Oracle® JRockit

JDK Tools Guide

Release R28

E15061-07

July 2016

ORACLE®

Oracle JRockit JDK Tools Guide, Release R28

E15061-07

Primary Author: Savija T.V.

Contents

Preface

This document describes how to use the Oracle JRockit JDK diagnostic tools.

About the Document

This document contains the following chapters:

- Chapter 1, "Overview of Oracle JRockit JDK Tools" provides an overview of the tools included in the Oracle JRockit JDK.
- Chapter 2, "Understanding Verbose Output" describes how to interpret verbose output generated by the Oracle JRockit JVM.
- Chapter 3, "Using Thread Dumps" describes how to get and use Oracle JRockit JVM thread dumps.
- Chapter 4, "Running Diagnostic Commands" how to run diagnostic commands and lists the available commands.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs 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.

Convention	Meaning
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Overview of Oracle JRockit JDK Tools

This chapter provides an overview of the following Oracle JRockit JDK tools and features that you can use to identify and resolve problems when running an application on the JVM.

- Customizable Verbose Logs
- Thread Dumps
- Diagnostic Commands
- Time Zone Updater
- Oracle JRockit Mission Control Client

1.1 Customizable Verbose Logs

Verbose logs can be enabled and configured by using the `-Xverbose` command-line option. These logs provide low-overhead, run-time information on various aspects of the JRockit JVM (for example, memory management and code optimizations). You can use the information in the verbose logs for monitoring, tuning, and diagnostics.

Chapter 2, "Understanding Verbose Output", describes some of the useful verbose modules and how to interpret their output.

1.2 Thread Dumps

Thread dumps, described in Chapter 3, "Using Thread Dumps", are snapshots of the state of all threads that are part of the process. These dumps reveal information about an application's thread activity, which can help you diagnose problems and optimize the performance of the application and the JVM.

1.3 Diagnostic Commands

Diagnostic commands enable you to communicate with a running Oracle JRockit JVM process. As described in Chapter 4, "Running Diagnostic Commands", these commands tell the JRockit JVM to perform tasks such as printing a heap report or a garbage collection activity report, or to enable a specific verbose module.

1.4 Time Zone Updater

The TZUpdater tool updates the installed JDK/JRE images with more recent time zone data to accommodate the U.S. 2007 daylight saving time changes (US2007DST) established in the U.S. Energy Policy Act of 2005.

Oracle recommends using the latest Oracle JRockit JDK release as the preferred method for delivering both time zone data updates and other product improvements, such as security fixes. If you are unable to use the latest JRockit JDK release, this tool provides a way to update time zone data while leaving other system configurations unchanged.

TZUpdater (version 1.3.33 and above) supports Oracle JRockit JDK. For more information about TZUpdater, see the TZUpdater documentation on the Oracle Technology Network at:
http://www.oracle.com/technetwork/java/javase/tzupdater-readme-136440.html.

You can download the TZUpdater tool from:
http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html#timezone.

1.5 Oracle JRockit Mission Control Client

In addition to the tools supplied in the JDK, Oracle JRockit includes a suite of monitoring, management, and analysis tools called the JRockit Mission Control Client.

This suite of tools helps you monitor, manage, profile, and eliminate memory leaks in your Java application without undue performance overhead. The performance overhead of the JRockit Mission Control Client is low because it uses data that is collected as part of the JRockit JVM's adaptive dynamic optimization. This also eliminates the problem with the Heisenberg anomaly that can occur when tools using byte code instrumentation alter the execution characteristics of the system.

The JRockit Mission Control Client features can be available on-demand, and the small performance overhead occurs only while the tools are running.

The JRockit Mission Control Client consists of the following tools, which are also available as plug-in applications for Eclipse:

- JRockit Browser
- JRockit Management Console
- JRockit Flight Recorder
- JRockit Memory Leak Detector

1.5.1 JRockit Browser

The JRockit Browser enables you to set up and manage all running instances of the JRockit JVM on your system. From the JRockit Browser, you can activate different tools, such as starting a flight recording from the JRockit Flight Recorder, connecting to a Management Console, and starting memory leak detection.

1.5.2 JRockit Management Console

The JRockit Management Console is used to monitor a JRockit JVM instance. Several Management Consoles can be running concurrently. The console captures and presents active data about memory usage, CPU usage, and other run-time metrics.

- On a JRockit Management Console connected to JRockit JVM 5.0 and 6, information from any JMX MBean deployed in the Oracle JRockit JVM internal MBean server also can be displayed.
- On a console connected to Oracle JRockit JVM 1.4, RMP capabilities are exposed by a JMX proxy.

1.5.3 JRockit Flight Recorder

The JRockit Flight Recorder is a performance monitoring and profiling tool that makes diagnostic information available, even following catastrophic failure, such as a system failure. The Flight Recorder is a rotating buffer of diagnostic and profiling data that is available on-demand. Like its aeronautic namesake, the Flight Recorder provides an account of events leading up to a system (or application) failure. Whenever a Java application running on Oracle JRockit is started with the Flight Recorder enabled, data is constantly collected in the Flight Recorder or its buffers. In the event of a system failure, the content of this recording can be transferred to a file and used to analyze what went wrong: a JVM crash, an unexpected application termination, or a power failure.

Data displayed by the Flight Recorder GUI is collected by the Flight Recorder Run Time component, which is part of the Oracle JRockit JVM. For more information about the Flight Recorder Run Time component, see *Oracle JRockit Flight Recorder Run Time Guide*.

Flight recordings are displayed in the Flight Recorder GUI, which is part of the JRockit Mission Control Client. It enables users who are running a Flight Recorder-compliant version of Oracle JRockit (that is, release R28.0 or later) to view the JVM's recordings, the current recording settings, and the run time parameters. The GUI consists of a series of tabs that aggregate performance data into logical groupings.

The Flight Recorder replaces the JRockit Runtime Analyzer (JRA) that was provided with earlier releases of the JRockit Mission Control Client.

1.5.4 JRockit Memory Leak Detector

The JRockit Memory Leak Detector helps you discover memory leaks in a Java application and determine their cause. The JRockit Memory Leak Detector's trend analyzer discovers slow leaks, shows detailed heap statistics (including referring types and instances to leaking objects) and allocation sites, and provides a quick drill down to the cause of the memory leak. The Memory Leak Detector uses advanced graphical presentation techniques to make it easier to navigate and understand complex information.

1.5.5 Mission Control Documentation

Comprehensive online help is included in the product for all of the tools described in this document. In addition, the *Introduction to Oracle JRockit Mission Control Client* document, which is available on the Oracle Technology Network, provides introductory information about configuration and set up, communication protocols, and use cases.

2

Understanding Verbose Output

The `-Xverbose` command-line option enables verbose output from the Oracle JRockit JVM. You can use the verbose output for monitoring, tuning, and diagnostics. This chapter describes some of the most useful verbose modules and how to interpret the output from them.

> **Note:** Output can differ between JVM releases. The exact format of the output is subject to change at any time.

This chapter contains the following sections:

- Section 2.1, "Memory Management Verbose Log Modules"
- Section 2.2, "Other Verbose Log Modules"

2.1 Memory Management Verbose Log Modules

Table 2–1 lists the verbose log modules available in the JRockit JVM for memory management and garbage collection.

Table 2–1 Memory Management Verbose Modules

Log Module	Description	Uses
`memory`	Basic garbage collection information	Tuning and monitoring
`nursery`	Nursery information	Tuning and monitoring
`memdbg`	Memory management information	Tuning, monitoring, and diagnostics
`compaction`	Compaction information	Tuning, monitoring, and diagnostics
`gcpause`	Garbage collection pause times	Tuning, monitoring, and diagnostics
`gcreport`	Garbage collection summary	Tuning and monitoring
`refobj`	Reference object information	Monitoring and diagnostics
`alloc`	Allocation and out of memory information	Monitoring and diagnostics

Most verbose modules are available at several log levels. With the exception of the `memdbg` module, this section covers only the default (`info`) log level. This is the most useful level for general tuning and monitoring purposes.

2.1.1 Verbose memory Log Module

The `-Xverbose:memory` (or `-Xverbose:gc`) module provides basic information about garbage collection events. The overhead for this module is very low, which means that you can enable it in a production environment.

2.1.1.1 Initial Output of the memory Module

At JVM startup, the `memory` module displays some basic numbers on the memory management system configuration and a guide to how to read the garbage collection output.

Example 2–1 shows an example of the initial output from the `memory` log module. Line numbers have been added.

Example 2–1 Initial Output of the memory Module

```
1: [INFO ][memory ] Running with 32 bit heap and compressed references.
2: [INFO ][memory ] GC mode: Garbage collection optimized for throughput, strategy: Generational
Parallel Mark & Sweep.
3: [INFO ][memory ] Heap size: 65536KB, maximum heap size: 3145728KB, nursery size: 32768KB.
4: [INFO ][memory ] <start>-<end>: <type> <before>KB-><after>KB (<heap>KB), <time> ms, sum of
pauses <pause> ms.
5: [INFO ][memory ] <start>  - start time of collection (seconds since jvm start).
6: [INFO ][memory ] <type>   - OC (old collection) or YC (young collection).
7: [INFO ][memory ] <end>    - end time of collection (seconds since jvm start).
8: [INFO ][memory ] <before> - memory used by objects before collection (KB).
9: [INFO ][memory ] <after>  - memory used by objects after collection (KB).
10: [INFO ][memory ] <heap>   - size of heap after collection (KB).
11: [INFO ][memory ] <time>   - total time of collection (milliseconds).
12: [INFO ][memory ] <pause>  - sum of pauses during collection (milliseconds).
13: [INFO ][memory ]            Run with -Xverbose:gcpause to see individual phases.
```

- Line 1 indicates whether compressed references are used.
- Line 2 describes the garbage collection mode used in this run, and the initial garbage collection strategy.
- Line 3 shows the initial and maximum heap size, and the initial nursery size.
- Lines 4–13 describe the format of the garbage collection output.

2.1.1.2 Garbage Collection Output of the memory Module

The `memory` module displays a line for each garbage collection.

Example 2–2 shows an example of verbose output from the `memory` module. Line numbers have been added.

Example 2–2 Garbage Collection Output of the memory Module

```
1: [INFO ][memory ] [YC#1] 0.981-1.019: YC 35001KB->33131KB (65536KB), 0.038 s, sum of pauses
37.851 ms, longest pause 37.851 ms.
2: [INFO ][memory ] [YC#2] 1.688-1.706: YC 58229KB->45536KB (65536KB), 0.019 s, sum of pauses
18.683 ms, longest pause 18.683 ms.
3: [INFO ][memory ] [OC#1] 1.706-1.718: OC 65536KB->57671KB (86176KB), 0.012 s, sum of pauses
10.364 ms, longest pause 10.364 ms.
4: [INFO ][memory ] [YC#3] 1.997-2.006: YC 65143KB->55299KB (86176KB), 0.008 s, sum of pauses 8.328
ms, longest pause 8.328 ms.
```

Each garbage collection output starts with a timestamp, which is the time in seconds since the JVM started. The format of these lines is described at line 4 in Example 2–1.

- Lines 1, 2 and 4 are output from young collections. The young collection on line 1 reduced the size of the heap from 35001 KB to 33131 KB.
- Line 3 is an output from an old collection, which reduced the size of the heap from 65536 KB to 57671 KB.

2.1.1.3 Page Fault Warning

Page faults cause memory access to become slow, and page faults during garbage collection might increase garbage collection time. Because of this, the verbose `memory` log module displays a warning whenever the number of page faults during the garbage collection was more than five percent of the number of pages in the heap. This output is not available on Windows platforms.

Example 2–3 shows a page fault warning.

Example 2–3 Page Fault Warning

```
[INFO ][memory ] [OC#1] Warning: Your computer has generated 9435 page faults during the last
garbage collection.
[INFO ][memory ] [OC#1] If you find this swapping problematic, consider running JRockit with a
smaller heap.
```

2.1.2 Verbose nursery Log Module

The `-Xverbose:nursery` log module provides details about young collections and nursery sizing. Some of this information is useful for monitoring and tuning the JRockit JVM. The overhead is very low, which means that you can enable this module in a production environment.

2.1.2.1 Young Collection Output of the nursery Module

Example 2–4 shows the output from the `nursery` log module during young and old collections. Line numbers have been added.

Example 2–4 Young Collection Output of the nursery Module

```
1: [INFO ][nursery] [YC#2] Young collection 2 started. This YC is running while the OC is in phase:
not running.
2: [INFO ][nursery] [YC#2] Setting forbidden area for keeparea: 0x40ffdff0-0x417fdfe8.
3: [INFO ][nursery] [YC#2] Next keeparea will start at 0x417fdfe8 and end at 0x41ffe000.
4: [INFO ][nursery] [OC#1] Setting keepAreaMarkers[0] to 0x43883f00.
5: [INFO ][nursery] [OC#1] Setting keepAreaMarkers[1] to 0x43dc8860.
6: [INFO ][nursery] [OC#1] Setting keepAreaMarkers[2] to 0x43f9ba88.
7: [INFO ][nursery] [OC#1] Next keeparea will start at 0x43dc8860 and end at 0x43f9ba88.
```

- Lines 1–3 are young collection output (denoted by `[YC#2]`); lines 4–7 are old collection output (denoted by `[OC#1]`).
- Line 1 shows the sequence number of the young collection. It also indicates that the old collection is not running while this young collection is running. Other possible old collection phases are marking, precleaning, and sweeping, which are the concurrent phases of the concurrent old collection.
- Lines 2–7 contain information related to the size and position of the keep area. This information is useful for advanced diagnostics.

2.1.2.2 Verbose Nursery-Size-Adjustment Output of the nursery Module

Some garbage collection modes and strategies adjust the nursery size at run time for optimal performance. The `nursery` log module shows some information about the nursery sizing calculations and nursery size changes.

Example 2–5 shows the nursery sizing output from an old collection. Line numbers have been added.

Example 2–5 Nursery-Size-Adjustment Output of the nursery Module

```
1: [INFO ][nursery] [OC#1] Nursery size increased from 0KB to 7474KB. Nursery list consists of 5
parts.
2: [INFO ][nursery] [OC#1] Nursery size remains at 7474KB. Nursery list consists of 5 parts.
```

- Line 1 shows that the extent to which the nursery size is being increased and the number of parts that the nursery contains.
- Line 2 shows that the size of the nursery was not changed.

2.1.3 Verbose memdbg Log Module

The `-Xverbose:memdbg` log module is an alias that starts several memory-related log modules to provide a complete picture of the memory management. The output from these modules is documented in the respective module. The `memdbg` log module sets the memory module to the `debug` level. Only output from memory at the `debug` level is documented in this section. This information can also be obtained by setting `-Xverbose:memory=debug`.

The overhead of the verbose `memdbg` output is low, which means it can be used in production environments.

2.1.3.1 Initial Output of the memdbg Module

Example 2–6 shows the initial output from the `memdbg` verbose module. Line numbers have been added.

Example 2–6 Initial Output of the memdbg Module

```
1: [DEBUG][memory ] Minimum TLA size is 2048 bytes.
2: [DEBUG][memory ] Preferred TLA size is 65536 bytes.
3: [DEBUG][memory ] TLA waste limit is 2048 bytes.
4: [DEBUG][memory ] Nursery object limit max is 65536 bytes.
5: [DEBUG][memory ] Nursery object limit percentage is 1.0%.
6: [DEBUG][memory ] Java heap regions:
7:  [000000003fffe000 - 0000000043ffe000] rw-- (pagesize=0x1000)
8:  [0000000043ffe000 - 000000007fffe000] ----
9:  [0000000080000000 - 0000000100000000] ----
10:  64MB committed, 3008MB reserved.
11: [DEBUG][memory ] Initial and maximum number of gc threads: 8, of which 8 parallel threads, 4
concurrent threads, and 8 yc threads.
12: [DEBUG][memory ] Maximum nursery percentage of free heap is 95%.
13: [DEBUG][memory ] Prefetch distance in workpacket: 4. Packet size: 493 objects.
14: [DEBUG][memory ] Using prefetch linesize: 64 bytes chunks: 512 bytes pf_dist: 256 bytes.
```

- Lines 1–5 shows the minimum and preferred thread local area sizes, the TLA waste limit, and information about the size of the objects that will be allocated in the nursery. These values depend on the heap size and the garbage collector, and the JRockit JVM release.

- Lines 6–10 display a memory map over the Java heap and the amount of memory committed and reserved in the virtual address space.

  There can be several heap regions. The regions might, but are not required to, be located at consecutive addresses.
- Line 11 shows the number of garbage collection threads.
- Line 12 shows the maximum size of the nursery.
- Lines 13 and 14 show information about prefetching, which is useful for advanced diagnostics and tuning.

2.1.3.2 Parallel Old Collection Output of the memdbg Module

The `memdbg` module adds information to the garbage collection output. For some tuning and diagnostic work, this information is essential. The output differs among garbage collection types.

Example 2–7 shows the verbose `memdbg` output from a parallel old collection. Line numbers have been added.

Example 2–7 Parallel Old Collection Output of the memdbg Module

```
1: [DEBUG][memory ] [OC#1] GC reason: Allocation request failed.
2: [DEBUG][memory ] [OC#1] 1.706: OC started.
3: [DEBUG][memory ] [OC#1] Starting parallel marking phase.
4: [DEBUG][memory ] [OC#1] SemiRef phase Finalizers run in single threaded mode.
5: [DEBUG][memory ] [OC#1] SemiRef phase WeakJNIHandles run in single threaded mode.
6: [DEBUG][memory ] [OC#1] SemiRef phase ClassConstraints run in single threaded mode.
7: [DEBUG][memory ] [OC#1] SemiRef phase FinalMemleak run in single threaded mode.
8: [DEBUG][memory ] [OC#1] Removing 4 permanent work packets from pool, now 60 packets.
9: [DEBUG][memory ] [OC#1] Total mark time: 5.145 ms.
10: [DEBUG][memory ] [OC#1] Ending marking phase.
11: [DEBUG][memory ] [OC#1] Starting parallel sweeping phase.
12: [DEBUG][memory ] [OC#1] Total sweep time: 4.735 ms.
13: [DEBUG][memory ] [OC#1] Ending sweeping phase.
14: [DEBUG][memory ] [OC#1] Expanded the heap from 65536KB to 86176KB.
15: [DEBUG][memory ] [OC#1] Page faults before OC: 1, page faults after OC: 1, pages in heap:
21544.
16: [DEBUG][memory ] [OC#1] Nursery size after OC: 7474KB. (Free: 7472KB Parts: 5)
```

- Line 1 displays the reason for the garbage collection.
- Line 2 displays a timestamp of when the old collection was started.
- Lines 3–10 describe the actions taken during the mark phase. This mark phase took 5.145 ms, including pauses and concurrent phases. In this case, the Java threads were paused during the entire mark phase.
- Lines 11–13 describe the sweep phase.
- Line 14 shows the extent to which the Java heap size was increased.
- Line 15 indicates the number of page faults before and after the old collection. Page faults during the garbage collection may slow the garbage collection.
- Line 16 shows how large the nursery is after the old collection.

2.1.3.3 Concurrent Old Collection Output of the memdbg Module

The `memdbg` memory module adds useful information to the garbage collection output. For tuning and diagnostic evaluation, this information is essential. The output differs among garbage collection types.

Example 2–8 shows the verbose `memdbg` output from a concurrent old collection. Line numbers have been added.

Example 2–8 Concurrent Old Collection Output of the memdbg Module

```
1: [DEBUG][memory ] [OC#1] GC reason: Allocation request failed.
2: [DEBUG][memory ] [OC#1] 1.768: OC started.
3: [DEBUG][memory ] [OC#1] Starting initial marking phase (OC1).
4: [DEBUG][memory ] [OC#1] SemiRef phase Finalizers run in single threaded mode.
5: [DEBUG][memory ] [OC#1] SemiRef phase WeakJNIHandles run in single threaded mode.
6: [DEBUG][memory ] [OC#1] SemiRef phase ClassConstraints run in single threaded mode.
7: [DEBUG][memory ] [OC#1] Initial marking phase promoted 0 objects (0KB).
8: [DEBUG][memory ] [OC#1] Starting concurrent marking phase (OC2).
9: [DEBUG][memory ] [OC#1] Concurrent mark phase lasted 5.137 ms.
10: [DEBUG][memory ] [OC#1] Starting precleaning phase (OC3).
11: [DEBUG][memory ] [OC#1] Precleaning phase lasted 0.300 ms.
12: [DEBUG][memory ] [OC#1] Starting final marking phase (OC4).
13: [DEBUG][memory ] [OC#1] SemiRef phase Finalizers run in single threaded mode.
14: [DEBUG][memory ] [OC#1] SemiRef phase WeakJNIHandles run in single threaded mode.
15: [DEBUG][memory ] [OC#1] SemiRef phase ClassConstraints run in single threaded mode.
16: [DEBUG][memory ] [OC#1] SemiRef phase FinalMemleak run in single threaded mode.
17: [DEBUG][memory ] [OC#1] Final marking phase promoted 0 objects (0KB).
18: [DEBUG][memory ] [OC#1] Adding 27 temporary work packets to permanent pool, now 91 packets.
19: [DEBUG][memory ] [OC#1] Total mark time: 11.528 ms.
20: [DEBUG][memory ] [OC#1] Ending marking phase.
21: [DEBUG][memory ] [OC#1] Starting concurrent sweeping phase.
22: [DEBUG][memory ] [OC#1] Total sweep time: 1.732 ms.
23: [DEBUG][memory ] [OC#1] Ending sweeping phase.
24: [DEBUG][memory ] [OC#1] Expanded the heap from 65536KB to 93216KB.
25: [DEBUG][memory ] [OC#1] GC Trigger is now 13.2%, was 12.0%.
26: [DEBUG][memory ] [OC#1] Page faults before OC: 1, page faults after OC: 1, pages in heap:
23304.
27: [DEBUG][memory ] [OC#1] Nursery size after OC: 29260KB. (Free: 29256KB Parts: 9)
```

- Line 1 shows the reason for the garbage collection.
- Line 2 shows a timestamp for when the old collection was started.
- Lines 3–20 describe the actions taken during the mark phase. Line 19 shows the time taken for the mark phase, which includes pauses and concurrent phases.
- Lines 21–23 describe the sweep phase. Line 20 shows the time taken for the sweep phase.
- Line 24 shows the extent to which the Java heap size was increased.
- Line 25 shows information about the garbage collection trigger.
- Line 26 shows the number of page faults before and after the old collection. Page faults during the garbage collection may slow the garbage collection.
- Line 27 shows the nursery size after the old collection.

2.1.3.4 Young Collection Output of the memdbg Module

Example 2–9 shows the verbose `memdbg` output from a young collection. Line numbers have been added.

Example 2–9 Young Collection Output of the memdbg Module

```
1: [DEBUG][memory ] [YC#1] GC reason: Allocation request failed.
2: [DEBUG][memory ] [YC#1] 0.981: YC started.
3: [DEBUG][memory ] [YC#1] Returning duplicate cardTablePart entry 2 (0x4201e000-0x4202e000)
```

```
4: [DEBUG][memory ] [YC#1] SemiRef phase Finalizers run in single threaded mode.
5: [DEBUG][memory ] [YC#1] SemiRef phase WeakJNIHandles run in single threaded mode.
6: [DEBUG][memory ] [YC#1] SemiRef phase ClassConstraints run in single threaded mode.
7: [DEBUG][memory ] [YC#1] YC promoted 26760 objects (24201KB).
8: [DEBUG][memory ] [YC#1] Page faults before YC: 1, page faults after YC: 1, pages in heap: 16384.
9: [DEBUG][memory ] [YC#1] Nursery size after YC: 32768KB. (Free: 24574KB Parts: 1)
```

- Line 1 shows the reason for the garbage collection. In this example, the garbage collection was started because an allocation failed. This is the typical reason for starting a young collection.
- Line 2 shows a timestamp for when the young collection was started.
- Line 3 shows what happens if several threads try to process the same memory. Note that this is the `debug` level.
- Lines 4–6 show information about different phases in the young collection.
- Line 7 shows the number (and size) of objects that were promoted during the young collection.
- Line 8 shows statistics about page faults before and after the garbage collection. Page faults during the garbage collection may slow the garbage collection.
- Line 9 shows the nursery size after the young collection.

2.1.3.5 Parallel Sweep in Concurrent Old Collections

If the heap becomes full during the mark phase of a concurrent old collection, the garbage collector will, by default, override the concurrent sweep phase and use a parallel sweep instead. Example 2–10 shows the output that this behavior generates.

Example 2–10 Output When a Promotion Fails

```
[INFO ][memory ] [OC#1] Changing GC strategy from: genconcon to: genconpar, reason: Emergency
parallel sweep requested.
```

2.1.4 Verbose Compaction Log Module

The JRockit JVM performs partial compaction of the heap at each old collection. Compaction reduces fragmentation in the heap, which makes object allocation faster. The verbose `compaction` log module displays information about the compaction. The overhead of this log module is low, which means that it can be enabled in production environments.

Example 2–11 shows the verbose output from the `compaction` log module. Line numbers have been added.

Example 2–11 Old Collection Output of the compaction Module

```
1: [INFO ][compact] [OC#1] Compaction reason: Normal.
2: [INFO ][compact] [OC#1] Compacting 256 of 4096 parts at index 3840. Compaction type is external.
Exceptional: No.
3: [INFO ][compact] [OC#1] Compaction area start: 0x43bfe000, end: 0x43ffe000. Timeout: 100.000 ms.
4: [INFO ][compact] [OC#1] Compactset limit (per thread): 37487 (dynamic), not using matrixes.
5: [INFO ][compact] [OC#1] Adjusted compaction area to start at 0x43bfe170 and end at 0x43ffe000.
6: [INFO ][compact] [OC#1] Average compact time ratio (move phase/total time): 0.855877.
7: [INFO ][compact] [OC#1] Too few references, doubling compact ratio.
8: [INFO ][compact] [OC#1] Adjusting compactset limit to 248535.
9: [INFO ][compact] [OC#1] Pause per 1000 refs, current: 0.201178, average: 0.201178. Target pause:
50.000.
10: [INFO ][compact] [OC#1] Compaction time, total: 4.191 ms (target 100.000 ms).
```

```
11: [INFO ][compact] [OC#1] Compaction moved 3967 objects and left 0 objects. Total moved size
4125680B.
12: [INFO ][compact] [OC#1] Compaction added 4193936B of free memory in 1 parts.
13: [INFO ][compact] [OC#1] Compaction time, move phase: 3.587 ms (target 50.000 ms).
14: [INFO ][compact] [OC#1] Compaction time, update phase: 0.604 ms (target 50.000 ms).
15: [INFO ][compact] [OC#1] Found 3967 references. Internal: 0  External: 3967.
16: [INFO ][compact] [OC#1] Updated 3967 references. Internal: 0  External: 3967.
```

- Line 1 shows the reason for the compaction. In this case, a normal compaction is triggered by the old collection. Other reasons for compaction can be that allocation requests require a larger compaction due to lack of free memory or that the heap is shrinking.
- Line 2 shows a summary of the upcoming compaction. In this example, 256 out of a total of 4096 heap parts will be compacted, starting at index 3840. The compaction type is external, which means that objects will be moved away from the compaction area. The compaction is not exceptional, which means that it can be stopped if it takes too long.
- Line 3 shows the physical addresses of the start and the end of the compaction area and for how long this compaction is allowed to run. This information is useful for advanced diagnostics.
- Line 4 shows the current compactset limit. The compactset contains references to the compaction area, and the size of this set determines how much compaction is performed during one old collection.
- Line 5 shows how the compaction area was adjusted. The heuristics adjusts the compaction area so that no active objects span the borders.
- Lines 6–9 is information from the heuristics. It informs you about changes to the compactset based on how much time the compaction took.
- Lines 10–12 show the result of the compaction. How much time it took, how many objects were moved, and how much free memory it returned to the allocator.
- Lines 13–14 show detailed information about the two components of the pause time caused by compaction, and the current pause targets for these components. In this example, moving objects took 3.587 ms and updating references to moved objects took 0.604 ms, while the target was 50 ms for each of the components.
- Lines 15–16 show the number of references found in the compaction area and updated due to moved objects. In this example, all 3967 references found were updated (which is good). Internal references are found within the compaction area, and external references originate from objects outside the compaction area. This information is useful for monitoring and tuning.

2.1.4.1 Output of a Skipped Compaction

Whenever a compaction is stopped, the verbose `compaction` log module will display information about the reason for the skipped compaction. In Example 2–12, the compactset for a garbage collection thread reached its top limit, which means that there were too many pointers to objects within the selected compaction area. In exceptional cases, the compaction may be skipped because there is not enough native memory available when trying to expand the data structure.

Example 2–12 Output of a Skipped Compaction

```
[INFO ][compact] [OC#3] Compaction was skipped due to 'Too many references, compact set limit
exceeded'.
```

```
[INFO ][compact] [OC#5] Compact set for thread '(OC Main Thread)' failed to extend beyond 890854
elements - Out of memory.
```

2.1.5 Verbose gcpause Log Module

The `-Xverbose:gcpause` log module displays information about individual garbage collection pauses. For monitoring, tuning, and diagnosing latencies, this information is essential. The overhead of the log module is low, which means that it can be used in production environments.

2.1.5.1 Parallel Old Collection Output of the gcpause Module

A parallel old collection pauses all Java threads during the entire garbage collection. The output from `-Xverbose:gcpause` during a parallel old collection is thus fairly simple, as shown in Example 2–13. Line numbers have been added.

Example 2–13 Parallel Old Collection Output of the gcpause Module

```
1: [INFO ][gcpause] [OC#1] [---]    11.511 ms (1.706000-1.718000)  OC
2: [INFO ][gcpause] [OC#1] [con]     0.039 ms (1.706000-1.707000)  OC:PreGC
3: [INFO ][gcpause] [OC#1] [pau]    10.364 ms (1.707000-1.717000)  OC:Main
4: [INFO ][gcpause] [OC#1] [con]     1.064 ms (1.717000-1.718000)  OC:PostGC
```

- Line 1 is a summary that shows that this old collection took 11.511 ms.
- Lines 2–4 show the times for each of the parts in the collection. `con` means that the part is done concurrently with the Java execution, `pau` means a pause in the Java execution.

2.1.5.2 Concurrent Old Collection Output of the gcpause Module

A mostly concurrent (or concurrent) old collection consists of several concurrent garbage collection phases interspersed by short pauses.

Example 2–14 shows the output from a mostly concurrent old collection. Line numbers have been added.

Example 2–14 Concurrent Old Collection Output of the gcpause Module

```
1: [INFO ][gcpause] [OC#1] [---]    14.946 ms (1.768000-1.783000)  OC
2: [INFO ][gcpause] [OC#1] [con]     0.058 ms (1.768000-1.769000)  OC:PreGC
3: [INFO ][gcpause] [OC#1] [pau]     5.180 ms (1.769000-1.774000)  OC:Initial
4: [INFO ][gcpause] [OC#1] [con]     5.393 ms (1.774000-1.779000)  OC:ConcurrentMark
5: [INFO ][gcpause] [OC#1] [pau]     3.112 ms (1.779000-1.782000)  OC:Main
6: [INFO ][gcpause] [OC#1] [con]     1.128 ms (1.782000-1.783000)  OC:PostGC
```

- Line 1 shows the total time taken for the old collection.
- Line 2–6 show the times for each of the parts in the collection. `con` means that the part is done concurrently with the Java execution; `pau` means a pause in the Java execution.

2.1.6 Verbose gcreport Log Module

The `-Xverbose:gcreport` log module displays a summary of garbage collection activity at the end of the run.

Example 2–15 shows the verbose `gcreport` output. Line numbers have been added.

Example 2–15 Output of the gcreport Module

```
1: [INFO ][gcrepor]
2: [INFO ][gcrepor] Memory usage report:
3: [INFO ][gcrepor]
4: [INFO ][gcrepor] Young Collections:
5: [INFO ][gcrepor]     number of collections = 30.
6: [INFO ][gcrepor]     total promoted =        166897 (size 170523360).
7: [INFO ][gcrepor]     max promoted =          26760 (size 24781800).
8: [INFO ][gcrepor]     total YC time =         0.380 s (total paused 0.378 s).
9: [INFO ][gcrepor]     mean YC time =          12.670 ms (mean total paused 12.596 ms).
10: [INFO ][gcrepor]     maximum YC Pauses =     29.256 , 37.851, 55.795 ms.
11: [INFO ][gcrepor]
12: [INFO ][gcrepor] Old Collections:
13: [INFO ][gcrepor]     number of collections = 5.
14: [INFO ][gcrepor]     total promoted =        0 (size 0).
15: [INFO ][gcrepor]     max promoted =          0 (size 0).
16: [INFO ][gcrepor]     total OC time =         0.128 s (total paused 0.119 s).
17: [INFO ][gcrepor]     mean OC time =          25.579 ms (mean total paused 23.739 ms).
18: [INFO ][gcrepor]     maximum OC Pauses =     16.818 , 26.550, 51.602 ms.
19: [INFO ][gcrepor]
20: [INFO ][gcrepor]
21: [INFO ][gcrepor]     number of internal compactions = 3.
22: [INFO ][gcrepor]     number of external compactions = 2.
23: [INFO ][gcrepor]       1 of these were stopped because they timed out.
24: [INFO ][gcrepor]
```

- Lines 4–10 show information about the young collections during this run.
- Line 5 shows the total number of young collections.
- Line 6 shows the total number of objects promoted and their total size in bytes.
- Line 7 shows the largest number of objects promoted during a single young collection and their total size in bytes.
- Line 8 shows the total time spent in young collections.
- Line 9 shows the average time spent in a single young collection.
- Line 10 shows the three longest pause times caused by a young collection.
- Lines 12–18 show statistics about old collections.
- Line 13 shows the total number of old collections.
- Line 14 shows the number of objects promoted during old collections and their total size in bytes.
- Line 15 shows the largest number of objects promoted during a single old collection and their total size in bytes.
- Line 16 shows the total time spent in old collections and the sum of all garbage collection pauses caused by old collections.
- Line 17 shows the average time spent in a single old collection and the average number of pauses during a single old collection.
- Line 18 shows the three longest old collection pauses.
- Line 21 shows the number of internal compactions.
- Line 22 shows the number of external compactions.
- Line 23 shows that one compaction was stopped due to a timeout.

2.1.7 Verbose refobj Log Module

The `-Xverbose:refobj` log module shows a summary of the number of reference objects and how they are handled at each garbage collection.

Example 2–16 shows the verbose `refobj` output. Line numbers have been added.

Example 2–16 Output of the refobj Module

```
1: [INFO ][refobj ] [YC#1] SoftRef: Reach: 10 Act: 0 PrevAct: 0 Null: 0
2: [INFO ][refobj ] [YC#1] WeakRef: Reach: 14 Act: 0 PrevAct: 0 Null: 0
3: [INFO ][refobj ] [YC#1] Phantom: Reach:  0 Act: 0 PrevAct: 0 Null: 0
4: [INFO ][refobj ] [YC#1] ClearPh: Reach:  0 Act: 0 PrevAct: 0 Null: 0
5: [INFO ][refobj ] [YC#1] Finaliz: Reach:  4 Act: 2 PrevAct: 0 Null: 0
6: [INFO ][refobj ] [YC#1] WeakHnd: Reach: 49 Act: 0 PrevAct: 0 Null: 0
7: [INFO ][refobj ] [YC#1] SoftRef: @Mark: 10 @Preclean: 0 @FinalMark:  0
8: [INFO ][refobj ] [YC#1] WeakRef: @Mark: 11 @Preclean: 0 @FinalMark:  3
9: [INFO ][refobj ] [YC#1] Phantom: @Mark:  0 @Preclean: 0 @FinalMark:  0
10: [INFO ][refobj ] [YC#1] ClearPh: @Mark:  0 @Preclean: 0 @FinalMark:  0
11: [INFO ][refobj ] [YC#1] Finaliz: @Mark:  0 @Preclean: 0 @FinalMark:  6
12: [INFO ][refobj ] [YC#1] WeakHnd: @Mark:  0 @Preclean: 0 @FinalMark: 49
13: [INFO ][refobj ] [YC#1] SoftRef: SoftAliveOnly: 10 SoftAliveAndReach: 0
14: [INFO ][refobj ] [YC#1] NOTE: This count only applies to a part of the heap.
```

- Lines 1–6 show the number of occurrences of each reference object type, finalizers, weak handles and object monitors. The references objects are categorized by status,
  - **Reachable**: Reference objects with reachable referents. A referent that is reachable on a harder level is considered reachable; for example, a referent of a soft reference is reachable if it also is hard reachable.
  - **Activated**: Activated references are such that the referent is no longer reachable on any harder level than this reference, which means that the reference can be cleared or put on a reference queue.
  - **Previously Activated**: References that have been activated at a previous garbage collection but have not yet been cleared are previously activated.
  - **Null**: Null references are such that the reference in the reference object is null, but the reference object itself still exists.
- Lines 7–12 show statistics about which garbage collection phases the reference objects were handled.
- Line 13 shows how many soft references are soft alive only and how many are also hard reachable.

2.1.8 Verbose alloc Log Module

The `-Xverbose:alloc` log module gives information about memory allocations. The module was introduced in R28.0.

Example 2–17 shows the `alloc` output. Line numbers have ben added.

Example 2–17 Output of the alloc Module

```
1: [INFO ][alloc  ] [YC#1] Pending requests at 'Before YC' - Total: 1, TLAs: 1 (approx 65536
bytes), objects: 0 (0 bytes). Max age: 0.
2: [INFO ][alloc  ] [YC#1] Satisfied 0 object and 1 tla allocations. Pending requests went from 1
to 0.
```

```
3: [INFO ][alloc  ] [YC#1] Pending requests at 'After YC' - Total: 0, TLAs: 0 (approx 0 bytes),
objects: 0 (0 bytes). Max age: 0.
```

- Line 1 shows information about the status of the object allocation queue before the garbage collection.
- Line 2 shows how many allocation requests were satisfied as a result of this collection.
- Line 3 shows information about the status of the object allocation queue after the garbage collection. You can compare this information to the information in line 1 to get an idea of how well object allocation is doing. Many objects in the allocation queue at the end of a garbage collection may be an indication that object allocation is difficult, for example due to heavy fragmentation.

2.2 Other Verbose Log Modules

Among the various log modules available in the JRockit JVM, `opt` and `exceptions` are two of the commonly used and most useful.

2.2.1 Output of the opt Module

The `-Xverbose:opt` log module displays information about code optimizations done by the optimizing compiler.

Example 2–18 shows the verbose `opt` output. Line numbers have been added.

Example 2–18 Output of the opt Module

```
1: [INFO ][opt ][00036] #1 (Opt) ObjAlloc.main([Ljava/lang/String;)V

2: [INFO ][opt ][00036] #1 3.756-3.758 0x0000000100060000-0x000000010006004E 2.10
ms 128KB 7633 bc/s (2.10 ms 7633 bc/s)
```

- Line 1 shows the names of two methods that are optimized.
- Line 2 shows the addresses of the methods and the time it took to optimize them.

You can use the verbose `opt` information to diagnose and monitor the optimizations.

2.2.2 Verbose exceptions Log Module

The `-Xverbose:exceptions` log module displays each Java exception that is thrown in the application. You can use this information to monitor and troubleshoot exceptions in your application.

Example 2–19 shows output from the verbose `exceptions` log module. Each line displays the name of the exception thrown and the exception message, if one is available.

Example 2–19 Output of the exceptions Module

```
[excepti][00004] java/lang/NullPointerException
[excepti][00004] java/lang/NullPointerException: null array passed into arraycopy
[excepti][00004] java/lang/ArrayIndexOutOfBoundsException
[excepti][00004] java/lang/ArrayIndexOutOfBoundsException
[excepti][00004] java/lang/NullPointerException: null array passed into arraycopy
```

`-Xverbose:exceptions=debug` prints the same information and provides stack traces for each exception.

3

Using Thread Dumps

This chapter describes how to get and use Oracle JRockit JVM thread dumps. For basic information about threads and thread synchronization, see *Oracle JRockit Introduction to the JDK*.

A thread dump is a snapshot of the state of all threads that are part of the process. The state of each thread is presented with a stack trace, which shows the contents of a thread's stack. Some of the threads belong to the Java application you are running, while others are JVM internal threads.

A thread dump reveals information about an application's thread activity that can help you diagnose problems and better optimize application and JVM performance; for example, thread dumps automatically show the occurrence of a deadlock. Deadlocks bring some or all of an application to a complete halt.

This chapter contains the following sections:

3.1 Creating Thread Dumps

To create a thread dump from a process, do either of the following:

- Press Ctrl-Break while the process is running (or send `SIGQUIT` to the process on Linux and Solaris).
- Enter the following at the command line:

  ```
  bin\jrcmd.exe pid print_threads
  ```

The thread dump is displayed at the command line.

> **Note:** For more information about the `jrcmd` command and `Ctrl-Break` handlers, see Section 4, "Running Diagnostic Commands."

3.2 Reading Thread Dumps

This section describes the typical contents of a thread dump. An example thread dump from beginning to end is shown. Example 3–1, Example 3–2, Example 3–3, Example 3–4, and Example 3–5 show the components of a thread dump). Next,

information about the main thread is shown, then all the JVM internal threads, followed by all other Java application threads (if there are any). Finally, information about lock chains is shown.

The example thread dump is taken from a program that creates three threads that are quickly forced into a deadlock. The application threads Thread-0, Thread-1, and Thread-2 correspond to three different classes in the Java code.

3.2.1 The Beginning of the Thread Dump

The thread dump starts with the date and time of the dump, and the version number of the JRockit JVM used. See Example 3–1.

Example 3–1 Initial Information in a Thread Dump

```
===== FULL THREAD DUMP ===============
Wed Feb 21 13:46:45 2007
Oracle JRockit(R) R28.0.0-109-73164-1.5.0_08-20061129-1428-windows-ia32
```

3.2.2 Stack Trace for the Main Application Thread

Example 3–2 shows the stack trace of the main application thread. There is a thread information line, followed by information about locks and a trace of the thread's stack at the moment of the thread dump.

Example 3–2 Main Thread in the Thread Dump

```
"Main Thread" id=1 idx=0x2 tid=48652 prio=5 alive, in native, waiting
-- Waiting for notification on: util/repro/Thread1@0x01226528[fat lock]
at jrockit/vm/Threads.waitForSignal(J)Z(Native Method)
at java/lang/Object.wait(J)V(Native Method)
at java/lang/Thread.join(Thread.java:1095)
^-- Lock released while waiting: util/repro/Thread1@0x01226528[fat lock]
at java/lang/Thread.join(Thread.java:1148)
at util/repro/DeadLockExample.main(DeadLockExample.java:23)
at jrockit/vm/RNI.c2java(IIII)V(Native Method)
-- end of trace
```

After the name and other identification information, the different status messages of the main application thread are printed. The main application thread in Example 3–2 is a running thread (`alive`). It is either executing JVM internal code or user-defined JNI code (`in native`). It is currently waiting for an object to be released (`waiting`). If a thread is waiting for a notification about a lock (by calling `Object.wait()`), this is indicated at the top of the stack trace (`Waiting for notification on`).

3.2.3 Locks and Lock Chains

For each thread, the JRockit JVM prints the following information:

- If the thread is trying to take a lock (to enter a synchronized block), but the lock is already held by another thread, this is indicated at the top of the stack trace, (`Blocked trying to get lock`).
- If the thread is waiting for a notification about a lock (by calling `Object.wait()`), this is indicated at the top of the stack trace (`Waiting for notification`).
- If the thread has taken locks, this is shown in the stack trace. After a line in the stack trace describing a function call is a list of the locks taken by the thread in that

function. This is described as `^-- Holding lock` where the `^--` is a reminder that the lock is taken in the function written above the line with the lock.

The semantics for waiting (for notification) for an object in Java is somewhat complex. First, to enter a synchronized block, you must take the lock for the object, and then you call `wait()` on that object. In the wait method, the lock is released before the thread goes to sleep waiting for a notification. When the thread receives a notification, `wait()` re-takes the lock before returning. If a thread has taken a lock, and is waiting for notification about that lock, the line in the stack trace that describes when the lock was taken is not shown as (`Holding lock`); it is shown as (`Lock released while waiting.`)

All locks are described as `Classname@0xLockID[LockType]`; for example:

```
java/lang/Object@0x105BDCC0[thin lock]
```

`Classname@0xLockID` describes the object to which the lock belongs. The classname is an exact description, the fully qualified classname of the object. `LockID`, is a temporary ID that is valid for a single thread dump. If a thread A holds a lock `java/lang/Object@0x105BDCC0`, and a thread B is waiting for a lock `java/lang/Object@0x105BDCC0` in a single thread dump, then it is the same lock. For subsequent thread dumps, `LockID` might be different, even if a thread holds the same lock. `LockType` describes the JVM internal type of the lock (fat, thin, recursive, or lazy). The status of active locks (monitors) is also shown in stack traces.

3.2.3.1 Presentation of Locks Out of Order

The lines with the lock information might not be correct due to compiler optimizations. This means two things:

- If a thread, in the same function, takes lock A first and then lock B, the order in which they are printed is unspecified.
- If a thread, in method `edit()` calls method `save()`, and takes a lock A in `save()`, the lock might be printed as being taken in `edit()`.

Typically, this is not be a problem. The order of the lock lines never move much from their correct positions. Also, lock lines will never be missing— all locks taken by a thread are shown in the thread dump.

3.2.4 JVM Internal Threads

Example 3–3 shows the traces of JVM internal threads. The threads have been marked as daemon threads, noted by their `daemon` state indicators. Daemon threads are either JVM internal threads (as in this case) or threads marked as daemon threads by `java.lang.Thread.setDaemon()`.

Example 3–3 First and Last Thread in a List of JVM Internal Threads

```
"(Signal Handler)" id=2 idx=0x4 tid=48668 prio=5 alive, in native, daemon
[...]
"(Sensor Event Thread)" id=10 idx=0x1c tid=48404 prio=5 alive, in native, daemon
```

Lock information and stack traces are not printed for the JVM internal threads in Example 3–3. This is the default setting.

If you want to see stack traces for the JVM internal threads, then use the parameter `nativestack=true` when you run the `print_threads` command. At the command line, enter the following:

```
bin\jrcmd.exe <pid> print_threads nativestack=true
```

3.2.5 Other Java Application Threads

Typically, you would be interested in the threads of the Java application you are running (including the main thread). All Java application threads, except the main thread, are shown near the end of the thread dump. Example 3–4 shows the stack traces of three different application threads.

Example 3–4 Additional Application Threads

```
"Thread-0" id=11 idx=0x1e tid=48408 prio=5 alive, in native, blocked
-- Blocked trying to get lock: java/lang/Object@0x01226300[fat lock]
at jrockit/vm/Threads.waitForSignal(J)Z(Native Method)
at jrockit/vm/Locks.fatLockBlockOrSpin(ILjrockit/vm/ObjectMonitor;II)V(Unknown Source)
at
jrockit/vm/Locks.lockFat(Ljava/lang/Object;ILjrockit/vm/ObjectMonitor;Z)Ljava/lang/Object;(Unknown
Source)
at
jrockit/vm/Locks.monitorEnterSecondStage(Ljava/lang/Object;I)Ljava/lang/Object;(Unknown Source)
at jrockit/vm/Locks.monitorEnter(Ljava/lang/Object;)Ljava/lang/Object;(Unknown Source)
at util/repro/Thread1.run(DeadLockExample.java:34)
^-- Holding lock: java/lang/Object@0x012262F0[thin lock]
^-- Holding lock: java/lang/Object@0x012262F8[thin lock]
at jrockit/vm/RNI.c2java(IIII)V(Native Method)
-- end of trace

"Thread-1" id=12 idx=0x20 tid=48412 prio=5 alive, in native, blocked
-- Blocked trying to get lock: java/lang/Object@0x012262F8[thin lock]
at jrockit/vm/Threads.sleep(I)V(Native Method)
at jrockit/vm/Locks.waitForThinRelease(Ljava/lang/Object;I)I(Unknown Source)
at jrockit/vm/Locks.monitorEnterSecondStage(Ljava/lang/Object;I)Ljava/lang/Object;(Unknown Source)
at jrockit/vm/Locks.monitorEnter(Ljava/lang/Object;)Ljava/lang/Object;(Unknown Source)
at util/repro/Thread2.run(DeadLockExample.java:48)
at jrockit/vm/RNI.c2java(IIII)V(Native Method)
-- end of trace

"Thread-2" id=13 idx=0x22 tid=48416 prio=5 alive, in native, blocked
-- Blocked trying to get lock: java/lang/Object@0x012262F8[thin lock]
at jrockit/vm/Threads.sleep(I)V(Native Method)
at jrockit/vm/Locks.waitForThinRelease(Ljava/lang/Object;I)I(Unknown Source)
at jrockit/vm/Locks.monitorEnterSecondStage(Ljava/lang/Object;I)Ljava/lang/Object;(Unknown Source)
at jrockit/vm/Locks.monitorEnter(Ljava/lang/Object;)Ljava/lang/Object;(Unknown Source)
at util/repro/Thread3.run(DeadLockExample.java:65)
^-- Holding lock: java/lang/Object@0x01226300[fat lock]
at jrockit/vm/RNI.c2java(IIII)V(Native Method)
-- end of trace
```

All three threads are in a blocked state (indicated by `blocked`), which means that they are all trying to enter synchronized blocks. Thread-0 is trying to take the lock `Object@0x01226300[fat lock]`, but this lock is held by Thread-2. Both Thread-2 and Thread-1 are trying to take `Object@0x012262F8[thin lock]` but this lock is held by Thread-0. This means that Thread-0 and Thread-2 form a deadlock, while Thread-1 is blocked.

3.2.6 Lock Chains

JRockit JVM automatically detects deadlocked, blocked, and open lock chains among the running threads. Example 3–5 shows all the lock chains created by the threads T1, T2, T3, T4, and T5. This information can be used to tune and troubleshoot your Java code.

Example 3–5 Deadlocked and Blocked Lock Chains

```
Circular (deadlocked) lock chains
=================================
Chain 6:
"Dead T1" id=16 idx=0x48 tid=3648 waiting for java/lang/Object@0x01225018 held by:
"Dead T3" id=18 idx=0x50 tid=900 waiting for java/lang/Object@0x01225010 held by:
"Dead T2" id=17 idx=0x4c tid=3272 waiting for java/lang/Object@0x01225008 held by:
"Dead T1" id=16 idx=0x48 tid=3648
Blocked lock chains
===================
Chain 7:
"Blocked T2" id=20 idx=0x58 tid=3612 waiting for java/lang/Object@0x01225310 held by:
"Blocked T1" id=19 idx=0x54 tid=2500 waiting for java/lang/Object@0x01224B60 held by:
"Open T3" id=13 idx=0x3c tid=1124 in chain 1
Open lock chains
================
Chain 1:
"Open T5" id=15 idx=0x44 tid=4048 waiting for java/lang/Object@0x01224B68 held by:
"Open T4" id=14 idx=0x40 tid=3380 waiting for java/lang/Object@0x01224B60 held by:
"Open T3" id=13 idx=0x3c tid=1124 waiting for java/lang/Object@0x01224B58 held by:
"Open T2" id=12 idx=0x38 tid=3564 waiting for java/lang/Object@0x01224B50 held by:
"Open T1" id=11 idx=0x34 tid=2876 (active)
```

3.3 Thread Status in Thread Dumps

This section describes the different statuses or states of a thread in a thread dump:

- Section 3.3.1, "Life States"
- Section 3.3.2, "Run States"
- Section 3.3.3, "Special States"

3.3.1 Life States

Table 3–1 describes the life states that a thread can show in a thread dump.

Table 3–1 Thread Life States

State	Description
`alive`	This is a typical, running thread. Virtually all threads in the thread dump will be (alive).
`not started`	The thread was requested to start running by `java.lang.Thread.start()`, but the actual OS process has not yet started or executed far enough to pass control to the JRockit JVM. It is unlikely to see this value. A `java.lang.Thread` object that is created, but has not had `start()` executed, will not show in the thread dump.
`terminated`	This thread has finished its `run()` and has also notified any threads joining on it, but it is still kept in the JVM internal thread structure for running threads. It is unlikely to see this value. A thread that has been terminated for a time longer than a few milliseconds will not show in the thread dump.

3.3.2 Run States

Table 3–2 describes the run states that a thread can show in a thread dump.

Table 3–2 Thread Run States

State	Description
`blocked`	This thread tried to enter a synchronized block, but the lock was taken by another thread. This thread is blocked until the lock gets released.
`blocked (on thin lock)`	This is the same state as `blocked`, but the lock in question is a thin lock.
`waiting`	This thread called `Object.wait()` on an object. The thread will remain there until some other thread sends a notification to that object.
`sleeping`	This thread called `java.lang.Thread.sleep()`.
`parked`	This thread called `java.util.concurrent.locks.LockSupport.park()`.
`suspended`	The thread's execution was suspended by `java.lang.Thread.suspend()` or a JVMTI agent call.

Note: Each state in Table 3–2 represents a scenario where the thread is currently blocked (that is, ineligible for scheduling by the operating system) for some (ideally temporary) reason.

Java execution threads that are not currently blocked (that is, eligible for scheduling by the operating system to run) will not have any of the above states.

3.3.3 Special States

Table 3–3 describes the special states that a thread can show in a thread dump. Note that these states are not mutually exclusive.

Table 3–3 Special Thread States

State	Description
`interrupted`	The user called `java.lang.Thread.interrupt()` on this thread.
`daemon`	This is either JVM internal thread or a thread that was marked as a daemon thread by `java.lang.Thread.setDaemon()`.
`in native`	This thread is executing native code: either user-supplied JNI code or JVM internal code.
`in suspend critical mode`	This thread is executing JVM internal code and has marked itself as `suspend critical`. Garbage collection is stopped for a specified time period.
`native_blocked`	This thread is executing JVM internal code and has tried to take a JVM internal lock. The thread is blocked because that lock is held by another thread.
`native_waiting`	This thread is executing JVM internal code and is waiting for notification from another thread about a JVM internal lock.

Note: None of the above states in Table 3–3 by themselves represent a situation where a Java execution thread would be blocked from making progress. By design, when a thread dump is collected, all Java execution threads must be paused momentarily in a known state in order to collect the necessary data (stack traces). As a result, in a thread dump, every Java execution thread will be in one of these states—`in native`, `native_blocked`, or `native_waiting`. As soon as the thread dump is finished, these threads will be able to continue making progress (as long as these threads are not also in one of the states listed in Table 3–2). For analyzing the Java application behavior, these threads should not be considered blocked.

3.4 Troubleshooting with Thread Dumps

This section contains information about using thread dumps for troubleshooting and diagnostics.

To use thread dumps for troubleshooting, beyond detecting deadlocks, you must take several thread dumps from the same process. For analyzing the behavior over longer durations, a better option would be to combine thread dumps with other diagnostic tools, such as the JRockit Flight Recorder, which is part of Oracle JRockit Mission Control.

3.4.1 Detecting Deadlocks

The Oracle JRockit JVM automatically analyzes the thread dump information and detects whether there exists any circular (deadlocked) or blocked lock chains in it. A blocked lock chain is not a deadlock: it indicates contention.

3.4.2 Detecting Processing Bottlenecks

To detect more than deadlocks in your threads, you must make several consecutive thread dumps. This lets you detect the occurrence of contention, when multiple threads are trying to get the same lock. Contention might create long open lock chains that, while not deadlocked, will degrade performance.

If you discover (in a set of consecutive thread dumps) that one or more threads in your application are temporarily stuck waiting for a lock to be released, then review the code of your Java application to see if the synchronization (serialization) is necessary or if the threads can be organized differently.

3.4.3 Viewing the Run-time Profile of an Application

By making several consecutive thread dumps, you can get an overview of the parts of your Java application that are used the most. Click the **Threads** tab in JRockit Management Console for more detailed information about the workload on the different parts of your application.

4

Running Diagnostic Commands

This chapter describes how to use diagnostic commands to communicate with a running Oracle JRockit JVM process.

The diagnostic commands tell the JRockit JVM to perform tasks such as printing a heap report or a garbage collection activity report, or enabling a specific verbose module.

This chapter contains the following sections:

For more information about each diagnostic command, see the "Diagnostic Commands" chapter in *Oracle JRockit Command Line Reference*.

4.1 Methods for Running Diagnostic Commands

You can send diagnostic commands to a running JVM process in several ways:

- By using jrcmd, a command-line tool that sends the commands to a given JRockit JVM process.
- By pressing Ctrl-Break, the JVM will search for the `ctrlhandler.act` file and execute the commands in it.
- By using the JRockit Management Console in Oracle JRockit Mission Control to send diagnostic commands to a running JRockit JVM process.

You can enable or disable diagnostic command by using the system property `-Djrockit.ctrlbreak.enable<name>=<true|false>`, where `name` is the name of the diagnostic command. The `run_class` handler is not enabled by default. To enable it, enter the following:

```
-Djrockit.ctrlbreak.enablerun_class=true
```

4.2 Using jrcmd

`jrcmd` is a command-line tool included with the JRockit JDK that you can use to send diagnostic commands to a running JVM process. `jrcmd` communicates with the JVM by using the JDK attach mechanism. This section provides a brief overview of `jrcmd`. It includes information about the following:

- Section 4.2.2, "jrcmd Examples"
- Section 4.2.3, "Known Limitations of `jrcmd`"

4.2.1 How to Use jrcmd

Enter `jrcmd` at the command line with the appropriate parameters.

Example:

```
jrcmd <jrockit pid> [<command> [<arguments>]] [-l] [-f file] [-p] -h]
```

- `<jrockit pid>` is either the process ID or the name of the Main class that runs the application.
- `[<command> [<arguments>]]` is any diagnostic command and its associated arguments; for example, `version`, `print_properties`, `command_line`.
- `-l` displays the counters exposed by this process. These counters are for internal use by Oracle and are not officially supported or documented.
- `-f` reads and executes commands from the file.
- `-p` lists JRockit JVM processes on the local machine.
- `-h` displays help.

If the PID is 0, commands will be sent to all Jrockit JVM processes. If no options are given, the default is `-p`. If you are running a Flight Recorder-specific diagnostic command and use 0 as the PID, the recorder does not remain active.

Note: `jrcmd` parameter strings are limited to 256 characters. If the parameter string exceeds this limit, an exception will be thrown.

4.2.2 jrcmd Examples

This section provides examples of using `jrcmd` for the following tasks:

- Listing JRockit JVM Processes
- Sending a Command to a Process
- Sending Several Commands

4.2.2.1 Listing JRockit JVM Processes

Do the following to list all JRockit JVM processes running on the machine:

- Run `jrcmd` or `jrcmd -p` to list the running JRockit JVMs; for example:

```
> jrcmd -P
10064 Sleeper
        -Xverbose:memory -Xmx30m
>
```

You will see the PID of the process (`10064`), the program it is currently running (`Sleeper`), and the parameters used to start the JVM (`-Xverbose:memory -Xmx30m`).

4.2.2.2 Sending a Command to a Process

To send a command to the process you identified in Section 4.2.2.1, "Listing JRockit JVM Processes", do the following:

1. Find the PID. See Section 4.2.2.1, "Listing JRockit JVM Processes" (`10064`).

2. Run `jrcmd` with that PID and the `version` command; for example:

```
> jrcmd 10064 version
```

This command sends the `version` command to the JRockit JVM. The response will be:

```
Oracle JRockit(R) build R28.0.0-679-130297-1.6.0_17-20100312-2122-windows-x86_
64, compiled mode
GC mode: Garbage collection optimized for throughput, strategy: genparpar
```

4.2.2.3 Sending Several Commands

You can create a file (similar to the `ctrlhandler.act` file) that contains several commands and execute all of them. Use this procedure:

1. Create a file called commands.txt with the following contents:
   - `version`
   - `timestamp`
2. Execute the file with `jrcmd`; for example:

```
> jrcmd 10064 -f commands.txt
```

The system will respond:

```
Oracle JRockit(R) build R28.1.0-102-137086-1.6.0_20-20100825-2121-windows-ia32,
compiled mode
GC mode: Garbage collection optimized for throughput, strategy: genparpar
==== Timestamp ====  uptime: 0 days, 02:41:21 time: Mon Sep 06 16:23:43 2010
```

3. Set the PID to `0` to send the commands to all running JRockit JVM processes.

4.2.3 Known Limitations of `jrcmd`

When using `jrcmd`, be aware of these limitations:

- In order to issue diagnostic commands to a process on Linux or Solaris, you must run `jrcmd` with the same user as the one running the Java process.
- When using `jrcmd` on Windows, you must run the Java process and `jrcmd` from the same Windows station. If you run the Java process as a Windows service and run `jrcmd` on your desktop, it will not work, because they are running in two separate environments.
- When a JRockit JVM is started as the root user and then changed to a less-privileged user, security restrictions will prevent `jrcmd` from communicating properly with the process.

  The following actions are permitted:

  – Root can list the running processes.

  – The less-privileged user can send commands to the process.

  The following actions are prohibited:

  – Root cannot send commands to the process; commands will be treated as a Ctrl-Break signal and print a thread dump instead.

  - Less privileged users cannot list the running JRockit JVM process but if they know the process ID (PID), they can send commands to the process using `jrcmd <pid> <command>`.

- If the default Windows temporary directory (java.io.temp) is on a FAT file system, `jrcmd` cannot discover local processes. For security reasons, local monitoring and management is only supported if your default Windows temporary directory is on a file system that supports setting permissions on files and directories (for example, on an NTFS file system). It is not supported on FAT file systems, which provide insufficient access controls.

4.3 Using the Ctrl-Break Handler

Another way you can run diagnostic commands is by pressing Ctrl-Break. When you press Ctrl-Break, the JRockit JVM will search for a file named `ctrlhandler.act` (see Example 4–1) in your current working directory. If it cannot find the file there, it will search in the directory that contains the JVM. If it cannot find this file there, it will revert to printing the thread dump. If it finds the file, it will read the file searching for command entries, each of which invoke the corresponding diagnostic command.

Example 4–1 ctrlhandler.act File

```
set_filename filename=c:\output.txt append=true
print_class_summary
print_object_summary increaseonly=true
print_threads
print_threads nativestack=true
print_utf8pool
jrarecording filename=c:\myjra.xml time=120 nativesamples=true
verbosity set=memory,memdbg,codegen,opt,sampling filename="c:\output"
timestamp

stop
# ctrl-break-handler will stop reading the file after it finds
# the stop key-word
#
# version - print JRockit version
#
# print_threads - the normal thread dump that lists all the currently
# running threads and there state
#
# print_class_summary - prints a tree of the currently loaded classes
#
# print_utf8pool - print the internal utf8 pool
#
# print_object_summary - display info about how many objects of each
# type that are live currently and how much size
# they use. Also displays points to information
#
# jrarecording - starts a jrarecording
#
# verbosity - changes the verbosity level , not functional in ariane142_04
#
# start_management_server - starts a management server
# kill_management_server - shuts the management server down
#   (the managementserver.jar has to be in the bootclasspath for
#   these command to work)
#
#
```

In the `ctrlhandler.act` file, each command entry starts with a Ctrl-Break handler name followed by the arguments to be passed to the Ctrl-Break handler. The arguments should be in the proper format (that is, `name=value`; for example, `set_filename filename=c:\output.txt append=true`). The acceptable property types are string, integer, and Boolean values.

You can disable Ctrl-Break functionality by using the following option:

```
-Djrockit.dontusectrlbreakfile=true
```

4.4 Getting Help

To get help about the available commands, use the command `help`. This will print all available commands.

For more information about each diagnostic command, see the "Diagnostic Commands" chapter in *Oracle JRockit Command Line Reference*.