayCritical or Get/ReleaseStringCritical
```

A JNI critical region arises when native code uses the JNI `GetPrimitiveArrayCritical()` or `GetStringCritical()` functions 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 time between the get and release is called a JNI critical section, and during that time the Java VM cannot reach a state that allows garbage collection to occur. The general recommendation is that other JNI functions should not be used when in a JNI critical section, and in particular any JNI function that blocks could potentially cause a deadlock. The warning printed by `-Xcheck:jni` is an indication of a potential issue; it does not always indicate an application bug.

1.23 -Xverbosegc

The `-Xverbosegc` option prints out detailed information about the Java heap before and after garbage collection. The syntax is:

```
-Xverbosegc [:help] | [0 | 1] [:file = [stdout | stderr | <filename>]]
```

The "`:help`" option prints a description of the verbosegc output format.

The "`0 | 1`" option controls the printing of help information. Specifying value "0" will cause the heap information to be printed after every Old Generation GC or Full GC. Specifying value "1" (the default) will cause the heap information to be printed after every GC.

The "`file = [stdout | stderr | <filename>]`" option specifies the output file. The default is `stderr`, which directs the output to the standard error stream. Alternative choices for the output file are `stdout` and a user-specified filename.

At every garbage collection, 20 fields are printed as follows:

```
GC: %1 %2 %3 %4 %5 %6 %7 %8 %9 %10 %11 %12 %13 %14 %15 %16 %17 %18 %19 %20
```

The following table contains brief descriptions of these 20 fields:

Table 1-15 Garbage Collection Field Information

Field	Information in Field
1	Type of GC: • 1: Scavenge (GC of New Generation only) • 2: Old Generation GC or a Full GC • 3: Complete background CMS GC • 4: Incomplete background CMS GC • 11: Ongoing CMS GC
2	Additional information based on GC type in field 1.
3	Program time at the beginning of the GC, in seconds.
4	GC invocation. Counts of background CMS GCs and other GCs are maintained separately.
5	Size of the object allocation request that forced the GC, in bytes.
6	Tenuring threshold—determines how long the newborn object remains in the New Generation.
7	Eden Sub-space (within the New Generation) occupied before GC.
8	Eden Sub-space (within the New Generation) occupied after GC.
9	Eden Sub-space (within the New Generation) current capacity.
10	Survivor Sub-space (within the New Generation) occupied before GC.
11	Survivor Sub-space (within the New Generation) occupied after GC.
12	Survivor Sub-space (within the New Generation) current capacity.
13	Old Generation occupied before GC.

Table 1-15 Garbage Collection Field Information *(continued)*

Field	Information in Field
14	Old Generation occupied after GC.
15	Old Generation current capacity.
16	Permanent Generation (storage of Reflective Objects) occupied before GC.
17	Permanent Generation (storage of Reflective Objects) occupied after GC.
18	Permanent Generation (storage of Reflective Objects) current capacity.
19	The total stop-the-world duration, in seconds.
20	The total time used in collection, in seconds.

For more details about these fields, use the `:help` option or refer to the Java Programmers Guide at the following website:

http://www.hp.com/products1/unix/java/infolibrary/prog_guide/index.html

To better understand how garbage collection works in the Java VM, read the article "Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1" (November 2002) by Nagendra Nagarajayya and J. Steven Mayer at the following website:

http://developers.sun.com/techtopics/mobility/midp/articles/garbagecollection2/#17.1

Additionally, HP recommends using the HPjtune tool, which can display graphically the information contained in a -Xverbosegc log. Refer to the HPjtune command for more information.

1.24 `-XX:+HeapDump` and `_JAVA_HEAPDUMP` Environment Variable

The `-XX:+HeapDump` option can be used to observe memory allocation in a running Java application by taking snapshots of the heap over time. Another way to get heap dumps is to use the `_JAVA_HEAPDUMP` environment variable; setting this environment variable allows memory snapshots to be taken without making any modifications to the Java command line. In order to enable this functionality, either use the command-line option or set the environment variable (for example, `export _JAVA_HEAPDUMP=1`) before starting the Java application. This option is available beginning with SDK 1.4.2.10 and JDK 1.5.0.03.

The output is similar to that produced by the `-Xrunhprof:heap=dump` option except that the thread and trace information is not printed to the output file.

With the `-XX:+HeapDump` option enabled, each time the process is sent a SIGQUIT signal, the Java VM produces a snapshot of the Java heap in hprof ASCII format. The name of the file has the following format: `java_<pid>_<date>_<time>_heapDump.hprof.txt`.

If `_JAVA_HEAPDUMP_ONLY` is set, then heap dumps are triggered by SIGVTALRM instead of SIGQUIT for this option. Only the heap dump is produced; that is, the thread and trace dump of the application to `stdout` is suppressed. Setting the `_JAVA_BINARY_HEAPDUMP` environment variable along with `_JAVA_HEAPDUMP_ONLY` produces a binary format heap dump when the SIGVTALRM is sent to the process instead of an ASCII one.

NOTE: A full GC is executed prior to taking the heap snapshot.

1.24.1 Other HeapDump Options

In addition to `-XX:+HeapDump`, there are three other HeapDump options available: -XX:+HeapDumpOnCtrlBreak , -XX:+HeapDumpOnOutOfMemoryError, and -XX:+HeapDumpOnly. Following is a table describing the four heap dump options. Additional information on these three heap dump options is provided following the table.

Table 1-16 Overview of HeapDump Options

Option	Trigger	hprof Format	Filename
-XX:+HeapDump	SIGQUIT	ASCII; set the _JAVA_BINARY_HEAPDUMP environment variable to get binary	java_<pid>_<date>_<time>_heapDump.hprof.txt
-XX:+HeapDumpOnCtrlBreak	SIGQUIT	Binary	java_<pid>.hprof.<millitime>
-XX:+HeapDumpOnOutOfMemoryError	Out of Memory	Binary	java_<pid>.hprof.<millitime> or the file specified by -XX:HeapDumpPath=file
-XX:+HeapDumpOnly	SIGVTALRM	ASCII; set the _JAVA_BINARY_HEAPDUMP environment variable to get binary	java_<pid>_<date>_<time>_heapDump.hprof.txt

1.24.2 -XX:+HeapDumpOnCtrlBreak

The -XX:+HeapDumpOnCtrlBreak option is available beginning with SDK 1.4.2.11 and JDK 1.5.0.05. It enables the ability to take snapshots of the Java heap when a SIGQUIT signal is sent to the Java process without using the JVMTI-based -Xrunhprof:heap=dump option. This option is similar to -XX:+HeapDump except the output format is in binary hprof format and the output is placed into a filename with the following naming convention: java_<pid>.hprof.<millitime>.

If the HP environment variable _JAVA_HEAPDUMP is set and this option is specified, then both hprof ASCII and binary dump files are created when a SIGQUIT is sent to the process. For example, the following file names are created: java_27298.hprof.1152743593943 and java_27298_060712_153313_heapDump.hprof.txt.

If JAVA_BINARY_HEAPDUMP is set and the -Xrunhprof:heap=dump command is given, then both hprof ASCII and binary files are produced for this option.

1.24.3 -XX:+HeapDumpOnOutOfMemoryError

The -XX:+HeapDumpOnOutOfMemoryError option is available beginning with SDK 1.4.2.11 and JDK 1.5.0.04. This option enables dumping of the Java heap when an "Out Of Memory" error condition occurs in the Java VM. The heap dump file name defaults to java_pid<pid>.hprof in the current working directory. The option -XX:HeapDumpPath=file may be used to specify the heap dump file name or a directory where the heap dump file should be created. The only heap dump format generated by the -XX:+HeapDumpOnOutOfMemoryError option is the hprof binary format.

One known issue exists: the -XX:+HeapDumpOnOutOfMemoryError option does not work with the low-pause collector (option -XX:+UseConcMarkSweepGC).

1.24.4 -XX:+HeapDumpOnly

Starting with SDK 1.4.2.11 and JDK 1.5.0.05, the -XX:+HeapDumpOnly option or the _JAVA_HEAPDUMP_ONLY environment variable can be used to enable heap dumps using the SIGVTALRM signal (signal 20). This interface is provided to separate the generation of thread and trace information triggered via SIGQUIT from the heap dump information. If the -XX:+HeapDumpOnly option is specified or the _JAVA_HEAPDUMP_ONLY environment variable is set, then the heap dump functionality is triggered by sending SIGVTALRM to the process. The printing of thread and trace information to stdout is suppressed.

The heap dump is written to a file with the following filename format: java_<pid>_<date>_<time>_heapDump.hprof.txt.

The default output format is ASCII. The output format can be changed to hprof binary format by setting the `_JAVA_BINARY_HEAPDUMP` environment variable. This environment variable can also be used with the `-XX:+HeapDump` option to generate hprof binary format with the SIGQUIT signal.

1.24.5 Using Heap Dumps to Monitor Memory Usage

By creating a series of heap dump snapshots, you can see how the number and size of objects varies over time. It is a good idea to collect at least three snapshots. The first one serves as a baseline. It should be taken after the application has finished initializing and has been running for a short time. The second snapshot should be taken after the residual heap size has grown significantly. Monitor this using -Xverbosegc and HPjtune. Try to take the last snapshot just before the heap has grown to a point where it causes problems resulting in the application spending the majority of its time doing full GCs. If you take other snapshots, spread them out evenly based on residual heap size throughout the running of the application.

Once you have collected the snapshots, read them into HPjmeter (run with -Xverbosegc to monitor memory usage). Use small heap sizes so that the analysis with `HPjmeter` requires less memory. Read two files in and compare them using the `File->Compare` option. You should be able to find out the types of objects that are accumulating in the Java heap. Select a type using the `Mark to Find` option and go back to a view of one of the snapshots. Go to the `Metric->Call Graph Tree` option and do a `Find`. You should be able to see the context of the object retention.

1.25 `-XX:OnError`

When a fatal error occurs, the Java VM 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 each 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 “%”.

Following is an example showing 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
```

Following is an example that launches `gdb` when an unexpected error is encountered. Once launched, `gdb` attaches to the Java VM process:

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

1.26 `-XX:+ShowMessageBoxOnError`

In addition to the `-XX:OnError` option, the Java VM can also be provided with the option `-XX:+ShowMessageBoxOnError`. When this option is set and a fatal error is encountered, the Java VM outputs information about the fatal error and ask the user if the debugger should be launched. The output and prompt are sent to the application console (standard input and standard output). Following is an example:

```
========================================================================
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 has occurred and the user is prompted whether to launch the debugger to attach to the process. If the user enters “y” or “yes” then `gdb` is launched.

In the previous example, the output includes the process id (10791) and also the thread id (1026). If the debugger is launched then one of the initial steps taken in the debugger should be to select the thread and obtain its stack trace.

While waiting for a response from the process, it is possible to use other tools to obtain a crash dump or query the state of the process.

Generally, `-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.

2 Useful System Tools for Java Troubleshooting

This chapter contains information about some system tools available on HP-UX that are useful when troubleshooting Java application problems. The tools discussed include: GlancePlus, tusc, Prospect, HP Caliper, sar, vmstat, iostat, swapinfo, top, netstat, and others.

2.1 GlancePlus

GlancePlus is a system performance monitoring and diagnostic tool. It lets you easily examine system activities, identify and resolve performance bottlenecks, and tune your system for more efficient operation. For more information on GlancePlus, refer to the following website:

http://www.managementsoftware.hp.com/products/gplus/index.html

2.2 `tusc`

`tusc` gives you another view into the system activity, in addition to Java stack traces, GlancePlus, and HPjmeter . It has many options, which you can display by entering the command `tusc -help`. For more information on `tusc`, refer to the following website:

http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,2894,00.html?jumpid=reg_R1002_USEN

2.3 Prospect

Prospect is a performance analysis tool. Beginning with Prospect revision 2.2.0, you can use Prospect to retrieve a profile of the compiled Java methods that the Java VM compiler creates in data space. In order to activate this functionality, you must have SDK 1.3.1.02 or following releases. For more information on the Prospect performance analysis tool, refer to the following website:

http://h21007.www2.hp.com/dspp/tech/tech_TechSoftwareDetailPage_IDX/1,1703,3282,00.html

2.4 HP Caliper

HP Caliper is a general-purpose performance analysis tool for applications running on Integrity systems. It helps you understand the execution of your applications and identify ways to improve their performance. For more information on the HP Caliper tool, refer to the following website:

http://h21007.www2.hp.com/dspp/tech/tech_TechSoftwareDetailPage_IDX/1,1703,1174,00.html?jumpid=reg_R1002_USEN

2.5 `sar`

The `sar` command is a tool to report various system activities, such as CPU, I/O, context switches, interrupts, page faults, and other kernel actions. For more information on this command, refer to the following website:

http://docs.hp.com/en/B2355-60127/sar.1M.html

2.6 `vmstat`

The `vmstat` command reports statistics about the process, virtual memory, trap, and CPU activity. For more information on this command, refer to the following website:

http://docs.hp.com/en/B2355-60127/vmstat.1.html

2.7 `iostat`

The `iostat` command iteratively reports I/O statistics for each active disk on the system. For more information on this command, refer to the following website:

http://docs.hp.com/en/B2355-60127/iostat.1.html

2.8 `swapinfo`

The `swapinfo` command displays information about device and file system paging space. For more information on this command, refer to the following website:

http://docs.hp.com/en/B2355-60127/swapinfo.1M.html

2.9 `top`

The `top` command displays the top processes on the system, periodically updating the information; raw CPU percentage is used to rank the processes. For more information on this command, refer to the following website:

http://docs.hp.com/en/B2355-60127/top.1.html

2.10 `netstat`

The `netstat` command displays statistics for network interfaces and protocols as well as the contents of various network-related data structures. It can show packet traffic, connections, error rates, and more. For more information on this command, refer to the following website:

http://docs.hp.com/en/B2355-60127/netstat.1.html

2.11 Other Tools

The Developer and Solution Partner Program's (DSPP) technical information website contains links to debugging information. There are links from this page to other websites containing technical papers, tips, tutorials, and more. To review this information, refer to the following website:

http://h21007.www2.hp.com/dspp/topic/topic_DetailSubHeadPage_IDX/1,4946,0-10301-TECHDOCUMENT,00.html

3 Getting Help from Hewlett-Packard

Sometimes you need help troubleshooting your Java application problems. Before opening a support call, search for information that may help you by referring to the Go Java! website:

http://www.hp.com/go/java

This site contains much information about Java, including known issues, release notes, patches, downloads, documentation, and more. If you still need troubleshooting help after looking at this website and you have a support contract with Hewlett-Packard (HP), follow the instructions outlined in this chapter to collect the necessary information before opening a support call.

3.1 Problem Report Checklist

Use this checklist to collect information before you request support. Providing more information when you initiate your support call reduces the time it takes for support engineers to start working on your problem.

1. Problem Description
   a. Did this Java application ever work?
   b. What is the problem (abort, hang, performance, and so on)?
   c. What messages are written to `stdout` or `stderr` relating to the problem?
   d. Does the problem occur every time the application is run or intermittently?
   e. What are the application details? Include the following:
      - Name of the application.
      - What the application does.
      - The command line and options used to start the application.
      - Description of the expected behavior.
      - Description of the actual behavior.
      - The application stack that you are running—for example, the webserver name or the application server name.
   f. How do you reproduce the problem? Provide source code and step-by-step instructions.
   g. Do you have a workaround for the problem? If so, describe it.
2. Problem Data
   a. Core file
   b. Fatal error log (`hs_err_pid<pid>.log`)
   c. Stack trace
3. System Information
   a. What version of HP-UX is on the system? Provide the output from the `uname -a` command.
   b. What patches are installed on the system?
   c. What window manager is being used? For example, Reflections X or X Windows. Or is the application running inside a browser? If so, which one?
4. Java Environment
   a. What is the version of the Java VM that is having the problem? Run the command `java -version` to retrieve this information.
   b. What are the values of the environment variables used by Java?
   c. What libraries are being loaded?
5. Contact Information
   a. Contact name
   b. Company name

c. Phone number
d. E-mail address

The following subsections provide instructions for collecting the necessary problem, system, and Java environment information. The final subsection contains instructions for packaging the files you need to send to Hewlett-Packard.

3.2 Collecting Problem Data

Three pieces of information are essential for analyzing most problems—the core file, the fatal error log, and the stack trace. Following are instructions for how to collect this information.

3.2.1 Collecting Core File Information

This section begins with a checklist to follow to make sure you can collect useful core files. It then reviews how you can generate a core file if one is not generated for you. Finally, there is a discussion about how to verify that your core file is valid.

3.2.1.1 Core File Checklist

Core files contain useful information, if they are complete. Sometimes you need to configure your system to make sure you can save complete core files. Consider the following items to ensure you can create complete core files.

1. Estimate the core file size.
2. Ensure your process can write large core files.
3. Verify you have enough free disk space.
4. Make sure the directory where the core file will reside supports a large file system. If not, write the core file to a directory that does.
5. Make sure you have the correct permissions to write core files.

Following are additional details on each of these steps.

3.2.1.1.1 Estimate Core File Size

The size of the `-Xmx` option affects the core file size. Use these rules to estimate the size of the Java core file:

- `-Xmx` is less than 1,500 MB. The core file will be less than or equal to 2 GB.
- `-Xmx` is between 1,500 and 2,400 MB. The core file will be less than or equal to 3 GB.
- `-Xmx` is greater than 2,400 MB. The core file will be less than or equal to 4 GB.

3.2.1.1.2 Ensure Process Can Write Large Core Files

Check your coredump block size to make sure it is set to unlimited using the `ulimit -a` command:

```
$ ulimit -a
time(seconds)        unlimited
file(blocks)         unlimited
data(kbytes)         4292870144
stack(kbytes)        8192
memory(kbytes)       unlimited
coredump(blocks)     4194303
```

If coredump is not set to unlimited, set it to unlimited using the `ulimit -c` command:

```
$ ulimit -c unlimited

$ ulimit -a
time(seconds)        unlimited
file(blocks)         unlimited
data(kbytes)         4292870144
```

```
stack(kbytes)          8192
memory(kbytes)         unlimited
coredump(blocks)       unlimited
```

3.2.1.1.3 Verify Amount of Disk Space

Check the amount of disk space available in the current working directory using the df -kP command:

```
$ df -kP /home/mycurrentdir
Filesystem          1024-blocks  Used   Available Capacity Mounted on
/dev/vg00/lvol5       1022152   563712   458440     56%   /home
```

3.2.1.1.4 Check If Directory Supports Large File Systems

Use the fsadm command as root to check if your directory supports large file systems. If you do not execute this command as root, you may not retrieve meaningful results. Following is an example:

```
<root>$ /usr/sbin/fsadm <mount_point>
```

Following is example output when the file system is set up to support large files and when it is not set up to support large files:

```
<root>$ /usr/sbin/fsadm /extra
fsadm: /etc/default/fs is used for determining the file system type
largefiles

<root>$ /usr/sbin/fsadm /stand
fsadm: /etc/default/fs is used for determining the file system type
nolargefiles
```

If the directory does not support large file systems, you need to write the core file to an alternate directory. Do this by setting the JAVA_CORE_DESTINATION environment variable (available starting with SDK 1.4.2) to the name of the directory and create the directory. For example:

```
$ export JAVA_CORE_DESTINATION=<alt_dir>
$ mkdir $JAVA_CORE_DESTINATION
```

Java creates a directory named core under the JAVA_CORE_DESTINATION directory where the core and hs_err_pid<pid>.log files are written. For example:

```
$ cd $JAVA_CORE_DESTINATION
$ ls
core.29757

$ ll
total 429296
-rw-------   1 test     users      219781020 Aug 29 12:33 core
-rw-rw-rw-   1 test     users           2191 Aug 29 12:33 hs_err_pid29757.log
```

3.2.1.1.5 Ensure Permissions Allow Core Files

Some Java processes run setuid; that is, a process where the effective uid or gid differs from the real uid or gid. On HP–UX 11.11 and later versions a kernel security feature prevents core file creation for these processes. Use the following command when you are logged in as the root user to enable core dumps of setuid Java processes:

```
$ echo "dump_all/W 1" | adb -w /stand/vmunix /dev/kmem
```

This capability is turned on only for the current boot.

3.2.1.2 Generating a Core File

Analyzing the core file is essential for troubleshooting problems. Core files are automatically generated for application aborts. For hung processes and performance issues, you need to generate them using gdb's dumpcore command.

The gdb dumpcore command forces the generation of a core file without killing a running process. This command causes a core file named core.<pid> to be created. The current process state is not modified when this command is issued.

Following is an example for a Java application running on an Integrity system:

```
$ echo "dumpcore\nq" > gdb_cmds
$ ps -u myuser | grep java
12290 pts/6    12:58 java
$ gdb --command=gdb_cmds -batch /opt/java1.4/bin/IA64N/java 12290
```

This generates a core file in the current directory with the name core.12290.

3.2.1.3 Verifying a Core File

Once you have successfully collected your core file, you should verify that it is complete and valid. To do this, open the core file in gdb and check the error and warning messages. If the message "<corefilename> is not a core dump: File format not recognized" is displayed when you open the file, your core file is invalid. Following is example output for a corrupt core file:

```
$ gdb lib/java core1
HP gdb 3.1.5 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 3.1.5 (based on GDB) is covered by the
GNU General Public License. Type "show copying" to see the conditions to
change it and/or distribute copies. Type "show warranty" for warranty/support.
..
"/home/sample/pics/4000069294/core1" is not a core dump: File format not recognized
```

Sometimes core files get truncated. Check for this by issuing the "what core" command. If you do not see the dld.sl version at the bottom of the what output, then the core file is truncated and is not usable. In the following example, the dld.sl version exists at the bottom of the what output so you know the core file is not truncated:

```
$ what core
 core:
some other library names and version information ...
92453-07 dld dld dld.sl B.11.48 EXP 051121
```

3.2.2 Collecting Fatal Error Log Information

When a Java application aborts, the fatal error log file (hs_err_pid<pid>.log) is generated. The contents of this file vary depending on the architecture and the Java version (for example, early Java versions generate less information in the fatal error log). Following is a summary of the type of information contained in this file:

1. The error causing the Java VM to abort, including the pc, process id, and thread id at which the error occurred. For example:

   ```
   # An unexpected error has been detected by HotSpot Virtual Machine:
   #
   #  SIGSEGV (11) at pc=7541df20, pid=25675, tid=1
   ```

2. The Java version and problematic frame. For example:

   ```
   # Java VM: Java HotSpot(TM) Server VM (1.4.2
   #                      1.4.2.10-060112-19:42-IA64N IA64 mixed mode)
   ```

```
# Problematic frame:
# j  spin.main([Ljava/lang/String;)V+5
```

3. Information about the current thread, including:
   a. the executing thread
   b. siginfo at the point of failure
   c. stack pointer and hex dump of the top of memory stack
   d. hex dump at the location of the current pc
   e. stack range and stack free space
4. Process information, including:
   a. a dump of all active threads at the time of the abort (SDK 1.4.2.04+)
   b. Java VM state (whether at safepoint or not) (SDK 1.4.2.10+)
   c. mutex state (SDK 1.4.2.10+)
   d. a summary of heap status; for example:

```
Heap
def new generation   total 5632K, used 144K [6d400000, 6da10000, 6e950000)
eden space 5056K,   2% used [6d400000, 6d424040, 6d8f0000)
from space 576K,   0% used [6d8f0000, 6d8f0000, 6d980000)
to   space 576K,   0% used [6d980000, 6d980000, 6da10000)
tenured generation   total 12480K, used 0K [6e950000, 6f580000, 71400000)
the space 12480K,   0% used [6e950000, 6e950000, 6e950200, 6f580000)
compacting perm gen  total 16384K, used 1118K [71400000, 72400000, 75400000)
the space 16384K,   6% used [71400000, 71517860, 71517a00, 72400000)
```

   e. dynamic libraries loaded by the process (SDK 1.4.2.04+)
   f. Java VM arguments (SDK 1.4.2.04+)
   g. Java-related environment variables
5. System Information. This includes operating system name, version, CPU, memory, and system load. For example:

```
OS: HPUX
uname:HP-UX B.11.23 U ia64
rlimit: STACK 98252k, CORE 2097151k, NOFILE 4096, AS infinity
load average:0.12 0.19 0.22

CPU:total 8 Processor          = McKinley
Processor features = branchlong
Memory: 4k page, physical 16743644k

vm_info: Java HotSpot(TM) Server VM (1.4.2.10-060112-19:42-IA64N)
for hp-ux-ia64 built on Jan 12 2006 20:09:37 by jinteg with aCC
```

3.2.3 Collecting Stack Trace Information

On PA-RISC systems, a stack trace is printed to `stderr` when the application aborts. On Integrity systems, branch and general register contents are printed to `stderr` when an application aborts. The stack trace (PA-RISC systems) and register contents (Integrity systems) are not printed to the `hs_err_pid<pid>.log` file; therefore, the contents of `stderr` should be captured into a file and sent to HP along with the `hs_err_pid<pid>.log`, core file, and libraries.

3.3 Collecting System Information

Along with HP-UX version information and information about which window manager is being used, it is also useful to know which patches are installed on the system. Use the `swlist` command to retrieve this list. For example:

```
$ /usr/sbin/swlist
    # Initializing...
    # Contacting target "mutant"...
    #
    # Target:  mutant:/
    #
    #
    # Bundle(s):
    #

     B3701AA     C.04.50.00     HP GlancePlus/UX Pak For HP-UX 11.23 (s800)
     B3901BA     C.11.23.03     HP C/ANSI C Developer's Bundle (S800)
     B3913DB     C.11.23.03     HP aC++ Compiler (S800)
     B6848BA     1.4.gm.46.9    Ximian GNOME 1.4 GTK+ Libraries for HP-UX
     B8465BA     A.02.00.08     HP WBEM Services for HP-UX
     B9073BA       B.11.23.07.00.00.03 HP-UX iCOD (Instant Capacity)
     BUNDLE11i     B.11.23.0409.3 Required Patch Bundle for HP-UX 11i v2 (B.11.23),
     September 2004
     Base-VXVM     B.04.10.011    Base VERITAS Volume Manager Bundle 4.1 for HP-UX
     CDE-ChineseS  B.11.23        Simplified Chinese CDE Environment
     CDE-ChineseT  B.11.23        Traditional Chinese CDE Environment
     CDE-English   B.11.23.0409   English CDE Environment
     CDE-French    B.11.23        French CDE Environment
     CDE-German    B.11.23        German CDE Environment
     CDE-Italian   B.11.23        Italian CDE Environment
     CDE-Japanese  B.11.23        Japanese CDE Environment
     CDE-Korean    B.11.23        Korean CDE Environment
     CDE-Spanish   B.11.23        Spanish CDE Environment
     CDE-Swedish   B.11.23        Swedish CDE Environment
     ...
     HPUXBaseAux   B.11.23.0512   HP-UX Base OS Auxiliary
     HPUXBaseOS    B.11.23        HP-UX Base OS
     ...
     Java15JDK     1.5.0.03.00    Java 1.5 JDK for HP-UX
```

3.4 Collecting Java Environment Information

In order to perform core file analysis, you need to collect information about some environment variables and libraries used by the failed application. The following subsections describe how to do this.

3.4.1 Environment Variables

To facilitate troubleshooting, it is important to know the value of the environment variables that can affect the behavior of Java applications (for example, `CLASSPATH`). To collect these application runtime environment variable values, run the following command under the same environment (that is, the same user) that the Java application was executed:

```
(ksh)$ env >  app_environment.txt
(csh)$ getenv > app_environment.txt
```

Include the `app_environment.txt` file when you send in your collected data files to Hewlett-Packard.

3.4.2 Libraries

In order to perform core file analysis, you must have access to libraries used by the failed application. The method used for determining which libraries were used depends on whether or not `gdb` is available on the system.

If `gdb` is not available, then locate files by either examining the `stdout` of the failed application or the `hs_err_pid<pid>.log` file. Either of these should list all the libraries used. Using this list, manually copy the files.

If `gdb` is available on the system where the failure occurred, issue `gdb`'s `packcore` command:

```
(gdb) packcore
```

This command creates a directory called packcore under the current directory, and in this directory a file called modules.tar is created. This tar file contains all libraries used by the application.

In some situations, only a core file can be obtained. In this case limited troubleshooting can take place since some crucial pieces of information are missing

There is one additional library that should be collected: libjunwind. This library is used by gdb to unwind Java bytecode frames; its routines help make stack traces more readable and understandable. Since this library is only used during debugging, it is not included in the tar file generated by getcore.

The following table shows the location of the libjunwind library for PA-RISC applications:

Table 3-1 Libjunwind Library Location for PA-RISC Systems

Application Type	libjunwind **Location**
PA1.1 applications (java -pa11)	/opt/<java_vers>/jre/lib/PA_RISC/server/libjunwind.sl
PA2.0 32–bit applications (default PA-RISC)	/opt/<java_vers>/jre/lib/PA_RISC2.0/server/libjunwind.sl
PA2.0 64–bit applications (java -d64)	/opt/<java_vers>/jre/lib/PA_RISC2.0W/server/libjunwind.sl

On Integrity systems, beginning with SDK 1.4.0.10 and JDK 1.5.0.03, there are two libjunwind libraries for each Java VM, libjunwind64.so and libunwind.so. The following table shows the location of these libraries for both 32–bit and 64–bit applications:

Table 3-2 Libjunwind Library Location for Integrity Systems

Application Type	libjunwind **Location**
32–bit applications	/opt/<java_vers>/jre/lib/IA64N/server/libjunwind*.so
64–bit applications	/opt/<java_vers>/jre/lib/IA64W/server/libjunwind*.so

3.5 Packaging Files

The core file, modules.tar file, and libjunwind library are all large, so they should be compressed to save time and disk space during their transmission. One method for compressing files is to use the Java archive tool, jar. This tool is included with all Java installations.

For example, to compress the core.7145, hs_err_pid7145.log, modules.tar, app_environment.txt, and libjunwind.sl files into file debug.jar, issue the following command:

```
jar cvf debug.jar core.7145 hp_err_pid7145.log modules.tar \
    app_environment.txt libjunwind.sl
```

Alternately, you can use compress or gzip to compress the files, and then use tar to bundle them together.

Glossary

GC Garbage collection.

gid Group id.

HotSpot VM The JDK comes with a virtual machine implementation called the Java HotSpot VM.

Java VM On HP implementations this is the same as the HotSpot VM.

JDK The Java Developer's Kit is the set of Java development tools consisting of the API classes, a Java compiler, and the Java virtual machine.

JMX Java Management Extensions technology provides the tools for building distributed, web-based, modular and dynamic solutions for managing and monitoring devices, applications, and service-driven networks.

JNI The JNI is the native programming interface for Java that is part of the JDK. It allows Java code to operate with applications and libraries written in other languages, such as C, C++, and assembly.

JRE The Java Runtime Environment provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language.

JVMTI The Java Virtual Machine Tool Interface provides both a way to inspect the state and to control the execution of applications running in the Java VM.

RMI Java Remote Invocation lets Java applications communicate across a network.

SDK The Java Software Developer's Kit is the set of Java development tools consisting of the API classes, a Java compiler, and the Java virtual machine.

setuid process A process where the effective uid or gid differs from the real uid or gid.

uid User id.

Index