Oracle® JRockit

Diagnostics and Troubleshooting Guide

Release R28

E15059-04

January 2011

This document describes how to diagnose and troubleshoot problems that may occur when you use Oracle JRockit.

ORACLE®

Oracle JRockit Diagnostics and Troubleshooting Guide, Release R28

E15059-04

Primary Author: Savija T.V.

Contents

7 Freezing JVM

8 About Crash Files

Preface

This document provides information to help you solve problems that you might encounter while running the Oracle JRockit JVM.

About This Document

This document contains the following chapters:

- Chapter 1, "Diagnostics Roadmap" outlines the steps you should follow to arrive at the best solution for your problem.
- Chapter 2, "Slow JVM Startup" describes how you can recognize and troubleshoot a slow-starting JVM.
- Chapter 3, "Long Latencies" describes how to recognize and troubleshoot long garbage collection pauses that have an adverse impact on system performance.
- Chapter 4, "Low Overall Throughput" describes how to recognize and troubleshoot when the application runs too slowly.
- Chapter 5, "Performance Degradation" describes measures you can take when your application begins to behave erratically, return incorrect results, or throw OutOfMemory exceptions.
- Chapter 6, "Crashing JVM" describes what to do when your system—the JVM or the application—stops sending signals.
- Chapter 8, "About Crash Files" provides information about the crash files that the JRockit JVM creates if it crashes.
- Chapter 7, "Freezing JVM" describes what to do when the JVM or Java application becomes unresponsive but has not crashed.
- Chapter 9, "Contacting Oracle for Support" provides the best practices to follow when you want to report a JRockit JVM problem to Oracle Support.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be

accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at http://www.oracle.com/accessibility/.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Diagnostics Roadmap

This chapter provides a roadmap to help you solve problems with the Oracle JRockit JVM.

The roadmap to solve problems with the Oracle JRockit JVM consists of the following steps:

- Step 1: Eliminate or Identify Common Causes
- Step 2: Observe the Symptoms
- Step 3: Identify and Resolve the Problem
- Step 4: Contact Oracle Support

Step 1: Eliminate or Identify Common Causes

Often, problems that you encounter while running an application on the JRockit JVM are caused by issues that can be diagnosed and solved with minimal effort. Table 1–1 provides a list of actions that you can take to quickly diagnose and solve basic JVM problems.

> **Note:** The list is not in any particular order. You can perform each action separately.

Table 1–1 Actions to Identify and Eliminate Common Causes for JRockit JVM Problems

Action	Effect or Significance
Reinstall the JRockit JVM	This action might correct problems that are caused by installation issues. You can find the relevant installation instructions in *Oracle JRockit Installation Guide*.
Install the latest patches for the application running on the JRockit JVM	This action might correct problems that are caused by issues in the application running on the JRockit JVM.
Install the latest release of the JRockit JVM	This action corrects problems that are fixed in a later release of the JRockit JVM. For information about problems fixed in the release, see the *Oracle JRockit Release Notes*.
Try to reproduce the problem on the same machine	A problem that can be reproduced every time a certain sequence of steps are performed could indicate a straightforward programming error. If, however, the problem occurs intermittently (at varying or fixed intervals), thread interaction and timing problems are more likely.

Table 1–1 (Cont.) Actions to Identify and Eliminate Common Causes for JRockit JVM

Action	Effect or Significance
Try to reproduce the problem on another machine	A problem that occurs on only one machine might be caused by the hardware (number of processors), the operating system, and the application software. For example, the visibility of a race condition in the JVM or a user Java application might be influenced by the speed at which certain operations are performed by the JVM.
Ensure that you are using a supported operating system with the latest patches installed	This action might correct problems that are caused by an unsupported operating system.
Ensure that third-party JNI code you run is of the latest version	This action might correct problems that are caused by issues in the third-party JNI code. If the JRockit JVM has crashed, you can find out the libraries (`.dll` or `.so` files) that are loaded by looking in the text crash file (`.dump`). You can then identify the updates you might need.
Turn off code optimization by using `-XnoOpt`	Optimization makes code more efficient. Turning off code optimization might help you identify problems related to optimization.

Step 2: Observe the Symptoms

When you encounter a problem, observe the application behavior carefully and record the symptoms.

Step 3: Identify and Resolve the Problem

Table 1–2 lists symptoms of common JRockit JVM problems, and points you to chapters that contain information to help you diagnose and resolve the problems. Perform the tasks described in the chapters corresponding to the problem you are trying to resolve.

Table 1–2 Symptoms of Common JRockit JVM Problems

Symptom	For Diagnostics Information, See
The JVM crashes and produces a dump.	Chapter 6, "Crashing JVM"
The JVM takes too long to start.	Chapter 2, "Slow JVM Startup"
Some transactions take too long to execute even though the overall throughput is acceptable.	Chapter 3, "Long Latencies"
The overall throughput is too low.	Chapter 4, "Low Overall Throughput"
After running well for a while, the JVM begins to perform poorly. For example: ■ The overall throughput degrades. ■ The overall throughput is unstable. ■ The JVM starts reporting wrong results. ■ The JVM throws exceptions when it should not.	Chapter 5, "Performance Degradation"
The JVM is freezing (without crashing).	Chapter 7, "Freezing JVM"

Step 4: Contact Oracle Support

If you are a licensed JRockit JVM user and are unable to resolve the problem on your own, contact Oracle Support.

For more information, see Chapter 9, "Contacting Oracle for Support".

2

Slow JVM Startup

This chapter provides information to help you diagnose and troubleshoot problems that cause the JRockit JVM to start slowly.

This chapter contains the following topics:

2.1 Possible Causes for Slow JVM Startup

An application might seem slow when it starts because,

- The application might be waiting to import files.
- A large number of methods might have to be compiled.
- There might be a problem in code optimization (especially on single-CPU machines).
- The problem might be caused by the Java application and not the JVM.
- You recently migrated from a third-party development JVM to the JRockit JVM as the production JVM.

  The JRockit JVM is a just-in-time (JIT) compiling JVM designed for long-running applications. It compiles methods into machine code when the methods are called for the first time. So the application is relatively slow at startup because numerous new methods are compiled. However, after the application starts, it runs fast. Moreover, as the application runs, the JVM optimizes frequently called methods, improving the performance further.

2.2 Diagnosing Slow JVM Startup

- Check whether the JVM compiles numerous methods at startup, by using the `-Xverbose:codegen` option. When you use this option, the following information about the methods being compiled is displayed:
  - Name of the method
  - Memory location

- Duration of the compilation process
- The total amount of time spent in compiling code since the application started.

The following is an example of the output of the `-Xverbose:codegen` option.

```
[INFO ][codegen][00004] #23 (Normal) java/lang/Object.registerNatives()V
[INFO ][codegen][00004] #23 0.113-0.114 0x0000000100011B60-0x0000000100011C6B
   0.80 ms 128KB 0 bc/s (11.60 ms 63252 bc/s)
[INFO ][codegen][00004] #24 (Normal)
   java/lang/OutOfMemoryError.<init>(Ljava/lang/String;)V
[INFO ][codegen][00004] #24 0.115-0.115 0x0000000100011C80-0x0000000100011C92
0.38
   ms 64KB 15662 bc/s (11.99 ms 61731 bc/s
```

Note that after the JIT compilation is complete, the application runs faster than during startup, because when the methods are called subsequently, the JVM runs the precompiled code.

- Check the time taken to optimize methods by using the `-Xverbose:opt` option.

2.3 Diagnosing Slow Application Startup

The startup could be slow when, for example, the application is searching for a data file, looking up data in a database, and so on.

If you suspect that the application is causing the slow startup, use the Flight Recorder tool to record and analyze the application data.

For information about the Flight Recorder tool, see the *Oracle JRockit Flight Recorder Run Time Guide*.

2.4 Measuring Timing

You can measure the timing inside the application by inserting the `System.nanoTime()` and `System.currentTimeMillis()` methods. Note that these methods consume additional resources at run time, but the performance impact should be minimal.

System.nanoTime()

This method returns a monotonic timer value by using the most precise available system timer. The returned value is in nanoseconds, however the factual resolution of the timer can vary depending on the operating system and the hardware. Note that there is no conventional zero point to which you can relate the timer value. So you must measure the time at least twice to get any meaningful data.

`nanoTime()` uses different methods on different operating systems:

- **Windows:** `QueryPerformanceCounter()`
- **Solaris**: `gethrtime()`
- **Linux**: `clock_gettime()` in `librt` if available; else `gettimeofday()`

To get information about timer resolution (and, on Linux, the method used to get a time value), start the JRockit JVM with the `-Xverbose:timing` command-line option.

The following is an example of a verbose timing report on Windows:

```
[INFO ][timing ] Fast time frequency is 1995000000hz
[INFO ][timing ] Drift is 0.00000021 = per day 0.018secs (max 300.000)
```

```
[INFO ][timing ] Hardware fast time enabled
[INFO ][timing ] Counter timer using resolution of 1995MHz
```

System.currentTimeMillis()

This method returns the current time in milliseconds. The current time is defined as the time since 00:00:00 UTC, January 1, 1970.

Milliseconds and Nanotime at Application Startup

To get the values of `System.currentTimeMillis()` and `System.nanoTime()` at the time the JVM started, use the `-Xverbose:starttime` command-line option.

The following is an example of the output for the `-Xverbose:starttime` command-line option:

```
INFO ][startti] VM start time: 1260962573921 millis 6922526 nanos
18442244770397334 ticks
```

The `millis` value is the same value that the `System.currentTimeMillis()` method would provide and the `nanos` value is the value that the `System.nanoTime()` method would provide.

2.5 Recommended Solutions for Slow JVM Startup

Tune for Faster Startup

Sometimes, the problem may be with how the JVM is tuned using command-line options.

For tips on how to tune the JVM for a faster startup, see the *Oracle JRockit Performance Tuning Guide*.

Eliminate Optimization Problems

At times, optimization could be the cause of the slow JVM startup, especially on single-CPU machines. For more information, see Section 5.2, "Troubleshoot Optimization Problems."

Eliminate Application Problems

If you determine that the slow start is due to problems in the Java application, investigate the cause of the problem from the application viewpoint. The problem is probably caused by methods that are subject to unnecessary synchronization or an insufficient number of synchronized resources. Try to locate the methods that are causing the bottleneck and, if possible, modify the code of your Java application.

Note that you can use the Flight Recorder tool to analyze synchronization problems. For more information, see the *Oracle JRockit Flight Recorder Run Time Guide*.

Contact Oracle Support

If the tuning solutions suggested in the *Oracle JRockit Performance Tuning Guide* do not help you solve the problem, contact Oracle Support as described in Chapter 9, "Contacting Oracle for Support"

3 Long Latencies

Long latencies might manifest, for example, as single transactions that time out in a transaction-based application while the overall performance is good. The problem usually is uneven performance and nondeterministic latencies.

This chapter includes the following topics:

3.1 Tune the JVM to Reduce Latency

Long latencies might indicate that the application is not tuned for short and deterministic pause times. Before engaging in time-consuming troubleshooting and mitigation tasks, try tuning the JVM to optimize for short pause times. For information about tuning the JVM for short pause times, see the *Oracle JRockit Performance Tuning Guide*.

Note that there are trade-offs exist between low latencies and high overall application throughput.

- High latencies that cause transactions to time out are often caused by pauses for garbage collection. To reduce the individual garbage-collection pause times the garbage collector runs in a **mostly concurrent** mode; that is, the garbage collection is, mostly, performed while the Java threads are still running. This causes some additional work for the garbage collector, which has to keep track of changes during the concurrent phases of the garbage collection. The garbage collections are also less efficient because objects that are allocated during the concurrent garbage collection, are not garbage collected until the next garbage collection cycle. This can force the JVM to collect garbage more frequently.
- If you have disabled or limited compaction by using the `-XXcompaction` command-line option, to reduce the pause times caused by compaction, the heap might become fragmented. (You can analyze fragmentation by using the Flight Recorder tool.)

You can increase the overall throughput while keeping the latencies low by allowing longer garbage collection pauses or by manually tuning the garbage collection. For more information, see the *Oracle JRockit Performance Tuning Guide*.

3.2 Troubleshooting Latency Issues

This section provides information to troubleshoot latency issues related mostly to concurrent garbage collection; for example, `-Xgc:deterministic` and `-Xgc:pausetime`.

3.2.1 GC Trigger Value Keeps Increasing

The garbage collection trigger (`gctrigger`) value is the amount of free heap space that should be available when a concurrent garbage collection starts, to allow the Java threads to continue allocating objects during the entire garbage collection. The `gctrigger` value changes at run time, to avoid situations where the heap becomes full during concurrent garbage collection.

Monitor the `gctrigger` value by using the output of the `-Xverbose:memdbg` option or by using the Flight Recorder tool.

- A continuously increasing `gctrigger` value indicates that the load on the application is too high for the concurrent garbage collector. Decrease the load on the application.
- A continuously increasing `gctrigger` value could also indicate that the Java heap size is too small; the behavior might improve if you increase the heap size.

3.2.2 GC Reason for Old Collections is Failed Allocations

Monitor the garbage collection reasons for the old collections by using either the `-Xverbose:memdbg` option or the Flight Recorder tool. The normal garbage collection reason for a mostly concurrent old collection is **heap too full**.

If the old collections are triggered frequently due to failed object allocation, the `gctrigger` is too low. Increase the `gctrigger` by using the `-XXgcTrigger` option or decrease the load on the application.

3.2.3 Long Young-Collection Pause Times

Monitor the pause times for young collections by using the output of the `-Xverbose:gcpause` option or through Flight Recorder recordings.

If the young-collection pause times are too long, decrease the nursery size by using the `-Xns` option; alternatively, run a single-generational garbage collector.

3.2.4 Long Pauses in Deterministic Mode

Monitor the garbage-collection pause times in a Flight Recorder recording. Check the pause parts for pause times that are too long. If the pause parts for `Compaction` are too long, decrease the pause target. If the pause parts in `Mark:Final`, especially the ones concerning `RefrenceQueues`, are too long, the problem might be due to numerous `java.lang.ref.Reference` objects in the application. Redesign the Java application using fewer reference objects. You could also try decreasing the heap size; this causes reference objects to be handled more frequently, but reduces the number of reference objects to handle at each old collection.

3.3 Contact Oracle Support

If the solutions provided in this section do not help you solve the performance degradation problem, contact Oracle Support, as described in Chapter 9, "Contacting Oracle for Support."

4

Low Overall Throughput

Low overall throughput manifests as, for example, a low score in benchmarks, too few transactions executing per minute in a transaction-based system, or long processing times for large batches of data.

Low overall throughput usually means that the JVM is not tuned to maximize application throughput. Before engaging in time-consuming troubleshooting and mitigation tasks, try tuning the JVM to optimize application throughput.

For information about tuning the JVM for optimal throughput, see the *Oracle JRockit Performance Tuning Guide*.

Note that a trade-off exists between overall application throughput and low individual latencies. A JVM that is tuned for optimal overall throughput, consumes as little CPU time as possible in garbage collection and memory management, allowing the Java application to run as much as possible. To minimize unnecessary overhead and extra work during garbage collection, the Java application should be paused for the entire duration of the garbage collection process. The individual pauses might be long; but, in the long run, the overall throughput would be maximized.

You can reduce the latencies without losing too much overall throughput by, for example, limiting the compaction or using a generational garbage collector. For more information, see the *Oracle JRockit Performance Tuning Guide*.

If the solutions provided in this section do not help you solve the performance degradation problem, contact Oracle Support, as described in Chapter 9, "Contacting Oracle for Support."

5 Performance Degradation

The JRockit JVM is designed to provide stable performance throughout an application run in large server environments. Occasionally, you might notice a degradation in performance after the application runs for a while.

This chapter provides information to help you identify and address performance degradation problems. It contains the following topics:

- Section 5.1, "Tune for Performance"
- Section 5.2, "Troubleshoot Optimization Problems"
- Section 5.4, "Contact Oracle Support"

5.1 Tune for Performance

When system performance begins to deteriorate, the problem is often with tuning. Incorrectly tuned compaction might, for example, cause the performance to degrade over time because the fragmentation of the heap increases until the garbage collector must perform a full compaction to avoid throwing an out-of-memory error.

Before engaging in time-consuming troubleshooting and mitigation tasks, try tuning the JRockit JVM for stable performance as described in the *Oracle JRockit Performance Tuning Guide*.

5.2 Troubleshoot Optimization Problems

Poor performance might be caused by optimization problems,. Such problems usually occur after the program has been running well for a while, and result in symptoms such as the following:

- The JVM crashes (for more information, see Chapter 6, "Crashing JVM").
- `NullPointerExceptions` are thrown from unexpected points in the program.
- A method returns wrong results.

Before reporting such problems, try running the JVM after disabling optimization (by using the `-XnoOpt` option).

If the performance improves after you disable optimization, you can assume that the performance degradation was caused by optimization problems. Follow the procedures for isolating and excluding the miscompiled methods as described in Section 6.5.1.3, "Exclude the Problem Method from the Optimization Process".

5.3 Troubleshoot Memory Leak Problems

A memory leak in Java causes the application to run slower over time, because the garbage collector must work harder to free memory. After a while, the JVM throws an out-of-memory error. Note that applications with a small memory leak can sometimes run for days until an out-of-memory error occurs.

To look for initial signs of a memory leak, you can take a Flight Recorder recording and check the heap usage after each old collection. A continuously rising memory usage could indicate a memory leak. For information about creating and interpreting a Flight Recorder recording, see the *Oracle JRockit Flight Recorder Run Time Guide*.

You can diagnose memory leaks by using the Memory Leak Detector, which helps you pinpoint the class that causes the memory leak. For information about using the Memory Leak Detector, see the *Oracle JRockit Mission Control Online Help*.

5.4 Contact Oracle Support

If the solutions provided in this section do not help you solve the performance degradation problem, contact Oracle Support, as described in Chapter 9, "Contacting Oracle for Support."

6 Crashing JVM

A Java application might stop running for several reasons. The most common reason is that the application finished running or was halted normally. Other reasons might be Java application errors, exceptions that cannot be handled, and irrecoverable Java errors like `OutOfMemoryError`. Occasionally, a JRockit JVM crash might occur, which means that the JVM encountered a problem from which it could not recover gracefully.

This chapter describes how to diagnose and troubleshoot JVM crashes. It includes information about the following topics:

6.1 Classify the Crash

The first step in diagnosing and resolving a JVM crash is to classify the crash; that is, try to determine where and why the crash occurred.

6.1.1 Using a Crash File

When the JRockit JVM crashes, it creates a snapshot of the state of the computer and the JVM process at the time of the crash, and writes the state information into the following crash files:

- **`.dump`** file: The `.dump` file is a text file that is like an executive summary of the complete memory image and the environment in which the JVM was running at the time of the crash. This file is produced by the JVM itself when it crashes and is useful for classifying crashes; it can also sometimes be used for identifying problems that have already been fixed. This file rarely reveals enough information to determine the cause of the problem.
- **`.core` file**: The `.core` file is a binary crash file produced on UNIX-like systems such as Linux and Solaris. By default, the `.core` file captures complete information about the entire JVM process at the time of the crash.
- **`.mdmp` file** (or a minidump file): The `.mdmp` file is the Windows equivalent of the `.core` file described earlier.

If you have a support agreement with Oracle, you can contact Oracle Support for troubleshooting JRockit JVM problems. The binary crash file (`.core` or `.mdmp`) is essential when you contact Oracle Support.

For more information about crash files, see Chapter 8, "About Crash Files."

6.1.2 Determine the Crash Type

Table 6–1 lists the symptoms you can look for and the probable crash types corresponding to those symptoms.

Table 6–1 Crash Symptoms and Crash Types

Symptom	For Troubleshooting Information, See
The `.dump` file indicates that the JVM process has run out of virtual memory.	Section 6.2, "Out-Of-Virtual-Memory Crash"
The size of the `.core` file (or `.mdmp`) file is close to the maximum virtual memory of the process on the operating system.	Section 6.2, "Out-Of-Virtual-Memory Crash"
The `.dump` file indicates that stack overflow errors have occurred.	Section 6.3, "Stack Overflow Crash"
On Linux only: The `.dump` file indicates that the `LD_ASSUME_KERNEL` environment variable is set.	Section 6.4, "Crash Caused by Unsupported Linux Configuration"
On Linux only: You are using a nonstandard or unsupported Linux configuration.	Section 6.4, "Crash Caused by Unsupported Linux Configuration"
You observe symptoms other than those listed in this table.	Section 6.5, "JVM Crash"

6.2 Out-Of-Virtual-Memory Crash

The JVM reserves virtual memory for many purposes; for example, the Java heap, Java methods, thread stacks, and JVM-internal data structures. In addition, native (JNI) code can also allocate memory. The process size consists of all the memory reserved by the JVM and any third-party libraries running inside the process, and is subject to operating system limitations.

If the virtual memory allocation of the JVM process exceeds the operating system limitations, the JVM runs out of virtual memory, which may cause it to crash.

This section contains the following topics:

- Section 6.2.1, "Verify the Out-Of-Virtual-Memory Error"
- Section 6.2.2, "Troubleshoot the Out-Of-Virtual-Memory Error"

6.2.1 Verify the Out-Of-Virtual-Memory Error

Before you start troubleshooting an out-of-virtual-memory error, you must verify that the error is indeed due to the JVM process running out of virtual memory.

Table 6–2 shows the maximum virtual memory available to a single process on various 32-bit operating systems. Virtual memory is practically unlimited on 64-bit platforms.

Table 6–2 Approximate Maximum Virtual Memory Available to IA32 Architectures

Operating System	Approximate Maximum Virtual Memory
Windows	2 GB
Windows /3GB Startup Option	3 GB
Linux (normally)	3 GB

You can verify whether an out-of-virtual-memory error has occurred in the following ways:

- **Look in the text crash file.**

  The text crash file `.dump` might indicate that memory allocations have failed. This is a strong indication that the JRockit JVM process has run out of virtual memory. For more information about the text crash file, see Chapter 8, "About Crash Files."

- **Check the size of the binary crash file.**

  When the JRockit JVM crashes, it generates a binary crash file (`.core` or `.mdmp`). By default, the binary crash file contains a copy of the entire JVM process.

  Check the size of the binary crash file to determine whether the JVM process has indeed run out of virtual memory.

  - Verify that the binary crash file size has not been limited with the command-line option `-XXdumpSize` or with the operating system command `ulimit` (Linux and Solaris only). Use the command `ulimit -a` to verify that the crash file size is unlimited on Linux and Solaris. If the size of the binary crash file has been limited, you cannot use it to verify that the JVM process has run out of virtual memory.
  - Verify whether the size of the binary crash file is larger than the size of the heap. This is a sanity check to ensure that the binary crash file has not been truncated (due to limited disk space, for example).
  - Check whether the size of the binary crash file is close to the maximum process size allowed by the operating system (see Table 6–2).

6.2.2 Troubleshoot the Out-Of-Virtual-Memory Error

After verifying that the JVM process has run out of virtual memory, you can start troubleshooting the problem as described in this section.

- Section 6.2.2.1, "Upgrade to the Latest JRockit JVM Release"
- Section 6.2.2.2, "Reduce the Java Heap Size"
- Section 6.2.2.3, "Use the Windows /3GB Startup Option"
- Section 6.2.2.4, "Check for Memory Leaks in JNI Code"
- Section 6.2.2.5, "Record Virtual Memory Usage"
- Section 6.2.2.6, "Contact Oracle Support"

6.2.2.1 Upgrade to the Latest JRockit JVM Release

Ensure that you run the latest JRockit JVM release. The memory usage problems that are causing the JVM crash might have been fixed in the latest JRockit JVM release.

6.2.2.2 Reduce the Java Heap Size

The Java heap is only a part of the total memory usage of the JVM. If the Java heap is too large, the JVM may fail to start or run out of virtual memory when Java methods are compiled and optimized or when native libraries are loaded. If this happens, you should try lowering the maximum heap size.

6.2.2.3 Use the Windows /3GB Startup Option

On Windows 2000 Advanced Server and Datacenter, Windows 2003, Windows XP, and subsequent Windows versions, you have the option of starting the operating system with the /3GB option (by specifying the option in the `BOOT.INI` file). This option changes the maximum virtual memory process size from 2 GB to 3 GB.

6.2.2.4 Check for Memory Leaks in JNI Code

Check the JNI code you are using for memory leaks. Incorrectly written or used JNI code might be causing memory leaks, which results in the Java process growing until it reaches the maximum virtual memory size on the platform.

6.2.2.5 Record Virtual Memory Usage

Recording the virtual memory usage can help in diagnosing the out-of-virtual-memory problem.

This section describes how you can collect virtual memory usage statistics on Windows and Linux.

Recording Virtual Memory Usage on Windows

Use the Windows tool **perfmon** to record the `PrivateBytes` process counter. Collect information about the amount of reserved virtual memory for the JVM process. To do this:

1. Open **Performance Monitor**, which you can find in the administrative tools.
2. Click **+** to open the **Add Counters** dialog box.
3. In the **Performance Object** list, select **Process**.
4. From the **Process** list, select the **Private Bytes** counter.
5. Select the process to monitor and click **Add**.

Recording Virtual Memory Usage on Linux

Create a script that records the virtual memory usage at regular intervals.

For example: `top -b -n 10 > virtualmemory.log`

This script runs `top` every ten seconds and puts the resulting data in the file, `virtualmemory.log`.

The virtual memory usage for all the running processes can be found in the `VIRT` column in the log file. To view only the current status, type `top` and press `[Shift]-[M]` to sort the output by memory usage. This usually puts the JVM processes at the top of the output.

Creating a recording such as `virtualmemory.log` can be useful as it enables you to check whether the JRockit JVM process is actually growing and provide evidence about the growth to Oracle Support.

6.2.2.6 Contact Oracle Support

If the solutions provided in this section do not help you resolve the problem, contact Oracle Support, as described in Chapter 9, "Contacting Oracle for Support."

6.3 Stack Overflow Crash

A stack overflow crash occurs when the JRockit JVM cannot handle a stack overflow error gracefully. According to the J2SE Javadoc, a gracefully handled `java.lang.StackOverflowError` is a `java.lang.VirtualMachineError` thrown to indicate that the JVM is broken or has run out of resources necessary for it to continue operating.

For more information, see the following J2SE `java.lang` Javadocs:

- Java SE 6:
  - Class `StackOverflowError`

    `http://java.sun.com/javase/6/docs/api/java/lang/StackOverflowError.html`
  - Class `VirtualMachineError`

    `http://java.sun.com/javase/6/docs/api/java/lang/VirtualMachineError.html`
- J2SE 5.0:
  - Class `StackOverflowError`

    `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/StackOverflowError.html`
  - Class `VirtualMachineError`

    `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/VirtualMachineError.html`

The JRockit JVM `.dump` file includes information about the number of stack overflow errors thrown.

6.3.1 Verify the Stack Overflow Crash

When the JRockit JVM crashes due to stack overflow, the text crash file (`.dump`) shows `Error Message: Stack overflow` near the top of the file. Other indications might be an extremely long stack trace in the crash file or no stack trace at all. If the `.dump` file shows something like `StackOverFlow: 2 StackOverFlowErrors occurred`, this is an indication that the crash might be triggered by a previous stack overflow problem.

6.3.2 Troubleshoot a Stack Overflow Crash

This section describes possible solutions for stack overflow errors.

6.3.2.1 Application Level Changes

Often, a stack overflow error is caused by the application being coded to require stack space that exceeds the memory limits of the JRockit JVM.

Examine the stack trace in the `.dump` file to determine whether the Java code can be changed to use less stack space. For example, the application code might contain recursive method calls. Deep recursions can cause StackOverflow errors.

6.3.2.2 Increase the Default Stack Size

If it is not possible to change the stack requirements of the application, you can change the thread stack size by using the `-Xss` command-line option.

6.3.2.3 Make the JRockit JVM More Robust Against Stack Overflow Errors

The `-XX:+CheckStacks` command-line option makes the JRockit JVM more robust against stack overflow errors. It usually prevents the JVM from dumping and throwing a `java.lang.StackOverflowError`.

Note that the `-XX:+CheckStacks` command-line option results in a slight performance penalty because the JVM touches pages on the stack.

6.3.2.4 Contact Oracle Support

If the solutions provided in this section do not help you resolve the problem, contact Oracle Support, as described in Chapter 9, "Contacting Oracle for Support."

6.4 Crash Caused by Unsupported Linux Configuration

If your application crashes when you run the JRockit JVM on Linux, even if the stack trace indicates a reason for the crash, you should ensure that you run the JRockit JVM on a supported Linux configuration.

Do the following:

- **Verify whether the version of your operating system is supported for the JRockit JVM**.

  For more information, see *Oracle JRockit JDK Supported Configurations* at `http://www.oracle.com/technology/software/products/ias/files/fusion_certification.html`.

- **Verify whether the correct glibc binary is installed**.

  Linux on IA32 must be configured to use the `glibc` compiled for the i686 architecture; otherwise, the JRockit JVM might crash.

  You can check the version of the `glibc` that is installed by running the following command:

  ```
  rpm -q --queryformat '\n%{NAME} %{VERSION} %{RELEASE} %{ARCH}\n' glibc
  ```

  If the output shows something like `glibc 2.3.4 2.25 i386`, you are using an unsupported `glibc` binary.

  Upgrade `glibc` to a version that is compiled for the i686 architecture.

- **Examine the thread library**.

  If you have a `.core` file in `gdb`, you can get a hint about the thread library you are using by running the command, `info shared`.

  Look at the path of the loaded `libpthreadx.so` file.

  If the file is in `/lib/`, check the result of the `rpm` command. If the output shows `i386`, you are using an unsupported `glibc`.

  Upgrade `glibc` to a version that is compiled for the i686 architecture.

If solutions provided in this section do not help you resolve the problem, contact Oracle Support, as described in Chapter 9, "Contacting Oracle for Support."

6.5 JVM Crash

A JVM crash could be caused by a programming error in the JRockit JVM or by errors in third-party library code.

Identifying and troubleshooting a JVM crash can help you find a temporary workaround until the problem is solved in the JRockit JVM. This may also help Oracle Support to identify and fix the problem faster.

The process of diagnosing and troubleshoot JVM crashes varies depending on whether the crash occurred during code generation or garbage collection.

6.5.1 Crash During Code Generation

This section describes how to identify and troubleshoot a JVM crash that occurs during code generation. It contains the following topics:

6.5.1.1 Identify the Method that Might Have Caused the Code-Generation Crash

If the JVM crashed while generating code, the most common cause could be a miscompiled method. If the JRockit JVM miscompiles a method, either the JVM crashes or the behavior of the method is different from the expected behavior.

The text crash file identifies the method that was being compiled at the time of the crash. The method is identified in a line that starts with `Method`, located near the beginning of the `.dump` file.

6.5.1.2 Verify Whether the Crash is Due to Optimization Problems

The JRockit JVM might miscompile methods due to a problem in the optimizing compiler.

To determine whether optimization is the cause of the crash, disable optimization by specifying the `-XnoOpt` command-line option and then restart the application, as shown in the following example:

```
java -XnoOpt myApplication
```

If the JRockit JVM runs the application as expected, the crash is caused by code optimization.

6.5.1.3 Exclude the Problem Method from the Optimization Process

The JRockit JVM might miscompile methods due to a problem in the optimizing compiler.

If the JRockit JVM stopped crashing after you disabled global optimization, you can exclude only the offending methods (that you identified earlier) from the optimization process by using optimization directives. If the application runs successfully without the offending method being optimized, this workaround should solve the problem.

Specifying Optimization Directives

You can control how the JRockit JVM optimizes code by specifying directives in a control file and then specifying the control file by using the `-XX:OptFile=`*`filename`* command-line option.

> **Note:** `-XX:OptFile` is an internal diagnostic option. So to use it, you should include `-XX:+UnlockDiagnosticVMOptions` as well on the command line, **before** `-XX:OptFile`.

Example 6–1 is for a control file containing a single optimization directive.

Example 6–1 Directive to Perform Optimized Compilation for Specific Methods

```
{
  match: [ "java/lang/FloatingDecimal.dtoa*", "java/lang/Object.*"],
  jit: { preset : opt, } ,
}
```

The directive in Example 6–1 directs the JRockit JVM to optimize the methods matching the patterns specified by the `match` keyword when the JVM compiles the method for the first time.

Example 6–2 Directives to Restrict Optimization to Specific Methods

```
[
   {
      match: ["java/lang/FloatingDecimal.dtoa*", "java/lang/Object.*"],
      jit: { preset : opt, } ,
   },
   {
      match: "*",
      hotspot : { enable : false },
   }
]
```

The directive in Example 6–2 directs the JRockit JVM to allow optimization (`hotspot`) for only the `java/lang/FloatingDecimal.dtoa*` and `java/lang/Object.*` methods.

Example 6–3 Directive to Control Method Inlining

```
{
   match: "java.lang.*",
   inline: ["+java.util.*", "-com.sun.*", "sun.*" ],
}
```

The directive in Example 6–3 directs the JRockit JVM to do the following:

- Allow inlining of method calls from `java.lang.*` to `java.util.*`
- Forbid inlining of method calls from `java.lang.*` to `com.sun.*`
- Allow, if applicable, inlining of method calls from `java.lang.*` to `sun.*`

Enabling Optimization Directives

After creating the control file containing the required optimization directives, specify the pathname of control file by using the `-XX:OptFile=`*`filename`* command-line option.

Note: `-XX:OptFile` is an internal diagnostic option. So to use it, you should include `-XX:+UnlockDiagnosticVMOptions` as well on the command line, **before** `-XX:OptFile`.

Verifying the Behavior of the Optimization Directives

To verify whether the directives work as expected, use the `-Xverbose:opt` command-line option and check the output; the methods that you excluded from the optimization process should not appear in the output. For more information about `-Xverbose:opt`, see the *Oracle JRockit Command-Line Reference*.

Guidelines for Creating Optimization Directives

- Use the `match` keyword to specify the patterns of classes and methods to which the directive must be applied. The value that you specify can be either a string containing a single pattern or an array of patterns.
- Use the `jit` keyword to specify any task for the JVM to perform when it compiles a method for the first time.
- Use the `inline` keyword to specify method calls that should be inlined.
- Enclose each directive in a pair of braces.
- The control file can contain multiple directives, in which case:
  - Adjacent directives must be separated by a comma.
  - All of the directives must be enclosed in an array.

6.5.1.4 Check Whether the Problem is Caused by an External Instrumentation Tool

If you determine that a miscompiled method is not the reason for the JVM crash, investigate whether any external instrumentation tool you are using (for example JProbe or OptimizeIt) is causing the problem. These tools can alter bytecode, which can cause unexpected behavior.

To eliminate the possibility of the problem being caused by external instrumentation tools, disable the tools and then run the application.

- If the JVM continues to crash after disabling the external tool, the problem is not caused by that tool.
- If the application runs as expected, consider using a different tool or running without the tool.

6.5.1.5 Contact Oracle Support

If solutions provided in this section do not help you resolve the problem, contact Oracle Support, as described in Chapter 9, "Contacting Oracle for Support."

For code generation crashes, you must provide the following data to Oracle Support:

- The `.core` or `.mdmp` file
- The `.dump` file
- The `.class` file containing the method that was being generated
- Source code for the class containing the method that was being generated

6.5.2 Crash During Garbage Collection

This section describes how to identify and troubleshoot a JVM crash that occurs during garbage collection. It contains the following topics:

- Section 6.5.2.1, "Identify the Garbage Collection Crash"
- Section 6.5.2.2, "Upgrade to the Latest Release of the JRockit JVM"
- Section 6.5.2.3, "Try the Following Workarounds"
- Section 6.5.2.4, "Contact Oracle Support"

6.5.2.1 Identify the Garbage Collection Crash

You can identify a garbage collection crash by looking at the stack trace in the text crash file (`.dump`).

If garbage collection functions are shown in the stack trace or if the thread that caused the crash is a garbage collection thread, the crash probably occurred during garbage collection.

Garbage collection functions in the stack trace are identified by prefixes such as `mm`, `gc`, `yc`, and `oc`.

6.5.2.2 Upgrade to the Latest Release of the JRockit JVM

The problem might have been fixed in the latest release of the JRockit JVM. Upgrade to the latest release and check whether the problem continues.

6.5.2.3 Try the Following Workarounds

- Change the Garbage Collector
- Disable Compaction
- Disable Inlining
- Use the Optimizing Compiler

Change the Garbage Collector

The garbage collection mode that you are using might be causing problems that you could avoid by changing to another garbage collection mode. For example, if you are using `-Xgc:pausetime` try switching to `-Xgc:throughput`. Note though, that if you change the garbage collection mode, you will not receive the same performance profile from the JRockit JVM.

If you are using the deterministic garbage collection mode, you cannot change to another garbage collection and yet retain the guarantees provided by deterministic garbage collection. In such a case, instead of changing the garbage collection mode, contact Oracle Support.

For more information, see the *Oracle JRockit Performance Tuning Guide*.

Disable Compaction

Bugs in heap compaction can sometimes cause problems leading to crashes during garbage collection. You can disable compaction by using the `-XXcompaction:enable=false` command-line option.

Note that using the `-XXcompaction:enable=false` option can lead to heap fragmentation; use it only for troubleshooting purposes. If the heap becomes too fragmented, you might encounter out-of-memory errors.

Disable Inlining

Erroneous inlining might cause broken code, which makes the garbage collector lose track of live objects. You can disable inlining by using the optimization directives as described in Specifying Optimization Directives.

Use the Optimizing Compiler

You might be experiencing garbage collection crashes because the nonoptimizing JIT compiler is generating broken code that makes the garbage collector lose track of live objects. Use the `-XX:+PreOpt` command at startup to use the optimizing compiler for everything. Note that using the optimizing compiler can slow down the JVM startup.

6.5.2.4 Contact Oracle Support

If the solutions provided in this section do not help you solve the problem, contact Oracle Support as described in Chapter 9, "Contacting Oracle for Support." When you contact Oracle Support, you must include the following information:

- For crashes in garbage collection, include a complete `.core` or `.mdmp` file, otherwise the support staff won't be able to resolve your issue. Verify that the `.core` or `.mdmp` file is at least as big as the Java heap.
- If you can reproduce the crash, include the steps you used to reproduce it.
- If you tried using another garbage collector indicate whether one garbage collector worked better than another or if crashes continued regardless of the collector used.
- Include information about any workaround you attempted.

7

Freezing JVM

When the JRockit JVM or Java application becomes unresponsive but has not crashed, it is considered to be frozen: the application stops responding to requests but the process still exists.

This chapter describes how to diagnose a freezing JVM and how to collect information that is essential for Oracle Support personnel to resolve the problem.

This chapter contains the following topics:

7.1 Diagnosing Where the Freeze is Occurring

A system can freeze in either the JVM or the application. To determine whether the freeze is occurring in the application or in the JVM, try to generate a thread dump.

- On Windows, press **Ctrl-Break**.
- On Linux and Solaris, send `SIGQUIT` (`kill -3`) to the parent Java Process ID.

Note: Alternatively, on all platforms, you can obtain the thread dump by using `jrcmd`; for example:

```
jrcmd nnnn "print_threads nativestack=true"
```

nnnn is the ID of the Java process. To view a list of the IDs of all Java processes running on the machine, run `jrcmd` without any command-line parameters.

If the system responds with a Java thread dump, then the application is freezing. For information about troubleshooting this type of freeze, see Section 7.2, "Troubleshooting a Java Application Freeze."

If you cannot obtain a thread dump, the JVM is freezing. For information about troubleshooting this type of freeze, see Section 7.3, "Troubleshooting a JVM Freeze."

7.2 Troubleshooting a Java Application Freeze

Review the thread dumps (look near the end of the thread dump) for locks and deadlocks.

If you cannot determine the cause of the freeze or if you cannot fix the problem easily by yourself, contact the developer of the application.

If neither you nor the application developer are able to diagnose the problem, contact Oracle Support as described in Chapter 9, "Contacting Oracle for Support." When you contact Oracle Support about a Java application freeze, provide the following information:

- Three thread dumps from the application when it works fine.
- Three thread dumps from the application when it has frozen.
- One 120s Flight Recorder recording from the application when it works fine.
- One 120s Flight Recorder recording from the application when it has frozen.

7.3 Troubleshooting a JVM Freeze

If you cannot get thread dumps with **Ctrl-Break** or by sending `SIGQUIT (kill -3)` to the parent Java process ID after a few attempts, the JVM has stopped handling signals and is freezing.

When this happens, you should force the JVM to crash and then contact Oracle Support for help. The crash file is essential for Oracle Support personnel to diagnose and troubleshoot the problem.

This section describes how you can force a frozen JRockit JVM to crash. It contains the following topics:

- Force the JRockit JVM to Crash (on a Linux System)
- Force the JRockit JVM to Crash (on a Windows System)
- Collecting State Information When the JRockit JVM is Running as a Service

7.3.1 Force the JRockit JVM to Crash (on a Linux System)

You can force a frozen JRockit JVM on a Linux system to crash by sending a SIGABRT signal, which aborts the process and causes a crash, thus producing a crash file.

To invoke `SIGABRT`, perform the following steps:

1. Find the process ID of the JRockit JVM process, by running the following command:

   Note: The following instructions are valid on 2.6 kernel-based Linux systems and on Red Hat Enterprise Linux 3.0 (or later versions).

   ```
   pstree -p user | grep java
   ```

   In this command, `user` is the Linux username (for example, `webadmin`) used to run the JRockit JVM process.

   - If the command results in excessive output, try `unset`ting the `LANG` environment variable first, by running `unset LANG`.
   - If the command shows only one Java process, proceed to step 2.
   - If the command shows several processes, print the command-line parameters for the required process, by running the following command:

     ```
     cat /proc/nnnn/cmdline | xargs --null -n1 echo
     ```

In this command, *nnnn* is the ID of the JRockit JVM process.

This command displays the command-line options for the specified JVM process. Check whether any of the command-line options could be causing the problem.

2. Find out the directory in which the binary crash file (`.core`) for the process will be created, by entering the following command:

```
ls -l /proc/nnnn/cwd
```

In this command, *nnnn* is the ID of the JRockit JVM process.

3. Create the binary crash file (and terminate the process) by running the following commands:

```
ulimit -c unlimited
kill -SIGABRT nnnn
```

In this command, *nnnn* is the ID of the JRockit JVM process.

7.3.2 Force the JRockit JVM to Crash (on a Windows System)

You can force a frozen JRockit JVM on a Windows system to crash by using the `windbg` command, as follows:

```
windbg.exe -Q -pd -p nnnn -c ".dump /u /ma hung.mdmp; q"
```

In this command, *nnnn* is the ID of the JRockit JVM process.

Note: `windbg` is included in the Debugging Tools for Windows package that you can download from:

http://www.microsoft.com/whdc/devtools/debugging/default.mspx

---

7.3.3 Collecting State Information When the JRockit JVM is Running as a Service

If the JRockit JVM starts as a service, you can collect thread dumps by doing one of the following:

- If you are collecting information from your own machine, run `jrcmd` with the `print_threads` diagnostic command.

  For more information about the `jrcmd` command, see the *Oracle JRockit JDK Tools*.

- If you are collecting information from another machine, use Oracle JRockit Mission Control with the `diagnostics` bean. For more information, see the *Oracle JRockit Mission Control Online Help*.

- If you run the JRockit JVM with Oracle WebLogic Server, you can use `beasvc -dump` to obtain thread dumps from the JVM.

  For more information, see the Oracle WebLogic Server documentation.

- If you want to see the messages that a server instance prints to `stdout` and `stderr` (including stack traces and thread dumps), redirect `stdout` and `stderr` to a file, see the Oracle WebLogic Server documentation.

  To make the WebLogic Server instance print a thread dump to `stdout`, do one of the following:

- Use the `weblogic.Admin THREAD_DUMP` command.
- Enter the following command at the command prompt:

  ```
  WL_HOME\bin\beasvc -dump -svcname:service-name
  ```

  In this command, *WL_HOME* is the directory in which you installed WebLogic Server and *service-name* is the Windows service that is running a server instance; for example.:

  ```
  D:\Oracle\Middleware\wlserver_10.3\server\bin\beasvc -dump
  -svcname:mydomain_myserver
  ```

8

About Crash Files

When the JRockit JVM crashes, it creates snapshots of the state of the computer and the JVM process at the time of the crash. These snapshots are in the form of two crash files: a text crash file (`.dump`); and a binary crash file (`.mdmp`, a Windows platform minidump or `.core`, a Linux and Solaris platform core file). The names of the crash files are `jrockit.`*`pid`*`.dump` and `jrockit.`*`pid`*`.mdmp/core`, where `pid` is the process ID (for example, `jrockit.72.dump`).

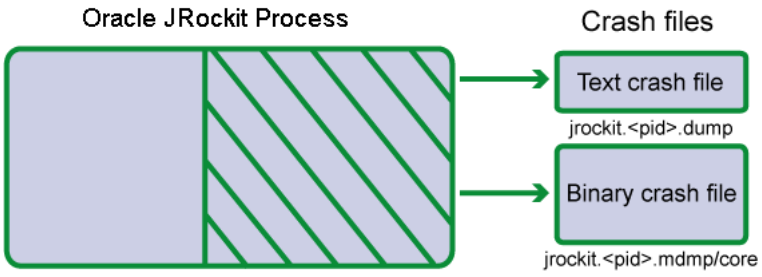
You can use information in the crash files to determine the problem that caused the JVM to crash. The information in the crash files is also essential for Oracle Support personnel to help troubleshoot problems with the JVM.

This chapter provides information about the differences between the crash files and how to enable and disable them. It also provides an example of a text crash file and describes how you can use the text crash file for troubleshooting.

This chapter contains the following topics:

8.1 Differences Between Text and Binary Crash Files

Figure 8–1 JRockit JVM Crash Files



You can open the text crash file (`.dump`) in any text editor. It contains information that can provide hints about the reasons for the crash (see Section 8.6.2, "Example of a Text Dump File."

You can open the binary crash file (`.core` or `.mdmp`) in a debugger. The crash file contains information about the entire JRockit JVM process.

8.2 Enabling Crash Files

Creation of crash files is enabled by default.

On Linux and Solaris systems, you must set `ulimit -c` to a value greater than zero (**recommendation**: `ulimit -c unlimited`). This value is measured in blocks, with each block equaling one kilobyte. You can specify the value from either the command line or a shell script.

8.3 Specifying the Location of the Crash Files

The text and binary crash files are saved to the current working directory.

If you want the crash files to be saved in a different directory, specify the directory by using the `JROCKIT_DUMP_PATH` environment variable.

- **On Linux and Solaris**:

  ```
  export JROCKIT_DUMP_PATH=path_to_directory
  ```

- **On Windows**:

  ```
  set JROCKIT_DUMP_PATH=path_to_directory
  ```

The directory that you specify must exist and must be writable.

8.4 Specifying the Size of the Binary Crash File

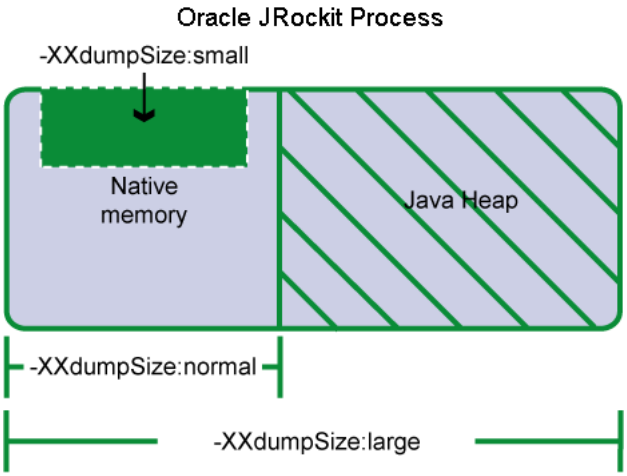
The text crash file is a small file, usually under 100KB. The binary crash file, however, is usually very large because, by default, the JRockit JVM logs the entire JVM process in the binary crash file. So you must ensure that there is adequate disk space for the binary crash file to be written to the disk.

You can set the size of the crash file by using the `-XXdumpSize` command-line option. The default setting is `-XXdumpSize:large`, which results in a binary crash file that contains information about the entire JVM process, including the Java heap. While `small` and `normal` settings are also available, neither of these sizes are adequate for troubleshooting a JVM crash, because, with these settings, the Java heap is excluded from the crash file.

If the disk on which the crash file is written does not have enough space for the binary crash file, the file becomes as large as possible; that is, it fills up the disk.

Figure 8–2 illustrates the differences in information between small, normal, and large binary crash files.

Figure 8–2 Difference in information saturation between small, normal, and large binary crash files



Note: When you contact Oracle Support, you must provide a large binary crash file.

8.5 Disabling Crash Files

Creation of crash files is enabled by default, so that, if a crash occurs, you can be sure to get as much information as possible about the application and the JRockit JVM process. At times, however, you might want to disable the creation of crash files (for example, when you have limited disk space).

Before disabling the creation of the crash files, note that the text crash file is a small, useful source of information to get an initial understanding of what could be wrong with the JVM. The binary crash file is essential when you contact Oracle Support.

- You can disable creation of the text crash file by using the `-XX:-DumpOnCrash` command-line option.
- You can disable creation of the binary crash file by using the `-XX:-CoreOnCrash` command-line option.

8.6 Troubleshooting by Using the Text Crash File

A JVM crash indicates that something happened that the JVM could not handle gracefully. The crash could be caused by a programming error in the JVM, a problem in the Java code, a problem in the JVM setup, or third-party libraries loaded in the JVM process.

The text crash file `.dump` is a good starting point to diagnose the problem.

8.6.1 Symptoms to Look for

A text crash file does not necessarily tell you exactly why the crash occurred, but it describes the environment in which the JRockit JVM was running and the state of the JVM when the crash occurred.

The following are a few easily identifiable symptoms that you can look for in the text crash file:

- If the `StackOverFlow` field indicates that stack overflow errors have occurred, the crash is likely to have been caused by a stack overflow.

  Look for reports of `Stack Overflow` in the `Error message` field or in the stack trace near the bottom of the crash file. Such occurrences might also indicate a stack overflow.

  For information about troubleshooting a crash caused by a stack overflow, see Section 6.3, "Stack Overflow Crash."

- If the `C Heap` field indicates that memory allocations have failed, the process might have run out of virtual memory. For information about troubleshooting crashes caused by virtual memory errors, see Section 6.2, "Out-Of-Virtual-Memory Crash."

If the text crash file shows a symptom different from the symptoms described in this section, try troubleshooting by using the information in Section 6.5, "JVM Crash."

8.6.2 Example of a Text Dump File

This section provides an example of a text crash file (`.dump`) and describes its parts.

Note that the text crash file does not provide a detailed description of what happened during the crash. The layout of the file that the JVM produces might differ from the layout shown in the example.

Table 8–1 Example of a Text Crash File (.dump)

Section	Contents

Beginning of the Text Crash File	`===== BEGIN DUMP`
	`==========`
	`JRockit dump produced after 0 days, 00:00:01 on Mon Dec 7 16:28:40 2009`
	
	`Additional information is available in:`
	`/localhome/mycomp/gdbtest/jrockit.17470.dump`
	`/localhome/mycomp/gdbtest/core or core.17470`

Error Message from the Operating System	`Error Message: Illegal memory access. [54]` `Signal info : si_signo=11, si_code=1 si_addr=0x7`
JRockit JVM Version	`Version : Oracle JRockit(R) R28.0.0-606-124955-1.6.0_17-20091130-2120-linux-ia32`

CPU and Memory Information	`CPU : Intel Pentium 4 (HT) SSE SSE2 NetBurst`
	`Number CPUs : 8`
	`Tot Phys Mem : 8248045568 (7865 MB)`

Operating System Version Information	`OS version : Red Hat Enterprise Linux AS release 4 (Nahant Update 8)` `Linux version 2.6.9-89.0.0.0.1.ELsmp (mockbuild@ca-build10.us.oracle.com) (gcc version 3.4.6 20060404 (Red Hat 3.4.6-11.0.1)) #1 SMP Tue May 19 04:23:49 EDT 2009 (i686)`

Thread and State Information	`Thread System: Linux NPTL`
	`LibC release : 2.3.4-stable`
	`Java locking : Lazy unlocking enabled (class banning) (transfer banning)`
	`State : JVM is running`

Table 8–1 (Cont.) Example of a Text Crash File (.dump)

Section	Contents
Command-Line Option Information	`Command Line : -Djava.library.path=. -Dsun.java.launcher=SUN_STANDARD Crash -static write`

Environment Information	`java.home : /localhome/jrockits/R28.0.0_R28.0.0-606_1.6.0/jre`
	`j.class.path : .`
	`j.lib.path : .`
	`JAVA_HOME : /localhome/jrockits/R27.6.3_R27.6.3-40_1.6.0`
	`LD_LIBRARY_PATH: /localhome/jrockits/R28.0.0_R28.0.0-606_1.6.0/jre/lib/i386/jrockit:/localhome/jrockits/R28.0.0_R28.0.0-606_1.6.0/jre/lib/i386:/localhome/jrockits/R28.0.0_R28.0.0-606_1.6.0/jre/../lib/i386`

Garbage Collection Information	`GC Strategy : Mode: throughput, with strategy: genparpar (basic strategy: genparpar)`
	`GC Status : OC is not running. Last finished OC was OC#0.`
	`: YC is not running. Last finished YC was YC#0.`
	`YC History : Ran 0 YCs since last OC.`
	`Heap : 0x76eb7000 - 0x7aeb7000 (Size: 64 MB)`
	`Compaction : (no compaction area)`
	`NurseryList : 0x76eb7000 - 0x78eb7000`
	`KeepArea : 0x786b6fe8 - 0x78eb7000`
	`KA Markers : [ 0x77eb6ff0, 0x786b6fe8 , 0x78eb7000 ]`

Registers and Stack Information	`Registers (from ThreadContext: 0xb7ba4e60:`
	`eax = 00000007 ecx = b7ba519c edx = 00000000 ebx = b7ba6f70`
	`esp = b7ba514c ebp = b7ba5150 esi = b7ba5178 edi = b7ba70f4`
	`es = 0000007b cs = 00000073 ss = 0000007b ds = 0000007b`
	`fs = 00000000 gs = 00000033`
	`eip = 74405765 eflags = 00000292`
	
	`Stack:`
	`(* marks the word pointed to by the stack pointer)`
	`b7ba514c: 00000007* b7ba5170 7440584c 00000007 b7ba6f70 00000001`
	`b7ba5164: 74bb4405 00000007 00000000 00000001 746a9f80 b7ba70f4`
	`b7ba517c: b7ba519c 00000007 00000000 74bb4400 744842a0 746a9f7b`
	`b7ba5194: b7ba7228 b7ba5178 770643d0 b7ba6f70 00000001 770650d8`

Thread Stack Trace Information	`Thread:`
	`"Main Thread" id=1 idx=0x4 tid=17471 lastJavaFrame=0xb7ba518c`
	`Stack 0: start=0xb71a5000, end=0xb7ba6000, guards=0xb71aa000 (ok), forbidden=0xb71a8000`
	`Thread Stack Trace:`
	`at write+15()@0x74405765`
	`at Java_Crash_staticWrite+28()@0x7440584c`
	`-- Java stack --`
	`at Crash.staticWrite(J)V(Native Method)[optimized]`
	`at Crash.run(Crash.java:62)`
	`at Crash.main(Crash.java:121)`
	`at jrockit/vm/RNI.c2java(IIIII)V(Native Method)[optimized]`
	`-- end of trace`

Table 8–1 (Cont.) Example of a Text Crash File (.dump)

Section	Contents
Memory Usage Information	Memory usage report: Total mapped 1133412KB (reserved=0KB) - Java heap 1048576KB (reserved=983040KB) - GC tables 35084KB - Thread stacks 13332KB (#threads=18) - Compiled code 384KB (used=256KB) - Internal 520KB - OS 7996KB - Other 0KB - JRockit malloc 24448KB (malloced=24397KB #290077) Does not track allocation sites! - Native memory tracking 1024KB (malloced=46KB #11) Does not track allocation sites! - Java class data 2048KB (malloced=1820KB #2455) Does not track allocation sites! Set the env variable TRACE_ALLOC_SITES=1 or use the print_memusage switch trace_alloc_sites=1 to enable alloc site tracing.

Beginning of the Text Crash File

This part of the `.dump` file contains information about when the crash occurred and for how long the JVM has been running.

It also provides a link to information to help you troubleshoot the crash.

The file locations refer to the locations of the text and binary crash files.

Error Message from the Operating System

This part of the `.dump` file contains the error message that the operation threw at the time of the crash. For more information about the error, see the documentation for your operating system.

JRockit JVM Version

This part of the `.dump` file indicates the release number of the JRockit JVM.

CPU and Memory Information

This part of the `.dump` file indicates the CPU in the system, the number of CPUs used, and the amount of memory consumed by the Java process, application, or the JRockit JVM.

Operating System Version Information

This part of the `.dump` file indicates the version of the operating system on which the JRockit JVM is running. Ensure that the JRockit JVM is running on a supported operating system.

For more information, see the see *Oracle JRockit JDK Supported Configurations* at `http://www.oracle.com/technology/software/products/ias/files/fusion_certification.html`.

Thread and State Information

This part of the `.dump` file indicates the thread system that the JRockit JVM used at the time of the crash and the state of the JVM. In this example, the JVM used the Native POSIX Thread Library (NPTL).

Command-Line Option Information

This part of the `.dump` file shows all the command-line options that were used at JVM startup.

Environment Information

This part of the `.dump` file provides information about the JVM environment:

- `JAVA_HOME` is the path to the Java home catalog; that is, the directory in which the JRockit JVM is installed.
- `_JAVA_OPTIONS` shows command-line options that are automatically passed to all newly started JRockit JVMs.
- `LD_LIBRARY_PATH` is a Linux- and Solaris-specific environment variable that can make the JRockit JVM find libraries other than the default system libraries. Sometimes, you might have to set this variable for running JNI code.

Garbage Collection Information

This part of the `.dump` file indicates the garbage collection mode.

Depending on the garbage collection mode used, this part of the `.dump` file might show other information such as the location of the nursery and keep area in the memory.

Registers and Stack Information

This part of the `.dump` file provides the following information:

- The `Registers` section is useful to Oracle Support personnel for troubleshooting. If the value of the **`esp`** register does not match the first number of the stack, the text crash file could be incorrect.
- If the `Stack` section shows `unreadable`, the crash is probably due to a stack overflow. The stack information is usually much longer than that shown in the example.

Thread Stack Trace Information

This part of the `.dump` file shows what the crashed thread was doing when the JRockit JVM crashed.

Memory Usage Information

This part of the `.dump` file shows details of memory usage (including native memory).

8.7 Generating Java Heap Dumps in the HPROF Binary Format

HPROF is a heap profiling tool that produces heap dumps in a specific format that heap analysis tools can parse.

You can set up the Oracle JRockit JVM to produce a dump of the Java heap in the HPROF binary format by using the following command-line options:

- `-XX:+HeapDumpOnOutOfMemoryError`: Enables generation of Java heap dumps when out-of-memory errors occur.

- `-XX:+HeapDumpOnCtrlBreak`: Enables generation of Java heap dumps when you press `Ctrl-Break`.

For more information about these and other related command-line options, see the *Oracle JRockit Command-Line Reference*.

You can also generate an HPROF-formatted dump of the Java heap by using the `hprof` diagnostic command. For more information, see *Oracle JRockit JDK Tools*.

For information about the HPROF tool, see `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html`.

9

Contacting Oracle for Support

If you have a service agreement with Oracle, you can contact Oracle Support for help with JRockit JVM problems.

Before contacting Oracle Support, perform the following actions:

- Try all the appropriate diagnostics and troubleshooting guidelines described in this document (*Oracle JRockit Diagnostics and Troubleshooting Guide*).
- Check whether the problem (or a similar problem) has been discussed in the JRockit forum at `http://forums.oracle.com/`.

  If the information available on the forum is not sufficient to help you solve the problem, post a question on the forum. Other JRockit users on the forum might respond to your question.
- Collect as much relevant data as possible about the problem. For example,
  - If a deadlock occurs, generate a thread dump.
  - If a crash occurs, locate the binary crash file (where applicable) and the appropriate error file.
- Document the environment and the actions performed just before you encountered the problem.

  If you run the JRockit JVM in a virtualized environment, information about the type of virtualization and the software used would be useful to Oracle Support.
- Where applicable, try to restore the original state of the system and reproduce the problem using the documented steps. This helps to determine whether the problem is reproducible or an intermittent issue.
- If the issue can be reproduced, try to narrow down the steps for reproducing the problem. Problems that can be reproduced by small test cases are typically easier to diagnose when compared with large test cases.

  Narrowing down the steps for reproducing problems enables Oracle Support to provide solutions for potential problems faster.

Note: When you send files (`.dump`, `.core`, and so on) to Oracle Support, remember to provide the MD5 checksum value for each file, so that Oracle Support personnel can verify the integrity of the files before using them for troubleshooting the problem.
