

Red Hat Enterprise MRG 1.1

Realtime Tuning Guide

Advanced tuning procedures for the Realtime
component of Red Hat Enterprise MRG



Lana Brindley

Red Hat Enterprise MRG 1.1 Realtime Tuning Guide

Advanced tuning procedures for the Realtime component of Red Hat Enterprise MRG

Edition 2

Author

Lana Brindley

lbrindle@redhat.com

Copyright © 2008 Red Hat, Inc

Copyright © 2008 Red Hat, Inc. This material may only be distributed subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version of the OPL is presently available at <http://www.opencontent.org/openpub/>).

Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588 Research Triangle Park, NC 27709 USA

This book contains advanced tuning procedures for the MRG Realtime component of the Red Hat Enterprise MRG distributed computing platform. For installation instructions, see the MRG Realtime Installation Guide.

Preface	v
1. Document Conventions	v
1.1. Typographic Conventions	vi
1.2. Pull-quote Conventions	vii
1.3. Notes and Warnings	viii
2. We Need Feedback!	viii
1. Before you start tuning your MRG Realtime system	1
2. General System Tuning	3
2.1. Using the Tuna Interface	3
2.2. Setting persistent tuning parameters	9
2.3. Interrupt and Process Binding	10
2.4. Filesystem determinism tips	14
2.5. gettimeofday speedup	15
2.6. Don't run extra stuff	16
2.7. Swapping and Out Of Memory Tips	17
2.8. Network determinism tips	18
2.9. syslog tuning tips	19
2.10. The PC Card Daemon	21
2.11. Reduce TCP performance spikes	21
2.12. Reducing the TCP delayed ack timeout	22
3. Realtime-Specific Tuning	23
3.1. Setting Scheduler Priorities	23
3.2. MRG Realtime Specific gettimeofday speedup	25
3.3. Using kdump and kexec with the MRG Realtime kernel	26
3.4. TSC timer synchronization on Opteron CPUs	30
3.5. Infiniband	31
3.6. Non-Uniform Memory Access	31
3.7. Using the ftrace Utility for Tracing Latencies	32
3.8. Latency Tracing Using trace-cmd	36
3.9. Using sched_nr_migrate to limit SCHED_OTHER processes.	38
4. Application Tuning and Deployment	39
4.1. Signal Processing in Realtime Applications	39
4.2. Using sched_yield and Other Synchronization Mechanisms	40
4.3. Mutex options	40
4.4. TCP_NODELAY and Small Buffer Writes	42
4.5. Setting Realtime Scheduler Priorities	43
4.6. Dynamic Libraries Loading	44
5. More Information	47
5.1. Reporting Bugs	47
5.2. Further Reading	48
A. Revision History	49

Preface

Red Hat Enterprise MRG

This book contains basic installation and tuning information for the MRG Realtime component of Red Hat Enterprise MRG. Red Hat Enterprise MRG is a high performance distributed computing platform consisting of three components:

1. *Messaging* — Cross platform, high performance, reliable messaging using the Advanced Message Queuing Protocol (AMQP) standard.
2. *Realtime* — Consistent low-latency and predictable response times for applications that require microsecond latency.
3. *Grid* — Distributed High Throughput (HTC) and High Performance Computing (HPC).

All three components of Red Hat Enterprise MRG are designed to be used as part of the platform, but can also be used separately.

MRG Realtime

Many industries and organizations need extremely high performance computing and may require low and predictable latency, especially in the financial and telecommunications industries. Latency, or response time, is defined as the time between an event and system response and is generally measured in microseconds (μ s). For most applications running under a Linux environment, basic performance tuning can improve latency sufficiently. For those industries where latency not only needs to be low, but also accountable and predictable, Red Hat have now developed a 'drop-in' kernel replacement that provides this. MRG Realtime is distributed as part of Red Hat Enterprise MRG and provides seamless integration with Red Hat Enterprise Linux 5.2. MRG Realtime offers clients the opportunity to define, measure, configure and record latency times across their organization.

About The MRG Realtime Tuning Guide

This book is laid out in three main sections: General system tuning, which can be performed on a Red Hat Enterprise Linux 5.2 kernel and MRG Realtime specific tuning, which should be performed on a MRG Realtime kernel in addition to the standard Red Hat Enterprise Linux 5.2 tunes. The third section is for developing and deploying your own MRG Realtime programs.

You will need to have the MRG Realtime kernel installed before you begin the tuning procedures in this book. If you have not yet installed the MRG Realtime kernel, or need help with installation issues, read the *MRG Realtime Deployment Guide*.

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](https://fedorahosted.org/liberation-fonts/)¹ set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

¹ <https://fedorahosted.org/liberation-fonts/>

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

Mono-spaced Bold

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

To see the contents of the file **my_next_bestselling_novel** in your current working directory, enter the **cat my_next_bestselling_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

Press **Enter** to execute the command.

Press **Ctrl-Alt-F1** to switch to the first virtual terminal. Press **Ctrl-Alt-F7** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **Mono-spaced Bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh *john@example.com***.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount */home***.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules (MPMs)*. Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.

Output sent to a terminal is set in Mono-spaced Roman and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Source-code listings are also set in Mono-spaced Roman but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;
```

```
public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object          ref    = iniCtx.lookup("EchoBean");
        EchoHome        home   = (EchoHome) ref;
        Echo            echo    = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



Note

A Note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.



Warning

A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: <http://bugzilla.redhat.com/bugzilla/> against the product **Red Hat Enterprise MRG**.

When submitting a bug report, be sure to mention the manual's identifier: *Realtime_Tuning_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Before you start tuning your MRG Realtime system

MRG Realtime is designed to be used on well-tuned systems for applications with extremely high determinism requirements. Kernel system tuning offers the vast majority of the improvement in determinism. For example, in many workloads thorough system tuning improves consistency of results by around 90%. This is why we typically recommend that customers first perform the [Chapter 2, General System Tuning](#) of standard Red Hat Enterprise Linux before using MRG Realtime.

Things to remember while you are tuning your MRG Realtime kernel

1. Be Patient

Realtime tuning is an iterative process; you will almost never be able to tweak a few variables and know that the change is the best that can be achieved. Be prepared to spend days or weeks narrowing down the set of tunings that work best for your system.

Additionally, always make long test runs. Changing some tuning parameters then doing a five minute test run is not a good validation of a set of tunes. Make the length of your test runs adjustable and run them for longer than a few minutes. Try to narrow down to a few different tuning sets with test runs of a few hours, then run those sets for many hours or days at a time, to try and catch corner-cases of max latencies or resource exhaustion.

2. Be Accurate

Build a measurement mechanism into your application, so that you can accurately gauge how a particular set of tuning changes affect the application's performance. Anecdotal evidence (e.g. "The mouse moves more smoothly") is usually wrong and varies from person to person. Do hard measurements and record them for later analysis.

3. Be Methodical

It is very tempting to make multiple changes to tuning variables between test runs, but doing so means that you do not have a way to narrow down which tune affected your test results. Keep the tuning changes between test runs as small as you can.

4. Be Conservative

It is also tempting to make large changes when tuning, but it is almost always better to make incremental changes. You will find that working your way up from the lowest to highest priority values will yield better results in the long run.

5. Be Smart

Use the tools you have available. The Tuna graphical tuning tool makes it easy to change processor affinities for threads and interrupts, thread priorities and to isolate processors for application use. The **taskset** and **chrt** command line utilities allow you to do most of what Tuna does. If you run into performance problems, the **ftrace** facility in the trace kernel can help locate latency issues.

6. Be Flexible

Rather than hard-coding values into your application, use external tools to change policy, priority and affinity. This allows you to try many different combinations and simplifies your logic. Once you have found some settings that give good results, you can either add them to your application, or set up some startup logic to implement the settings when the application starts.

How Tuning Improves Performance

Most performance tuning is performed by manipulating processors (Central Processing Units or CPUs). Processors are manipulated through:

Interrupts:

In software, an interrupt is an event that calls for a change in execution.

Interrupts are serviced by a set of processors. By adjusting the affinity setting of an interrupt we can determine on which processor the interrupt will run.

Threads:

Threads provide programs with the ability to run two or more tasks simultaneously.

Threads, like interrupts, can be manipulated through the affinity setting, which determines on which processor the thread will run.

It is also possible to set scheduling priority and scheduling policies to further control threads.

By manipulating interrupts and threads off and on to processors, you are able to indirectly manipulate the processors. This gives you greater control over scheduling and priorities and, subsequently, latency and determinism.

MRG Realtime Scheduling Policies

Linux uses three main scheduling policies:

SCHED_OTHER (sometimes called **SCHED_NORMAL**)

This is the default thread policy and has dynamic priority controlled by the kernel. The priority is changed based on thread activity. Threads with this policy are considered to have a realtime priority of 0 (zero).

SCHED_FIFO (First in, first out)

A realtime policy with a priority range of from 1 - 99, with 1 being the lowest and 99 the highest.

SCHED_FIFO threads always have a higher priority than **SCHED_OTHER** threads (for example, a **SCHED_FIFO** thread with a priority of 1 will have a higher priority than *any* **SCHED_OTHER** thread). Any thread created as a **SCHED_OTHER** thread has a fixed priority and will run until it is blocked or preempted by a higher priority thread.

SCHED_RR (Round-Robin)

SCHED_RR is an optimization of **SCHED_FIFO**. Threads with the same priority have a quantum and are round-robin scheduled amongst all equal priority **SCHED_RR** threads. This policy is rarely used.

General System Tuning

This section contains general tuning that can be performed on a standard Red Hat Enterprise Linux installation. It is important that these are performed first, in order to better see the benefits of the MRG Realtime kernel.

It is recommended that you read these sections first. They contain background information on how to modify tuning parameters and will help you perform the other tasks in this book:

- [Section 2.1, “Using the Tuna Interface”](#)
- [Section 2.2, “Setting persistent tuning parameters”](#)

When are you ready to begin tuning, perform these steps first, as they will provide the greatest benefit:

- [Section 2.3, “Interrupt and Process Binding”](#)
- [Section 2.4, “Filesystem determinism tips”](#)

When you are ready to start some fine-tuning on your system, then try the other sections in this chapter:

- [Section 2.5, “**gettimeofday** speedup”](#)
- [Section 2.6, “Don’t run extra stuff”](#)
- [Section 2.7, “Swapping and Out Of Memory Tips”](#)
- [Section 2.8, “Network determinism tips”](#)
- [Section 2.9, “**syslog** tuning tips”](#)
- [Section 2.10, “The PC Card Daemon”](#)
- [Disabling the **pcscd** Daemon](#)
- [Section 2.11, “Reduce TCP performance spikes”](#)
- [Section 2.12, “Reducing the TCP delayed ack timeout”](#)

When you have completed all the tuning suggestions in this chapter, move on to [Chapter 3, Realtime-Specific Tuning](#)

2.1. Using the Tuna Interface

Throughout this book, instructions are given for tuning the MRG Realtime kernel directly. The Tuna interface is a tool that assists you with making changes. It has a graphical interface, or can be run through the command shell.

Tuna can be used to change attributes of threads (scheduling policy, scheduler priority and processor affinity) and interrupts (processor affinity). The tool is designed to be used on a running system, and changes take place immediately. This allows any application-specific measurement tools to see and analyze system performance immediately after the changes have been made.

Although changes made in Tuna take immediate effect, changes will not be persistent across reboots. For making persistent changes, see [Section 2.2, “Setting persistent tuning parameters”](#)

Installing Tuna Using Yum

1. Install Tuna using the **yum** command.

```
# yum install tuna
```

2. Although Tuna can be run as an unprivileged user, not all processes will be available for configuration. For this reason, in most cases you will need to run Tuna as the root user:

```
# tuna
```

Using Tuna from the shell prompt

1. Use the **--help** option to see all the available options. These are also listed at [Table 2.1, “Tuna Options”](#):

```
# tuna --help
Usage: tuna [OPTIONS]
-h, --help                Give this help list
-g, --gui                 Start the GUI
...[output truncated]...
```

2. Use the **--show_threads** command to view the current policies and priorities:

```
# tuna --threads 7861 --show_threads
thread      ctxt_switches
pid SCHED_ rtpri affinity voluntary nonvoluntary  cmd
    7861   FIFO    50      0xff      31251      15915  IRQ-4 serial
```

To change policy and priority information on threads, use the **--priority=[POLICY]:RTPRIO** command:

```
# tuna --threads 7861 --priority=RR:40
```

Use the **--show_threads** command to check the changes:

```
# tuna --threads 7861 --show_threads
thread      ctxt_switches
pid SCHED_ rtpri affinity voluntary nonvoluntary  cmd
    7861    RR    40      0xff      33318      16957  IRQ-4 serial
```

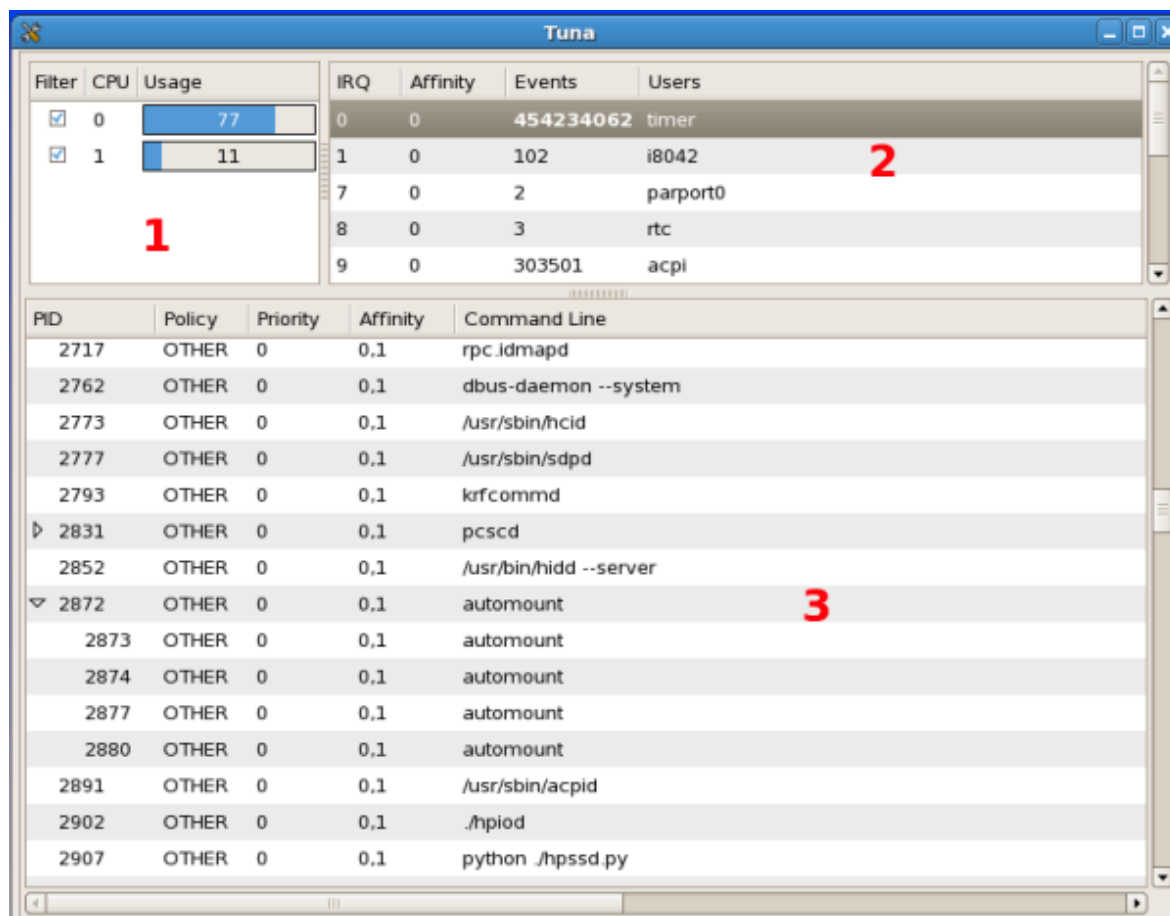
3. When passing commands to Tuna using the command line, it is possible to pass multiple commands in one line. In this case, Tuna will process the commands sequentially. For example,

using the commands from the previous step, instead of typing each command individually, they could be passed on a single line, and Tuna will process them in order:

```
# tuna --threads 7861 --show_threads --priority FIFO:50 --show_threads
thread      ctxt_switches
pid SCHED_  rtpri affinity voluntary nonvoluntary  cmd
   7861      RR    40    0xff    55504      28254  IRQ-4 serial
thread      ctxt_switches
pid SCHED_  rtpri affinity voluntary nonvoluntary  cmd
   7861  FIFO    50    0xff    55508      28256  IRQ-4 serial
```

Tuna Options	
--help	Display the help list
--gui	Start the graphical user interface
--cpus=<i>CPU-LIST</i>	The CPUs to be controlled by Tuna
--affect_children	Operation will affect children threads as well as the parent threads
--filter	Filter the display to only show the affected entities
--isolate <i>CPU-LIST</i>	Move all threads away from the specified CPUs
--include <i>CPU-LIST</i>	Allow all threads to run on the specified CPUs
no_kthreads	Operation will not effect kernel threads
--move <i>CPU-LIST</i>	Move selected entities to the specified CPUs
--priority=[<i>POLICY</i>]:<i>RTPRIO</i>	Set the thread to have the specified scheduler policy and priority
--show_threads	Show the thread list
--save=<i>FILE NAME</i>	Used to specify a file name to save the kernel threads scheduler policy, RT priority and processor affinity. This information is saved in the same format used in /etc/groups
--sockets=<i>CPU-SOCKET-LIST</i>	The CPU sockets to be controlled by Tuna. This option takes into account the CPU topology, such as the cores that share a single processor cache, and that are on the same physical chip.
--threads=<i>THREAD-LIST</i>	The threads to be controlled by Tuna
--no_uthreads	Operation will not affect user threads
--spread	Spread the specified threads evenly between the selected CPUs
--what_is	To see further help on selected entities

Table 2.1. Tuna Options



The main Tuna window is divided into three sections. The window can be resized and the sections are divided by grab bars for adjustment. As values change, entries are shown in bold.

1. The CPU List

This list shows all online CPUs and their current usage.

The check-box beside the name of the CPU is used to filter the task list at the bottom of the window. Only tasks that belong to checked CPUs will be displayed.

Right-click on a CPU to display isolation options. Selecting **Isolate CPU** will cause all tasks currently running on that CPU to move to the next available CPU. This can be chosen on one or more CPUs simultaneously, depending on how many CPUs are available on your system.

2. The IRQ List

This list shows all the active Interrupt Requests (IRQs), their process ID (PID) and policy and priority information.

The IRQ list has a right-click menu. The **Refresh IRQ list** option is provided as IRQs affinity changes may not occur until the next interrupt. Select **Set IRQ Attributes** to open the IRQ Attributes dialog box.

3. The Task List

This list shows all tasks except kernel threads.

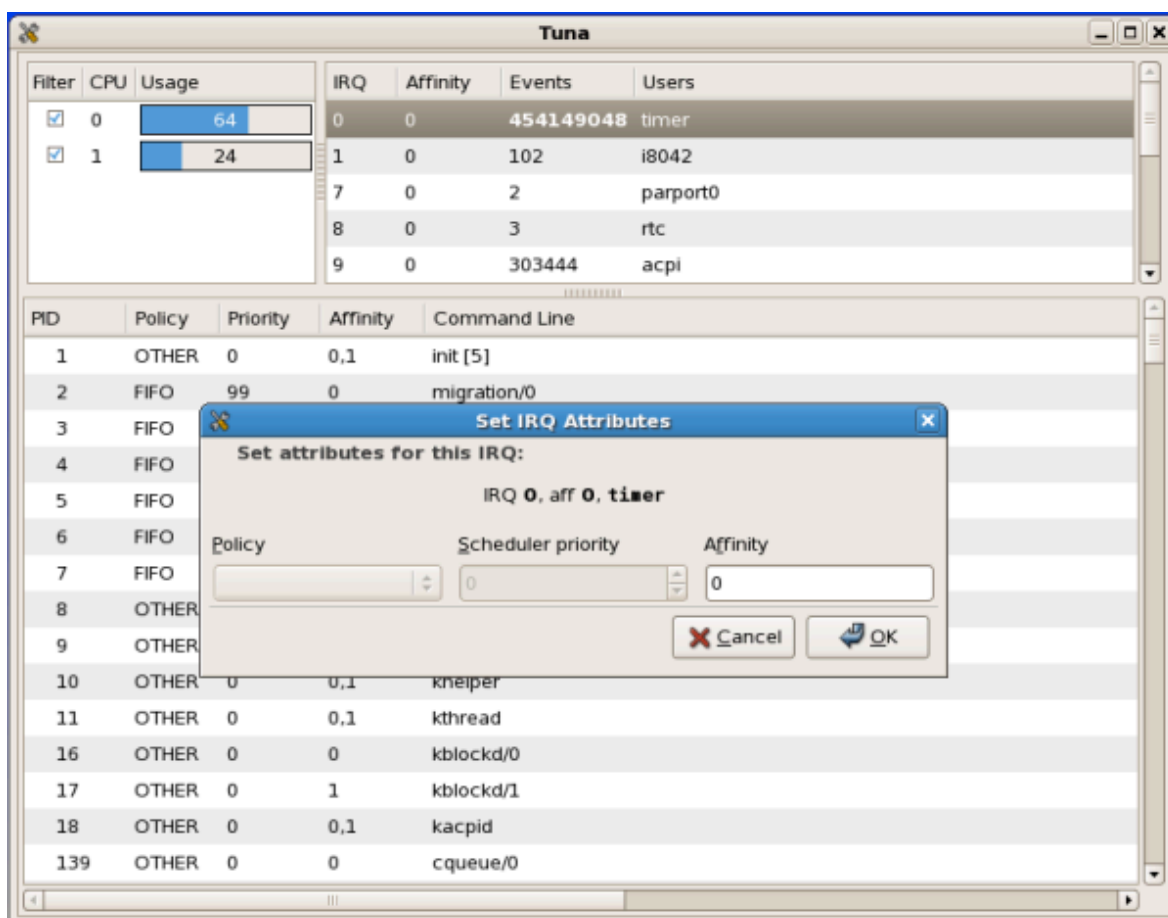
When a process is threaded, the task list shows the original thread with all the other threads collapsed below it. Click on the arrow to the left of the process to expand the thread.

The right-click menu on the task list is similar to that of the IRQ list. Use **Refresh task list** will refresh the list with any changes and the **Set process attributes** will open the Set Process Attributes dialog box.



Important

Any IRQ with a PID of 0 (zero) is a NODELAY IRQ and is not implemented as a kernel thread. Setting the scheduling policy and priority for NODELAY IRQs will have no effect.



Right click on an IRQ and select **Set IRQ Attributes** to open the IRQ Attributes dialog box.

The IRQ Attributes dialog shows current information about the IRQ. It has three adjustable attributes:

1. Scheduling Policy

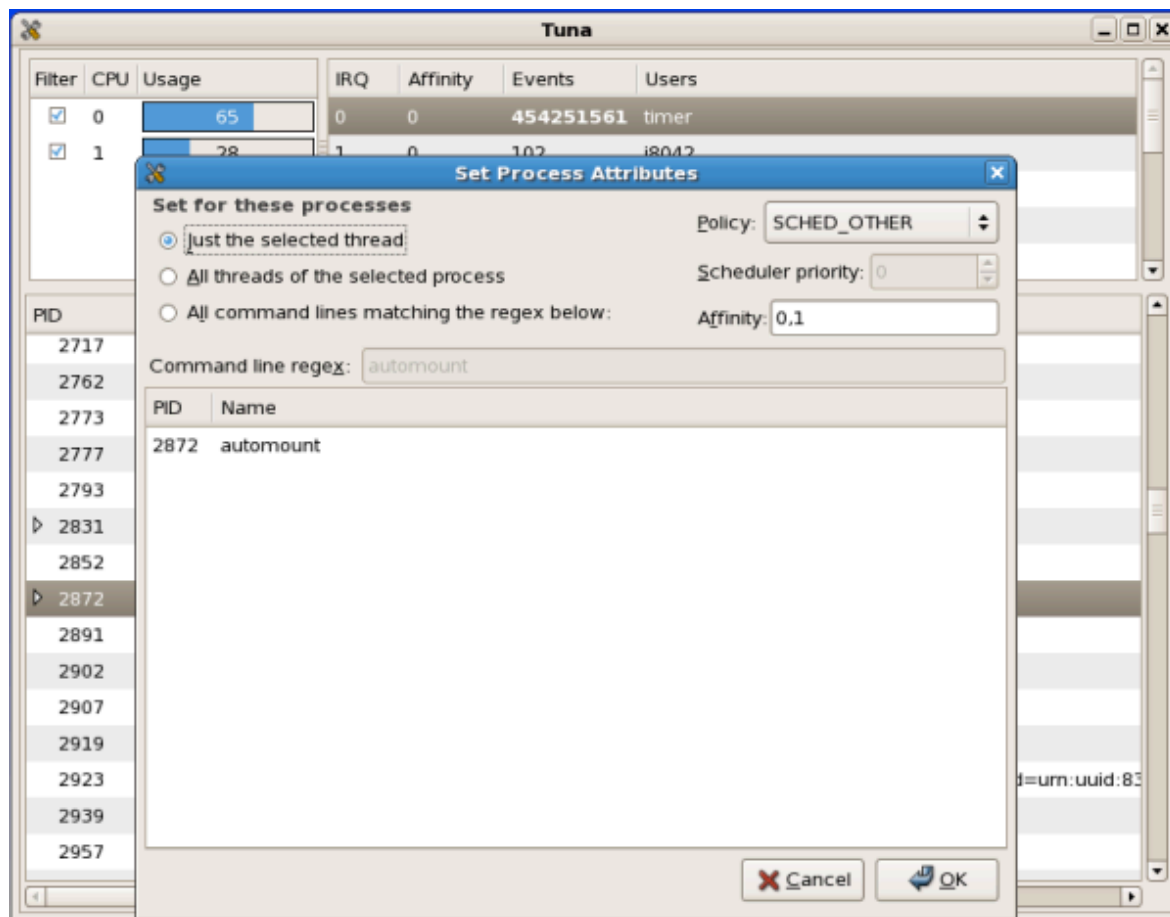
A drop down list of the available policies.

2. Scheduler Priority

A drop down list of the available priorities. This attribute will be disabled if the selected IRQ cannot have a set priority.

3. Affinity

A numeric list of CPUs on which the IRQ can be run. This entry can be in the form of a comma-delimited list of CPU numbers, a range using square brackets, or a combination of both. For example: **0, [2-4], 7, 8**. This would instruct the IRQ to run on CPUs 0, 2, 3, 4, 7 and 8.



Right click on a task and select **Set Process Attributes** to open the Process Attributes dialog box.

The Process Attributes dialog shows current information about the task. It allows you to set scheduling policy, scheduler priority, and CPU affinity for a task or set of tasks.

1. Thread Selection

Just the selected thread is selected by default. If the task has more than one thread, use **All threads of the selected process** to make changes to all of the threads for that task. To use a regular expression (regex) to search for tasks, select **All command lines matching the regex below:**. This will activate the **Command line regex:** field and you can enter the regex. This field supports the ***** and **?** wildcards, and will match the entire command line. The task list will update to show only those tasks that match the regex.

2. Policy, Priority and Affinity

The **Policy** drop down box contains the available scheduling policy options.

The **Scheduler Priority** drop down box contains the available priorities. This attribute will be disabled if the selected tasks cannot have a set priority.

The **Affinity** field contains a numeric list of CPUs on which the selected tasks can be run. This entry can be in the form of a comma-delimited list of CPU numbers, a range using square brackets, or a combination of both.

3. Task List

This shows a list of the the tasks currently being adjusted based on the thread and regex selections made.

Using Tuna - An Example

Suppose you have a system with 4 or more processors, and two applications - **Foo** and **Bar**. You want to run the applications on dedicated processors and choose processor 1 for **Foo** and processor 2 for **Bar**.

1. The first thing to do is move everything off the chosen processors. Right-click on **CPU 1** in the CPU list and select **Isolate CPU** from the menu. Repeat for **CPU 2**. The task list shows you that none of the tasks are running on those processors now.
2. Suppose that **Foo** is a single task with several threads, and you want the task and all its threads running on **CPU 1**. Find **Foo** in the process list, right-click on it and choose **Set process attributes** from the menu. In the **Set Process Attributes** dialog, select the radio button for **All threads of the selected process**. In the **Affinity** text box, change the text to **1**. You can also choose to change the scheduling policy and scheduler priority at this time. Click on **OK** to save your changes and close the dialog box.
3. Suppose that **Bar** is an application that has **--none** as its first command line argument. Right-click anywhere in the task list and choose **Set process attributes** from the menu. In the dialog, select the radio-button for **All command lines matching the regex below:**. Type **bar --none *** in the **Command line regex** text box. You will see the task list in the dialog box update to include all the matching processes (including all threads). Change the **Affinity** to **2**. Make any changes you want for the scheduler and priority. Click on **OK** to save your changes and close the dialog box.

2.2. Setting persistent tuning parameters

This book contains many examples on how to specify kernel tuning parameters. Unless stated otherwise, the instructions will cause the parameters to remain in effect until the system reboots or they are explicitly changed. This approach is effective for establishing the initial tuning configuration.

Once you have decided what tuning configuration works for your system, you will probably want those parameters to be persistent across reboots. Which method you choose depends on the type of parameter you are setting.

Editing the `/etc/sysctl.conf` file

For any parameter that begins with `/proc/sys/`, including it in the `/etc/sysctl.conf` file will make the parameter persistent.

1. Open the `/etc/sysctl.conf` file in your chosen text editor
2. Remove the `/proc/sys/` prefix from the command and replace the central `/` character with a `.` character.

For example: the command `echo 2 > /proc/sys/kernel/vsyscall164` will become `kernel.vsyscall164`.

3. Insert the new entry into the `/etc/sysctl.conf` file with the required parameter

```
# Enable gettimeofday(2)
kernel.vsyscall164 = 2
```

4. Run `# sysctl -p` to refresh with the new configuration

```
# sysctl -p
...[output truncated]...
kernel.vsyscall164 = 2
```

Editing files in the `/etc/sysconfig/` directory

Files in the `/etc/sysconfig/` directory can be added for most other parameters. As files in this directory can differ significantly, instructions for these will be explicitly stated where appropriate.

Alternatively, check the *Red Hat Enterprise Linux Deployment Guide* available from the [Red Hat Documentation website](http://redhat.com/docs)¹ for information on the `/etc/sysconfig/` directory.

Editing the `/etc/rc.d/rc.local` file

Use this option only as a last resort!

1. Adjust the command as per the [Editing the `/etc/sysctl.conf` file](#) instructions.
2. Insert the new entry into the `/etc/rc.d/rc.local` file with the required parameter

2.3. Interrupt and Process Binding

Realtime environments need to minimize or eliminate latency when responding to various events. Ideally, interrupts (IRQs) and user processes can be isolated from one another on different dedicated CPUs.

Interrupts are generally shared evenly between CPUs. This can delay interrupt processing through having to write new data and instruction caches, and often creates conflicts with other processing occurring on the CPU. In order to overcome this problem, time-critical interrupts and processes can be dedicated to a CPU (or a range of CPUs). In this way, the code and data structures needed to process this interrupt will have the highest possible likelihood to be in the processor data and instruction caches. The dedicated process can then run as quickly as possible, while all other non-time-critical processes run on the remainder of the CPUs. This can be particularly important in cases where the speeds involved are in the limits of memory and peripheral bus bandwidth available. Here, any wait for memory to be fetched into processor caches will have a noticeable impact in overall processing time and determinism.

¹ <http://redhat.com/docs>

In practice we have found that optimal performance is entirely application specific. For example, in tuning applications for different companies which perform similar functions, the optimal performance tunings were completely different. For one firm, isolating 2 out of 4 CPUs for operating system functions and interrupt handling and dedicating the remaining 2 CPUs purely for application handling was optimal. For another firm, binding the network related application processes onto a CPU which was handling the network device driver interrupt yielded optimal determinism. Ultimately, tuning is often accomplished by trying a variety of settings to discover what works best for your organization.



Important

For many of the processes described here, you will need to know the CPU mask for a given CPU or range of CPUs. The CPU mask is typically represented as a 32-bit bitmask (on 32-bit machines), but can also be expressed as a decimal or hexadecimal number. For example: The CPU mask for CPU 0 only is 00000000000000000000000000000001 as a bitmask, 1 as a decimal, and 0x00000001 as a hexadecimal. The CPU mask for both CPU 0 and 1 is 00000000000000000000000000000011 as a bitmask, 3 as a decimal, and 0x00000003 as a hexadecimal.

Disabling the **irqbalance** daemon

This daemon is enabled by default and periodically forces interrupts to be handled by CPUs in an even, fair manner. However in realtime deployments, applications are typically dedicated and bound to specific CPUs, so the **irqbalance** daemon is not required.

1. Check the status of the **irqbalance** daemon

```
# service irqbalance status
irqbalance (pid PID) is running...
```

2. If the **irqbalance** daemon is running, stop it using the **service** command.

```
# service irqbalance stop
Stopping irqbalance:          [ OK ]
```

3. Use **chkconfig** to ensure that **irqbalance** does not restart on boot.

```
# chkconfig irqbalance off
```

Partially Disabling the **irqbalance** daemon

An alternative approach is to disable **irqbalance** only on those CPUs that have dedicated functions, and enable it on all other CPUs. This can be done by editing the **/etc/sysconfig/irqbalance** file.

1. Open **/etc/sysconfig/irqbalance** in your preferred text editor and find the section of the file titled **FOLLOW_ISOLCPUS**.

```
...[output truncated]...
# FOLLOW_ISOLCPUS
#     Boolean value.  When set to yes, any setting of
#     IRQ_AFFINITY_MASK above
#     is overridden, and instead computed to be the same mask that is
#     defined
#     by the isolcpu kernel command line option.
#
#FOLLOW_ISOLCPUS=no
```

2. Enable FOLLOW_ISOLCPUS by removing the # character from the beginning of the line and changing the value to yes.

```
...[output truncated]...
# FOLLOW_ISOLCPUS
#     Boolean value.  When set to yes, any setting of
#     IRQ_AFFINITY_MASK above
#     is overridden, and instead computed to be the same mask that is
#     defined
#     by the isolcpu kernel command line option.
#
FOLLOW_ISOLCPUS=yes
```

3. This will make **irqbalance** operate only on the CPUs not specifically isolated. This is most effective for machines with more than two processors, but works just as well on a dual-core machine.

Manually Assigning CPU Affinity to Individual IRQs

1. You can see which IRQ your devices are on by viewing the **/proc/interrupts** file.

```
# cat /proc/interrupts
```

This file contains a list of IRQs. Each line shows the IRQ number, the number of interrupts that happened in each CPU, followed by the IRQ type and a description.

```
CPU0          CPU1
0:   26575949      11      IO-APIC-edge  timer
1:         14         7      IO-APIC-edge  i8042
...[output truncated]...
```

2. To instruct an IRQ to run on only one processor, **echo** the CPU mask (as a decimal number) to **/proc/interrupts**. In this example, we are instructing the interrupt with IRQ number 142 to run on CPU 0 only.

```
# echo 1 > /proc/irq/142/smp_affinity
```

3. This change will only take effect once an interrupt has occurred. To test the settings, generate some disk activity, then check the `/proc/interrupts` file for changes. Assuming that you have caused an interrupt to occur, you should see that the number of interrupts on the chosen CPU have risen, while the numbers on the other CPUs have not changed.

Binding Processes to CPUs using the **taskset** utility

The **taskset** utility uses the process ID (PID) of a task to view or set the affinity, or can be used to launch a command with a chosen CPU affinity. In order to set the affinity, **taskset** requires the CPU mask expressed as either a decimal or hexadecimal number.

1. To set the affinity of a process that is not currently running, use **taskset** and specify the CPU mask and the process. In this example, **my_embedded_process** is being instructed to use only CPU 4 (using the decimal version of the CPU mask).

```
# taskset 8 /usr/local/bin/my_embedded_process
```

2. It is also possible to set the CPU affinity for processes that are already running by using the **-p (- -pid)** option with the CPU mask and the PID of the process you wish to change. In this example, the process with a PID of 7013 is being instructed to run only on CPU 0.

```
# taskset -p 1 7013
```



Note

The **taskset** utility will only work if Non-Uniform Memory Access (NUMA) is not enabled on the system. See [Section 3.6, “Non-Uniform Memory Access”](#) for more information on this.

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `chrt(1)`
- `taskset(1)`
- `nice(1)`
- `renice(1)`
- `sched_setscheduler(2)`

For a description of the Linux scheduling scheme

2.4. Filesystem determinism tips

Journal activity can introduce latency through ordering changes and committing data and metadata. Often, journaling filesystems can do things in such a way that they slow the system down.

The most common filesystem for use on Linux machines is the third extended filesystem, or ext3, which is a journaling filesystem. Its predecessor - ext2 - is a non-journaling filesystem that is almost completely compatible with ext3. Unless your organization specifically requires journaling, consider using ext2. In many of our best benchmark results, we utilize the ext2 filesystem and consider it one of the top initial tuning recommendations.

If using ext2 is not a suitable solution for your system, consider disabling **atime** under ext3 instead. There are very few real-world situations where **atime** is necessary, however it is enabled by default in Red Hat Enterprise Linux for longstanding legacy reasons. By disabling **atime** journal activity is avoided where it is not necessary. It can also help with reducing power consumption as the disk is not required to do as many needless writes, giving the disk more opportunities to enter a low-power state.

Disabling **atime**

1. Open the **/etc/fstab** file using your chosen text editor and locate the entry for the root mount point.

```
LABEL=/          /          ext3    defaults          1 1
...[output truncated]...
```

2. Edit the options sections to include the terms **noatime** and **nodiratime**. **noatime** prevents access timestamps being updated when a file is read and **nodiratime** will stop directory inode access times being updated.

```
LABEL=/          /          ext3    noatime,nodiratime  1 1
```

3. The **tmpwatch** file on Red Hat Enterprise Linux is set by default to clean files in **/tmp** based on their **atime**. If this is the case on your system, then the instructions above will result in users' **/tmp/*** files being emptied every day. This can be resolved by starting **tmpwatch** with the **--mtime** option.

```
--- /etc/cron.daily/tmpwatch.orig +++ /etc/cron.daily/tmpwatch @@ -3,6
+3,6 @@
/usr/sbin/tmpwatch 720 /var/tmp
for d in /var/{cache/man,catman}/{cat?,X11R6/cat?,local/cat?}; do
  if [ -d "$d" ]; then
- /usr/sbin/tmpwatch -f 720 "$d" + /usr/sbin/tmpwatch --mtime -f 720
  "$d"
  fi
```

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `bdflush(2)`
- `mkfs.ext2(8)`
- `mkfs.ext3(8)`
- `mount(8)` - for information on **atime**, **nodiratime** and **noatime**
- `chattr(1)`

2.5. gettimeofday speedup

Many application workloads (especially databases and financial service applications) perform extremely frequent **gettimeofday** or similar time function calls. Optimizing the efficiency of this calls can provide major benefits.

A Virtual Dynamic Shared Object (VDSO), is a shared library that allows application in user space to perform some kernel actions without as much overhead as a system call. The VDSO is often used to provide fast access to the **gettimeofday** system call data.

Enabling the VDSO instructs the kernel to use it's definition of the symbols in the VDSO, rather than the ones found in any user-space shared libraries, particularly the **glibc**. The effects of enabling the VDSO are system-wide - either all processes use it or none do.

When enabled, the VDSO overrides the **glibc** definition of **gettimeofday** with it's own. This removes the overhead of a system call, as the call is made direct to the kernel memory, rather than going through the **glibc**.

The VDSO boot parameter has three possible values:

0

Provides the most accurate time intervals at μs (microsecond) resolution, but also produces the highest call overhead, as it uses a regular system call

1

Slightly less accurate, although still at μs resolution, with a lower call overhead

2

The least accurate, with time intervals at the ms (millisecond) level, but offers the lowest call overhead

Enable **gettimeofday** with VDSO

There is a Virtual Dynamic Shared Object (VDSO) implemented in the **glibc** runtime library. The VDSO maps some of the kernel code, which is necessary to read **gettimeofday** in the user-space. Standard Red Hat Enterprise Linux 5.2 allows the **gettimeofday** function to be performed entirely in user-space, removing the system call overhead.

1. VDSO behavior is enabled by default. The value used to enable the VDSO affects the behavior of **gettimeofday**. It can be enabled by echoing the desired value to `/proc/sys/kernel/syscall64`.

```
# echo 1 > /proc/sys/kernel/syscall64
```

2. In addition to the above, the MRG Realtime kernel includes a further **gettimeofday** performance optimization. See [Section 3.2, “MRG Realtime Specific **gettimeofday** speedup”](#).



Important

Currently the **gettimeofday** speed up is implemented only for 64 bit architectures (AMD64 and Intel 64) and is not available on x86 machines.

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `gettimeofday(2)`

2.6. Don't run extra stuff

This is a common tool for improving performance, yet one that is often overlooked. Some 'extra stuff' to look for:

- Graphical desktop

Do not run graphics there they are not absolutely required, especially on servers. To avoid running the desktop software, open the `/etc/inittab` file with your preferred text editor and locate the following line:

```
id:5:initdefault:
...[output truncated]...
```

This setting changes the runlevel that the machine automatically boots into. By default, the runlevel is **5** - full multi-user mode, using the graphical interface. By changing the number in the string to **3**, the default runlevel will be full multi-user mode, but without the graphical interface.

```
id:3:initdefault:
...[output truncated]...
```

- Sendmail

Unless you are actively using Sendmail on the system you are tuning, disable it. If it is required, ensure it is well tuned or consider moving it to a dedicated machine.

- Remote Procedure Calls (RPCs)
- Network File System (NFS)

- Mouse Services

If you are not using Gnome, then you probably won't need a mouse either. Remove the hardware and uninstall **gpm**.

Remember to also check your third party applications, and any components added by external hardware vendors.

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `rpc(3)`
- `nfs(5)`
- `gpm(8)`

2.7. Swapping and Out Of Memory Tips

Memory Swapping

Swapping pages out to disk can introduce latency in any environment. To ensure low latency, the best strategy is to have enough memory in your systems so that swapping is not necessary. Always size the physical RAM as appropriate for your application and system. Use **vmstat** to monitor memory usage and watch the **si** (swap in) and **so** (swap out) fields. They should remain on zero as much as possible.

Out of Memory (OOM)

Out of Memory (OOM) refers to a computing state where all available memory, including swap space, has been allocated. Normally this will cause the system to panic and stop functioning as expected. There is a switch that controls OOM behavior in `/proc/sys/vm/panic_on_oom`. When set to **1** the kernel will panic on OOM. A setting of **0** instructs the kernel to call a function named **oom_killer** on an OOM. Usually, **oom_killer** can kill rogue processes and the system will survive.

1. The easiest way to change this is to **echo** the new value to `/proc/sys/vm/panic_on_oom`.

```
# cat /proc/sys/vm/panic_on_oom
1

# echo 0 > /proc/sys/vm/panic_on_oom

# cat /proc/sys/vm/panic_on_oom
0
```

2. It is also possible to prioritize which processes get killed by adjusting the **oom_killer** score. In `/proc/PID/` there are two tools labelled **oom_adj** and **oom_score**. Valid scores for **oom_adj** are in the range -16 to +15. This value is used to calculate the 'badness' of the process using an algorithm that also takes into account how long the process has been running, amongst

other factors. To see the current **oom_killer** score, view the **oom_score** for the process. **oom_killer** will kill processes with the highest scores first.

This example adjusts the **oom_score** of a process with a PID of 12465 to make it less likely that **oom_killer** will kill it.

```
# cat /proc/12465/oom_score
79872

# echo -5 > /proc/12465/oom_adj

# cat /proc/12465/oom_score
78
```

3. There is also a special value of -17, which disables **oom_killer** for that process. In the example below, **oom_score** returns a value of 0, indicating that this process would not be killed.

```
# cat /proc/12465/oom_score
78

# echo -17 > /proc/12465/oom_adj

# cat /proc/12465/oom_score
0
```

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- swapon(2)
- swapon(8)
- vmstat(8)

2.8. Network determinism tips

Transmission Control Protocol (TCP)

TCP can have a large effect on latency. TCP adds latency in order to obtain efficiency, control congestion, and to ensure reliable delivery. When tuning, consider the following points:

- Do you need ordered delivery?
- Do you need to guard against packet loss?

Packet loss is not always bad. Transmitting the packet again can cause greater delays.

- If you must use TCP, consider disabling the Nagle buffering algorithm by using **TCP_NODELAY** on your socket. The Nagle algorithm collects small outgoing packets to send all at once, and can have a detrimental effect on latency.

Network Tuning

There are numerous tools for tuning the network. Here are some of the more useful:

Interrupt Coalescing

To reduce network traffic, packets can be collected and a single interrupt generated.

Use the **-C (--coalesce)** option with the **ethtool** command to enable.

Congestion

Often, I/O switches can be subject to back-pressure, where network data builds up as a result of full buffers.

Use the **-A (--pause)** option with the **ethtool** command to change pause parameters and avoid network congestion.

Infiniband (IB)

Infiniband is a type of communications architecture often used to increase bandwidth and provide quality of service and failover. It can also be used to improve latency through Remote Direct Memory Access (RDMA) capabilities.

Network Protocol Statistics

Use the **-S (--statistics)** option with the **netstat** command to monitor network traffic.

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `ethtool(8)`
- `netstat(8)`

2.9. syslog tuning tips

syslog forwards log messages from any number of programs over a network. The less often this occurs, the larger the pending transaction is likely to be. If the transaction is very large an I/O spike can occur. To prevent this, keep the interval reasonably small.

Using **syslogd** for system logging.

The system logging daemon, called **syslogd**, is used to collect messages from a number of different programs. It also collects information reported by the kernel from the kernel logging daemon **klogd**. Typically, **syslogd** will log to a local file, but it can also be configured to log over a network to a remote logging server.

1. To enable remote logging, you will first need to configure the machine that will receive the logs. **syslogd** uses configuration settings defined in the `/etc/sysconfig/syslog` and `/etc/syslog.conf` files. To instruct **syslogd** to receive logs from remote machines, open `/etc/sysconfig/syslog` in your preferred text editor and locate the **SYSLOGD_OPTIONS=** line.

```
# Options to syslogd
# -m 0 disables 'MARK' messages.
# -r enables logging from remote machines
# -x disables DNS lookups on messages recieved with -r
# See syslogd(8) for more details

SYSLOGD_OPTIONS="-m 0"

...[output truncated]...
```

2. Append the `-r` parameter to the options line:

```
SYSLOGD_OPTIONS="-m 0 -r"
```

3. Once remote logging support is enabled on the remote logging server, each system that will send logs to it must be configured to send its syslog output to the server, rather than writing those logs to the local filesystem. To do this, edit the `/etc/syslog.conf` file on each client system. For each of the various logging rules defined in that file, you can replace the local log file with the address of the remote logging server.

```
# Log all kernel messages to remote logging host.
kern.*      @my.remote.logging.server
```

The example above will cause the client system to log all kernel messages to the remote machine at `@my.remote.logging.server`.

4. It is also possible to configure **syslogd** to log all locally generated system messages, by adding a wildcard line to the `/etc/syslog.conf` file:

```
# Log all messages to a remote logging server:
*.*        @my.remote.logging.server
```



Important

Note that **syslogd** does not include built-in rate limiting on its generated network traffic. Therefore, we recommend that remote logging on MRG Realtime systems be confined to only those messages that are required to be remotely logged by your organization. For example, kernel warnings, authentication requests, and the like. Other messages should be locally logged instead.

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- syslog(3)
- syslog.conf(5)
- syslogd(8)

2.10. The PC Card Daemon

The **pcscd** daemon is used to manage connections to PC and SC smart card readers. Although **pcscd** is usually a low priority task, it can often use more CPU than any other daemon. This additional background noise can lead to higher pre-emption costs to realtime tasks and other undesirable impacts on determinism.

Disabling the **pcscd** Daemon

1. Check the status of the **pcscd** daemon

```
# service pcscd status
pcscd (pid PID) is running...
```

2. If the **pcscd** daemon is running, stop it using the **service** command.

```
# service pcscd stop
Stopping PC/SC smart card daemon (pcscd):          [ OK ]
```

3. Use **chkconfig** to ensure that **pcscd** does not restart on boot.

```
# chkconfig pcscd off
```

2.11. Reduce TCP performance spikes

In order to reduce a performance spike with relation to timestamps generation, change the values of the TCP related entries with the **sysctl** command.

1. Use **sysctl** to change the value of **net.ipv4.neigh.default.unres_qlen** from 3 to 100. The **-w** option will make the change persistent between reboots:

```
# /sbin/sysctl -w net.ipv4.neigh.default.unres_qlen=100
```

2. Make the same change for **net.ipv4.neigh.eth0.unres_qlen**:

```
# /sbin/sysctl -w net.ipv4.neigh.eth0.unres_qlen=100
```

2.12. Reducing the TCP delayed ack timeout

Some applications that send small network packets can experience latencies due to the TCP delayed acknowledgement timeout. This value defaults to 40ms. To avoid this problem, try reducing the **tcp_delack_min** timeout value. This changes the minimum time to delay before sending an acknowledgement systemwide.

1. Write the desired minimum value, in microseconds, to **/proc/sys/net/ipv4/tcp_delack_min**:

```
# echo 1 > /proc/sys/net/ipv4/tcp_delack_min
```

Realtime-Specific Tuning

Once you have completed the tunes in [Chapter 2, General System Tuning](#) you are ready to start MRG Realtime specific tuning. You must have the MRG Realtime kernel installed for these procedures.



Important

Do not attempt to use the tools in this section without first having completed [Chapter 2, General System Tuning](#). You will not see a performance improvement.

When are you ready to begin MRG Realtime tuning, perform these steps first, as they will provide the greatest benefit:

- [Section 3.1, “Setting Scheduler Priorities”](#)

When you are ready to start some fine-tuning on your system, then try the other sections in this chapter:

- [Section 3.2, “MRG Realtime Specific `gettimeofday` speedup”](#)
- [Section 3.3, “Using `kdump` and `kexec` with the MRG Realtime kernel”](#)
- [Section 3.4, “TSC timer synchronization on Opteron CPUs”](#)
- [Section 3.5, “Infiniband”](#)
- [Section 3.6, “Non-Uniform Memory Access”](#)

This chapter also includes information on two performance monitoring tools:

- [Section 3.7, “Using the `ftrace` Utility for Tracing Latencies”](#)
- [Section 3.8, “Latency Tracing Using `trace-cmd`”](#)

[Section 3.9, “Using `sched_nr_migrate` to limit `SCHED_OTHER` processes.”](#)

When you have completed all the tuning suggestions in this chapter, move on to [Chapter 4, Application Tuning and Deployment](#)

3.1. Setting Scheduler Priorities

The MRG Realtime kernel allows fine grained control of scheduler priorities. It also allows application level programs to be scheduled at a higher priority than kernel threads. This can be useful but may also carry consequences. It is possible that it will cause the system to hang and other unpredictable behavior if crucial kernel processes are prevented from running as needed. Ultimately the correct settings are workload dependent.

Priorities are defined in groups, with some groups dedicated to certain kernel functions:

Priority	Threads	Description
1	Low priority kernel threads	Priority 1 is usually reserved for those tasks that just want to be above SCHED_OTHER
2 - 69	Available for use	Range used for typical application priorities

Priority	Threads	Description
70 - 79	Soft IRQs	
80	NFS	RPC, Locking and Authentication threads for NFS
81 - 89	Hard IRQs	Dedicated interrupt processing threads for each IRQ in the system
90 - 98	Available for use	For use <i>only</i> by very high priority application threads
99	Watchdogs and migration	System threads that must run at the highest priority

Table 3.1. Priority Map

Using **rtctl** to Set Priorities

1. Priorities are set using a series of levels, ranging from 0 (lowest priority) to 99 (highest priority). The system startup script **rtctl** initializes the default priorities of the kernel threads. By requesting the status of the **rtctl** service, you can view the priorities of the various kernel threads.

```
# service rtctl status
2  TS      - [kthreadd]
3  FF      99 [migration/0]
4  FF      99 [posix_cpu_timer]
5  FF      50 [softirq-high/0]
6  FF      50 [softirq-timer/0]
7  FF      90 [softirq-net-tx/]
...[output truncated]...
```

2. The output is in the format:

```
[PID] [scheduler policy] [priority] [process name]
```

In the **scheduler policy** field, a value of **TS** indicates a policy of *normal* and **FF** indicates a policy of *FIFO* (first in, first out).

3. The **rtctl** system startup script relies on the **/etc/rtgroups** file. To make changes, open the **/etc/rtgroups** file in your preferred text editor.

```
kthreads:o:0:[.*\]$
watchdog:f:99:[watchdog.*\]
migration:f:99:[migration\/.*\]
posix_cpu_timer:f:99:[posix_cpu_timer\]
hardirq:f:95:[IRQ-.*\]
...[output truncated]...
```


4. Each line represents a process. You can change the priority of the process by adjusting the parameters. The entries in this file are in the format:

```
[group name]:[scheduler policy]:[scheduler priority]:[regular expression]
```

In the **scheduler policy** field, the following values are accepted:

o	Sets a policy of <i>other</i> . If the policy is set to o , the scheduler priority field will be set to 0 and ignored.
b	Sets a policy of <i>batch</i> .
f	Sets a policy of <i>FIFO</i> .
*	If the policy is set to * , no change will be made to any matched thread policy.

The **regular expression** field matches the thread name to be modified.

5. After editing the file, you will need to restart the **rtctl** service to reload it with the new settings:

```
# service rtctl stop

# service rtctl start
Setting kernel thread priorities: done
```

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `rtctl(1)`
- `rtgroups(5)`

3.2. MRG Realtime Specific **gettimeofday** speedup

In addition to the **gettimeofday(2)** speedup listed in [Section 2.5, “**gettimeofday** speedup”](#), the MRG Realtime kernel contains a further **gettimeofday** performance optimization. This method caches the most recently used time value in a global system file. If another **gettimeofday** call is performed within the ms (hz) then it is not necessary to re-read the hardware clock. As a result, applications which do not require microsecond precision benefit.

Enable the MRG Realtime **gettimeofday** speedup

This setting is not enabled by default. When you start it, it needs to be enabled on a global basis.

1. Firstly you will need to append the line **kernel.vsyscall64 = 2** to the `/etc/sysctl.conf` file. This causes the **gettimeofday** function to be performed entirely in user-space, removing the system call overhead.

```
echo "kernel.syscall164 = 2" >> /etc/sysctl.conf
```

2. To make the change effective immediately and persistent between reboots, use the **-w** option with the **sysctl** command to update the setting:

```
/sbin/sysctl -w kernel.syscall164=2
```

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `sysctl(8)`
- `gettimeofday(2)`

3.3. Using **kdump** and **kexec** with the MRG Realtime kernel

If **kdump** is enabled on your system, the standard boot kernel will reserve a small section of system RAM and load the **kdump** kernel into the reserved space. When a kernel panic or other fatal error occurs, **kexec** is used to boot into the **kdump** kernel. **Kexec** is a fastboot mechanism that allows the **kdump** kernel to boot without going through BIOS. The **kdump** kernel boots up using only the reserved RAM and sends an error message to the console. It will then write a dump of the boot kernel's address space to a file for later debugging. Because **kexec** does not go through the BIOS, the memory of the original boot is retained, and the crash dump is much more detailed. Once this is done, the **kdump** kernel performs a reboot, which will reset the machine and bring the boot kernel back up.



Important

In Red Hat Enterprise Linux 5.2 there is no dedicated **kdump** kernel. It uses the main kernel instead. The MRG Realtime kernel cannot be used as a **kdump** kernel, but it supports the use of a separate **kdump** kernel. It is recommended that you use the MRG Realtime kernel as the boot kernel, and the Red Hat Enterprise Linux 5.2 kernel as the **kdump** kernel.

There are three methods for enabling **kdump** under Red Hat Enterprise Linux 5.2. The first method uses a tool called **rt-setup-kdump**. The second adds a command line to the boot kernel, and the third uses the graphical system configuration tool included with Red Hat Enterprise Linux 5.2.

Creating a basic **kdump** kernel with **rt-setup-kdump**

1. The **rt-setup-kdump** tool is part of the **rt-setup** package, which can be installed using **yum**:

```
# yum install rt-setup
```

2. Run the tool by invoking it at the shell prompt as the root user. This will set the Red Hat Enterprise Linux 5.2.1 or Red Hat Enterprise Linux 5.2.2 kernel to be the kdump kernel:

```
# rt-setup-kdump
```

Enabling **kdump** with **grub.conf**

1. Firstly, you will need to check that you have the **kexec-tools** package installed.

```
# rpm -q kexec-tools
kexec-tools-1.101-194.4.el5
```

2. By default, the crash dump is saved in the **/var/crash** file. If you wish to change this, simply uncomment and adjust the **path** value in the **/etc/kdump.conf** file. This can be a local file, or on another server.

```
...[output truncated]...
#raw /dev/sda5
#ext3 /dev/sda3
#ext3 LABEL=/boot
#ext3 UUID=03138356-5e61-4ab3-b58e-27507ac41937
#net my.server.com:/export/tmp
#net user@my.server.com

path /path/to/file

#core_collector makedumpfile -c
#link_delay 60
#kdump_post /var/crash/scripts/kdump-post.sh
#extra_bins /usr/bin/lftp
#extra_modules gfs2
#default shell
```

3. Open the **/etc/grub.conf** file in your preferred text editor and add a **crashkernel** line to the boot kernel. This line takes the form:

```
crashkernel=[MB of RAM to reserve]M@[memory location]M
```

A typical **crashkernel** line would reserve 128 megabytes (128M) at 16 megabytes (16M), which is equivalent to the address 0x1000000:

```
crashkernel=128M@16M
```

A typical MRG Realtime `/etc/grub.conf` file would have the MRG Realtime kernel as the boot kernel, and the **crashkernel** line added to the Red Hat Enterprise Linux kernel:

```
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title Red Hat Enterprise Linux (realtime) (2.6.24.7-81.el5rt)
    root (hd0,0)
    kernel /vmlinuz-2.6.24.7-81.el5rt ro root=/dev/HelpdeskRHEL5/Root rhgb
    quiet
    initrd /initrd-2.6.24.7-81.el5rt.img
title Red Hat Enterprise Linux Client (2.6.24.7-81.el5)
    root (hd0,0)
    kernel /vmlinuz-2.6.24.7-81.el5 ro root=/dev/HelpdeskRHEL5/Root rhgb
    quiet crashkernel=128M@16M
    initrd /initrd-2.6.24.7-81.el5.img
```

4. Once you have saved your changes, restart the system to set up the reserved memory space. You can then turn on the **kdump** init script and start the **kdump** service:

```
# chkconfig kdump on

# service kdump status
Kdump is not operational

# service kdump start
Starting kdump:                [ OK ]
```

5. If you want to check that the **kdump** is working correctly, you can simulate a panic using **sysrq**:

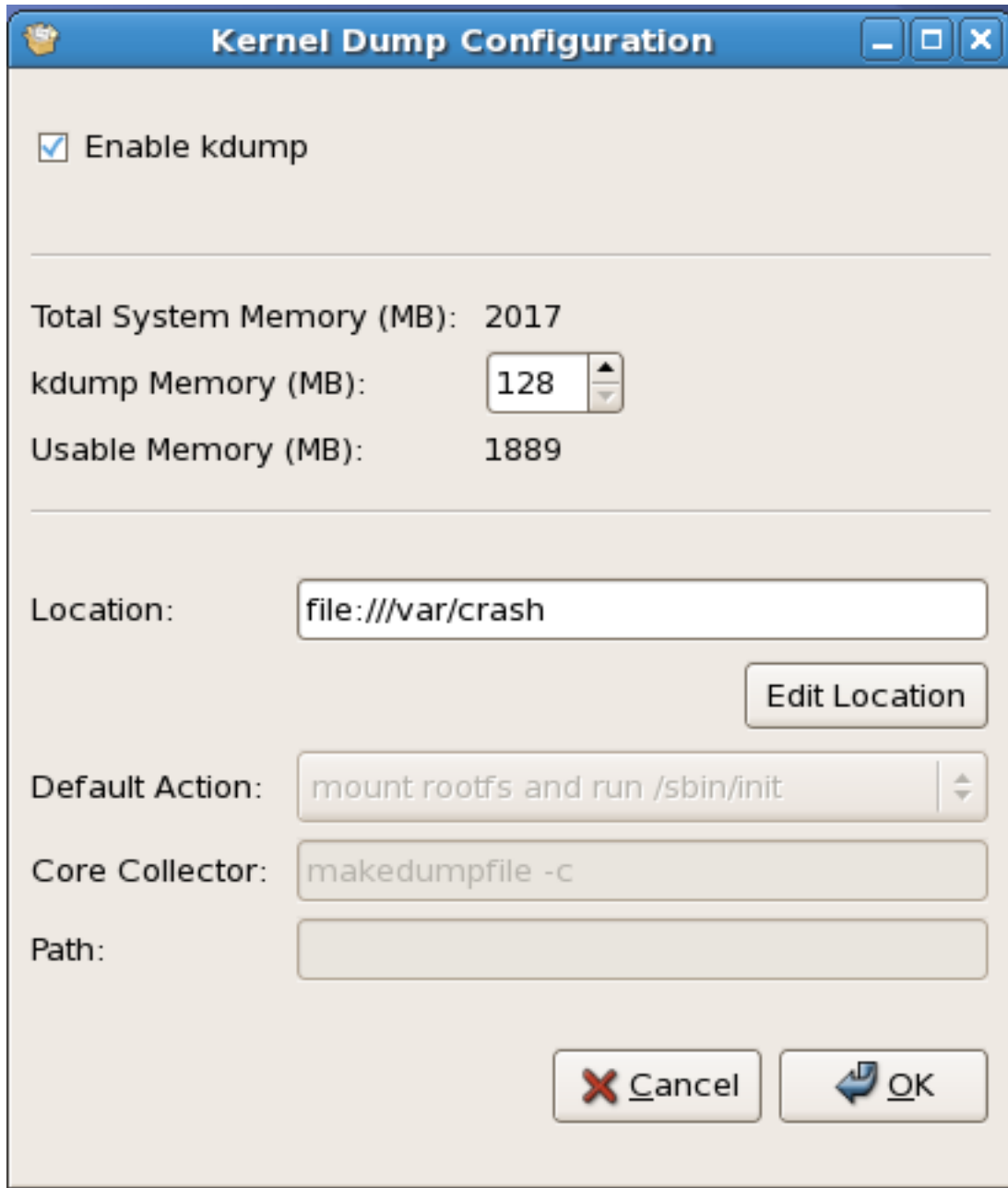
```
# echo "c" > /proc/sysrq-trigger
```

This will cause the kernel to panic and the system will boot into the **kdump** kernel. Once your system has been brought back up with the boot kernel, you should be able to check the log file at the location you specified.

Enabling **kdump** with **system-config-kdump**

1. Select the **Kdump** system tool from the **System|Administration** menu, or use the following command from the command line:

```
# system-config-kdump
```



The image shows a 'Kernel Dump Configuration' dialog box. It has a title bar with a yellow star icon and standard window controls. The main area contains several settings: a checked checkbox for 'Enable kdump', a section showing memory statistics (Total System Memory: 2017 MB, kdump Memory: 128 MB, Usable Memory: 1889 MB), a 'Location' field set to 'file:///var/crash' with an 'Edit Location' button, a 'Default Action' dropdown set to 'mount rootfs and run /sbin/init', a 'Core Collector' field set to 'makedumpfile -c', and an empty 'Path' field. At the bottom are 'Cancel' and 'OK' buttons.

Kernel Dump Configuration

☒ Enable kdump

Total System Memory (MB): 2017
kdump Memory (MB): 128
Usable Memory (MB): 1889

Location: file:///var/crash Edit Location

Default Action: mount rootfs and run /sbin/init

Core Collector: makedumpfile -c

Path:

Cancel OK

2. Select the check box labeled **Enable kdump** and adjust the necessary settings for memory reservation and dump file location. Click **OK** to save your changes.



Important

Always check the `/etc/grub.conf` file to ensure that the tool has adjusted the correct kernel. The MRG Realtime kernel should be the default boot kernel and the Red Hat Enterprise Linux kernel should be used as the crash kernel.

3. If you want to check that the **kdump** is working correctly, you can simulate a panic using **sysrq**:

```
# echo "c" > /proc/sysrq-trigger
```

This will cause the kernel to panic and the system will boot into the **kdump** kernel. Once your system has been brought back up with the boot kernel, you should be able to check the log file at the location you specified.



Note

Some hardware needs to be reset during the configuration of the **kdump** kernel. If you have any problems getting the **kdump** kernel to work, edit the `/etc/sysconfig/kdump` file and add `reset_devices=1` to the `KDUMP_COMMANDLINE_APPEND` variable.



Important

On IBM LS21 machines, the following warning message may occur when attempting to boot the **kdump** kernel:

```
irq 9: nobody cared (try booting with the "irqpoll" option)
handlers:
[<ffffffff811660a0>] (acpi_irq+0x0/0x1b)
turning off IO-APIC fast mode.
```

Some systems will recover from this error and continue booting, while some will freeze after displaying the message. This is a known issue. If you see this error, add the line `acpi=noirq` as a boot parameter to the **kdump** kernel. Only add this line if this error occurs as it can cause boot problems on machines not affected by this issue.

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `kexec(8)`
- `/etc/kdump.conf`

3.4. TSC timer synchronization on Opteron CPUs

The current generation of AMD64 Opteron processors are susceptible to a large **gettimeofday** skew when **cpufreq** is enabled while using the Time Stamp Counter (TSC). MRG Realtime provides a method to prevent this on Opteron systems by forcing all processors to simultaneously change to the same frequency. As a result, the TSC on a single processor never increments at a different rate than the TSC on another processor.

Enabling TSC timer synchronization

1. Open the `/etc/grub.conf` file in your preferred text editor and add the line **nohpet nopmtimer powernow-k8.tscsync=1** to the MRG Realtime kernel. This forces the use of TSC and enables simultaneous core processor frequency transitions.

```
...[output truncated]...
title Red Hat Enterprise Linux (realtime) (2.6.24.7-81.el5rt)
    root (hd0,0)
    kernel /vmlinuz-2.6.24.7-81.el5rt ro root=/dev/HelpdeskRHEL5/Root rhgb
    quiet nohpet nopmtimer powernow-k8.tscsync=1
    initrd /initrd-2.6.24.7-81.el5rt.img
```

2. You will need to restart your system for the changes to take effect.

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `gettimeofday(2)`

3.5. Infiniband

Infiniband is a type of communications architecture often used to increase bandwidth and provide quality of service and failover. It can also be used to improve latency through Remote Direct Memory Access (RDMA) capabilities.

Support for Infiniband under MRG Realtime does not differ from the support offered under Red Hat Enterprise Linux 5.2

3.6. Non-Uniform Memory Access

Non-Uniform Memory Access (NUMA) is a design used to allocate memory resources to a specific CPU. This can improve access time and results in fewer memory locks. Although this appears as though it would be useful for reducing latency, NUMA systems have been known to interact badly with realtime applications, as they can cause unexpected event latencies.

As mentioned in [Binding Processes to CPUs using the `taskset` utility](#) the `taskset` utility will only work if NUMA is not enabled on the system. If you want to perform process binding in conjunction with NUMA, use the `numactl` command instead of `taskset`.

For more information about the NUMA API, see Andi Kleen's whitepaper [An NUMA API for Linux](#)¹.

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `numactl(8)`

¹ <http://www.halobates.de/numaapi3.pdf>

3.7. Using the **ftrace** Utility for Tracing Latencies

One of the diagnostic facilities provided with the MRG Realtime kernel is **ftrace**, which is used by developers to analyze and debug latency and performance issues that occur outside of user-space. The **ftrace** utility has a variety of options that allow you to use the utility in a number of different ways. It can be used to trace context switches, measure the time it takes for a high-priority task to run, the length of time interrupts are disabled for, or list all the kernel functions executed during a given period.

Some tracers, such as the function tracer, will produce exceedingly large amounts of data, which can turn trace log analysis into a time-consuming task. However, it is possible to instruct the tracer to begin and end only when the application reaches critical code paths.

The **ftrace** utility is not enabled in the production version of the MRG Realtime kernel as it creates additional overhead. If you wish to use the **ftrace** utility you will need to download and install the **trace** variant of the MRG Realtime kernel.



Note

For instructions on how to install kernel variants, see the *MRG Realtime Installation Guide*.

Using the **ftrace** Utility

1. Once you are using the **trace** variant of the MRG Realtime kernel, you can set up the **ftrace** utility. You will need to create a **/debug** directory and then mount it to use the **debugfs** file system.

```
# mkdir /debug  
  
# mount -t debugfs nodev /debug
```

2. In the **/debugfs/tracing/** directory there is a file named **available_tracers**. This file contains all the available tracers for the installed version of **ftrace**. To see the list of available tracers, use the **cat** command to view the contents of the file:

```
# cat /debugfs/tracing/available_tracers  
  
events wakeup preemptirqsoff preemptoff irqsoff ftrace sched_switch none
```

events

Traces system events such as timers, system calls, interrupts, context switches, process wake up and others

wakeup

Traces the maximum latency in between the highest priority process waking up and being scheduled. Only RT tasks are considered by this tracer (**SCHED_OTHER** tasks are ignored as of now).

preemptirqsoff

Traces the areas that disable pre-emption and interrupts and records the maximum amount of time for which pre-emption or interrupts were disabled.

preemptoff

Similar to the **preemptirqsoff** tracer but traces only the maximum interval for which pre-emption was disabled.

irqsoff

Similar to the **preemptoff** tracer but traces only the maximum interval for which interrupts were disabled.

ftrace

Records the kernel functions called during a tracing session. The **ftrace** utility can be run simultaneously with any of the other tracers.

sched_switch

Traces context switches between tasks.

none

Disables tracing.

- To manually start a tracing session, first select the tracer you wish to use from the list in **available_tracers** and then use the **echo** to insert the name of the tracer into **/debugfs/tracing/current_tracer**:

```
# echo events > /debugfs/tracing/current_tracer
```

**Important**

When using the **echo**, ensure you place a space character in between the value and the **>** character. At the shell prompt, using **0>**, **1>**, and **2>** (without a space character) refers to standard input, standard output and standard error. Using them by mistake could result in severe unintended consequences.

- To check if the **ftrace** utility is running, use the **cat** command to view the **/proc/sys/kernel/ftrace_enabled** file. A value of **1** indicates that **ftrace** is running, and **0** indicates that it is not running.

```
# cat /proc/sys/kernel/ftrace_enabled
1
```

By default, the tracer is enabled. To turn the tracer on or off, **echo** the appropriate value to the **/debug/tracing/tracing_enabled** file.

```
# echo 0 > /proc/sys/kernel/ftrace_enabled
```

```
# echo 1 > /proc/sys/kernel/ftrace_enabled
```

5. Adjust details and parameters of the tracers by changing the values for the various files in the **/debugfs/tracing/** directory. Some examples are:

Set the maximum latency to 0:

```
# echo 0 > /debugfs/tracing/tracing_max_latency
```

Define the threshold above which latencies should be considered. This is defined in microseconds:

```
# echo 200 > /debugfs/tracing/tracing_thresh
```

6. Start tracing:

```
# echo 1 > /debugfs/tracing/tracing_enabled
```

7. Run the test application. A reasonable test might be:

```
/bin/ls -l
```

8. Stop tracing:

```
# echo 0 > /debugfs/tracing/tracing_enabled
```

9. View the trace logs:

```
# cat /debugfs/tracing/trace
# cat /debugfs/tracing/latency_trace/
```

10. To store the trace logs, copy them to another file:

```
# cat /debugfs/tracing/latency_trace > /tmp/lat_trace_log
```

11. There are a number of options available for changing the format of the output. These options are stored in **/debug/tracing/iter_ctrl**:

--print-parent	Show the parent of the functions.
--sym-offset	Add the offset into the function.

--sym-addr	Add the address of a symbol.
--verbose	Increase the verbosity of the tracer output.

12. Use the **cat** command to view the current configuration:

```
# cat /debug/tracing/iter_ctrl
```

To set a single option on the tracer output configuration, **echo** the option name to the **/debug/tracing/iter_ctrl** file.

```
# echo verbose > /debug/tracing/iter_ctrl
```

To disable a single option on the tracing output configuration, **echo** the option name with the test **no** before it to the **/debug/tracing/iter_ctrl** file.

```
# echo noverbose > /debug/tracing/iter_ctrl
```



Note

If you use a single **>** with the **echo** command, it will override any existing value in the file. If you wish to append the value to the file, use **>>** instead.

13. The **ftrace** utility can be filtered by altering the settings in the **/debug/tracing/set_ftrace_filter** file. If no filters are specified in the file, all processes are traced. Use the **cat** to view the current filters:

```
# cat /debug/tracing/set_ftrace_filter
```

14. To change the filters, **echo** the name of the process to be traced. The filter allows the use of a ***** wildcard at the beginning or end of a search term. Some examples of filters are:

- Trace only the **schedule** process:

```
# echo schedule > /debug/tracing/set_ftrace_filter
```

- Trace all processes that end with **lock**:

```
# echo *lock > /debug/tracing/set_ftrace_filter
```

- Trace all processes that start with **spin_**:

```
# echo spin_* > /debug/tracing/set_ftrace_filter
```

- Trace all processes with **cpu** in the name:

```
# echo *cpu* > /debug/tracing/set_ftrace_filter
```



Note

The `*` wildcard for the tracer filter will only work at the beginning or end of a word. For example: `spin_*` and `*lock` will work, but `spin_*lock` will not.

3.8. Latency Tracing Using `trace-cmd`

trace-cmd is a MRG Realtime function that traces all kernel function calls, and some special events. It records what is happening in the system during a short period of time, providing information that can be used to analyze system behavior.

The **trace-cmd** tool is not enabled in the production version of the MRG Realtime kernel as it creates additional overhead. If you wish to use the **trace-cmd** tool you will need to download and install either the **trace** or **debug** variants of the MRG Realtime kernel.



Note

For instructions on how to install kernel variants, see the *MRG Realtime Installation Guide*.

1. Once you are using either the **trace** or **debug** variants of the MRG Realtime kernel, you can install the **trace-cmd** tool using **yum**.

```
# yum install trace-cmd
```

2. You will need to create a **/debugfs** directory and then mount it to use the **debugfs** file system.

```
# mkdir /debugfs
```

```
# mount -t debugfs debugfs /debugfs
```

3. You can choose to make the **debugfs** directory mount automatically on boot. You can do this by opening the **/etc/fstab** file in your preferred text editor, and adding the following line:

```
/debugfs    /debugfs    debugfs    defaults    0    0
```

4. To start the utility, type **trace-cmd** at the shell prompt, along with the options you require, using the following syntax:

```
# trace-cmd [-f] [command]
```

The use of the **-f** option sets Function Tracing and can be used with any other trace command.

The commands instruct **trace-cmd** to trace in specific ways.

Command	Trace Type	Description
-s	Context switch	Traces the context switches between tasks.
-i	Interrupts off	Records the maximum time that an interrupt is disabled. When a new maximum is recorded, it replaces the previous maximum.
-p	Pre-emption off	Records the maximum time that pre-emption is disabled. When a new maximum is recorded, it replaces the previous maximum.
-b	Pre-emption and interrupts off	Records the maximum time that pre-emption <i>or</i> interrupts are disabled. When a new maximum is recorded, it replaces the previous maximum.
-w	Wakeup	Traces and records the maximum time for the highest priority task to get scheduled after it has been woken up.
-e	Event tracing	
-f	Function tracing	Can be used with any other trace
-l	Prints log in the latency_trace format	Can be used with any other trace

5. In this example, the **trace-cmd** utility will:
- Select the **context switch** tracing method
 - Enable the latency tracer
 - Run the **ls -la** command
 - Turn the latency tracer off again

```
# trace-cmd -s /bin/ls -la > /tmp/latency_log.txt
```

3.9. Using `sched_nr_migrate` to limit `SCHED_OTHER` processes.

If a **SCHED_OTHER** task spawns a large number of other tasks, they will all run on the same CPU. The migration task or **softirq** will try to balance these tasks so they can run on idle CPUs. The **sched_nr_migrate** option can be set to specify the number of tasks that will move at a time. Because realtime tasks have a different way to migrate, they are not directly affected by this, however when **softirq** moves the tasks it locks the run queue spinlock that is needed to disable interrupts. If there are a large number of tasks that need to be moved, it will occur while interrupts are disabled, so no timer events or wakeups will happen simultaneously. This can cause severe latencies for realtime tasks when the **sched_nr_migrate** is set to a large value.

Adjusting the value of the `sched_nr_migrate` variable

1. Increasing the **sched_nr_migrate** variable gives high performance from **SCHED_OTHER** threads that spawn lots of tasks, at the expense of realtime latencies. For low realtime task latency at the expense of **SCHED_OTHER** task performance, then the value should be lowered. The default value is 8.
2. To adjust the value of the **sched_nr_migrate** variable, you can **echo** the value directly to **/proc/sys/kernel/sched_nr_migrate**:

```
# echo 2 > /proc/sys/kernel/sched_nr_migrate
```

3. As this is a kernel process, you can also use the **sysctl** command.

```
# sysctl kernel.sched_nr_migrate=2
```

Application Tuning and Deployment

This page contains tips related to enhancing and developing MRG Realtime Applications.



Note

In general, try to use *POSIX* (Portable Operating System Interface) defined APIs. The MRG Realtime developers are compliant with POSIX standards and latency reduction in the MRG Realtime kernel is also based on POSIX.

Further Reading

For further reading on developing your own MRG Realtime applications, start by reading the [RTWiki Article](#)¹.

4.1. Signal Processing in Realtime Applications

Traditional UNIX™ and POSIX signals have their uses, especially for error handling, but they are not suitable for use in realtime applications as an event delivery mechanism. The reason for this is that the current Linux kernel signal handling code is quite complex, due mainly to legacy behavior and the multitude of APIs that need to be supported. This complexity means that the code paths that may be taken when delivering a signal are not always the optimal path and quite long latencies may be experienced by applications.

The original motivation behind UNIX™ signals was to multiplex one thread of control (the process) between different "threads" of execution. Signals behave somewhat like operating system interrupts - when a thread is delivered to an application, the application's context is saved and it starts executing a previously registered signal handler. Once the signal handler has completed, the application returns to executing where it was when the signal was delivered. This can get complicated in practice.

Signals are too non-deterministic to trust them in a realtime application. A better option is to use POSIX Threads (pthreads) to distribute your workload and communicate between various components. You can coordinate groups of threads using the pthreads mechanisms of mutexes, condition variables and barriers and trust that the code paths through these relatively new constructs are much cleaner than the legacy handling code for signals.

Further Reading

For more information, or for further reading, the following links are related to the information given in this section.

- RTWiki's [Build an RT Application](#)²
- Ulrich Drepper's [Requirements of the POSIX Signal Model](#)³

¹ http://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application

4.2. Using `sched_yield` and Other Synchronization Mechanisms

The `sched_yield` system call is used by a thread allowing other threads a chance to run. Often when `sched_yield` is used, the thread can go to the end of the run queues, taking a long time to be scheduled again, or it can be rescheduled straight away, creating a busy loop on the CPU. The scheduler is better able to determine when and if there are actually other threads wanting to run. Avoid using `sched_yield` on any RT task.

POSIX Threads (Pthreads) have abstractions that will provide more consistent behavior across kernel versions. However, this can also mean that the system has less time to process networking packets, leading to considerable performance loss. This type of loss can be difficult to diagnose as there are no significant changes in the networking components of the system. It can also result in a change in behavior of some applications.

For more information, see Arnaldo Carvalho de Melo's paper on [Earthquaky kernel interfaces](http://vger.kernel.org/~acme/unbehaved.txt)⁴.

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `pthread.h(P)`
- `sched_yield(2)`
- `sched_yield(3p)`

4.3. Mutex options

Standard Mutex Creation

Mutual exclusion (mutex) algorithms are used to prevent processes simultaneously using a common resource. A fast user-space mutex (futex) is a tool that allows a user-space thread to claim a mutex without requiring a context switch to kernel space, provided the mutex is not already held by another thread.



Note

In this document, we use the terms *futex* and *mutex* to describe POSIX thread (pthread) mutex constructs.

1. When you initialize a `pthread_mutex_t` object with the standard attributes, it will create a private, non-recursive, non-robust and non priority inheritance capable mutex.
2. Under pthreads, mutexes can be initialized with the following strings:

```
pthread_mutex_t my_mutex;
```

⁴ <http://vger.kernel.org/~acme/unbehaved.txt>


```
pthread_mutex_init(&my_mutex, NULL);
```

3. In this case, your application may not be benefiting of the advantages provided by the pthreads API and the MRG Realtime kernel. There are a number of mutex options that should be considered when writing or porting an application.

Advanced Mutex Options

In order to define any additional capabilities for the mutex you will need to create a **pthread_mutexattr_t** object. This object will store the defined attributes for the mutex.



Important

For the sake of brevity, these examples do not include a check of the return value of the function. This is a basic safety procedure and one that you should always perform.

1. Creating the mutex object:

```
pthread_mutex_t my_mutex;

pthread_mutexattr_t my_mutex_attr;

pthread_mutexattr_init(&my_mutex_attr);
```

2. Shared and Private mutexes:

Shared mutexes can be used between processes, however they can create a lot more overhead.

```
pthread_mutexattr_setpshared(&my_mutex_attr, PTHREAD_PROCESS_SHARED);
```

3. Realtime priority inheritance:

Priority inversion problems can be avoided by using priority inheritance.

```
pthread_mutexattr_setprotocol(&my_mutex_attr, PTHREAD_PRIO_INHERIT);
```

4. Robust mutexes:

Robust mutexes are released when the owner dies, however this can also come at a high overhead cost. **_NP** in this string indicates that this option is non-POSIX or not portable.

```
pthread_mutexattr_setrobust_np(&my_mutex_attr, PTHREAD_MUTEX_ROBUST_NP);
```

5. Mutex initialization:

Once the attributes are set, initialize a mutex using those properties.

```
pthread_mutex_init(&my_mutex, &my_mutex_attr);
```

6. Cleaning up the attributes object:

After the mutex has been created, you can keep the attribute object in order to initialize more mutexes of the same type, or you can clean it up. The mutex is not affected in either case. To clean up the attribute object, use the **_destroy** command.

```
pthread_mutexattr_destroy(&my_mutex_attr);
```

The mutex will now operate as a regular **pthread_mutex**, and can be locked, unlocked and destroyed as normal.

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- `futex(7)`
- `pthread_mutex_destroy(P)`

For information on **pthread_mutex_t** and **pthread_mutex_init**

- `pthread_mutexattr_setprotocol(3p)`

For information on **pthread_mutexattr_setprotocol** and **pthread_mutexattr_getprotocol**

- `pthread_mutexattr_setprioceiling(3p)`

For information on **pthread_mutexattr_setprioceiling** and **pthread_mutexattr_getprioceiling**

4.4. TCP_NODELAY and Small Buffer Writes

As discussed briefly in [Transmission Control Protocol \(TCP\)](#), by default TCP uses Nagle's algorithm to collect small outgoing packets to send all at once. This can have a detrimental effect on latency.

Using TCP_NODELAY and TCP_CORK to improve network latency

1. Applications that require lower latency on every packet sent should be run on sockets with **TCP_NODELAY** enabled. It can be enabled through the **setsockopt** command with the sockets API:

```
# int one = 1;

# setsockopt(descriptor, SOL_TCP, TCP_NODELAY, &one, sizeof(one));
```

2. For this to be used effectively, applications must avoid doing small, logically related buffer writes. Because **TCP_NODELAY** is enabled, these small writes will make TCP send these multiple buffers as individual packets, which can result in poor overall performance.

If applications have several buffers that are logically related and that should be sent as one packet it could be possible to build a contiguous packet in memory and then send the logical packet to TCP, on a socket configured with **TCP_NODELAY**.

Alternatively, create an I/O vector and pass it to the kernel using **writev** on a socket configured with **TCP_NODELAY**.

3. Another option is to use **TCP_CORK**, which tells TCP to wait for the application to remove the cork before sending any packets. This command will cause the buffers it receives to be appended to the existing buffers. This allows applications to build a packet in kernel space, which may be required when using different libraries that provides abstractions for layers. To enable **TCP_CORK**, set it to a value of **1** using the **setsockopt** sockets API (this is known as "corking the socket"):

```
# int one = 1;

# setsockopt(descriptor, SOL_TCP, TCP_CORK, &one, sizeof(one));
```

4. When the logical packet has been built in the kernel by the various components in the application, tell TCP to remove the cork. TCP will send the accumulated logical packet right away, without waiting for any further packets from the application.

```
# int zero = 0;

# setsockopt(descriptor, SOL_TCP, TCP_CORK, &zero, sizeof(zero));
```

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- [tcp\(7\)](#)
- [setsockopt\(3p\)](#)
- [setsockopt\(2\)](#)

4.5. Setting Realtime Scheduler Priorities

Using **rtctl** to set scheduler priorities is described at [Using rtctl to Set Priorities](#). In the example given in that chapter, some kernel threads have been given a very high priority. This is to have the default priorities integrate well with the requirements of the Real Time Specification for Java (RTSJ). RTSJ requires a range of priorities from 10-89, so many kernel thread priorities are positioned at 90 and above. This avoids unpredictable behavior if a long-running Java application blocks essential system services from running.

For deployments where RTSJ is not in use, there is a wide range of scheduling priorities below 90 which are at the disposal of applications. It is usually dangerous for user level applications to run at priority 90 and above - despite the fact that the capability exists. Preventing essential system services from running can result in unpredictable behavior, including blocked network traffic, blocked virtual memory paging and data corruption due to blocked filesystem journaling.

Extreme caution should be used if scheduling any application thread above priority 89. If any application threads are scheduled above priority 89 you should ensure that the threads only run a very short code path. Failure to do so would undermine the low latency capabilities of the MRG Realtime kernel.

Setting Real-time Priority for Non-privileged Users

Generally, only root users are able to change priority and scheduling information. If you require non-privileged users to be able to adjust these settings, the best method is to add the user to the **Realtime** group.



Important

You can also change user privileges by editing the `/etc/security/limits.conf` file. This has a potential for duplication and may render the system unusable for regular users. If you *do* decide to edit this file, exercise caution and always create a copy before making changes.

Further Reading

For more information, or for further reading, the following links are related to the information given in this section.

- There is a testing utility called **signaltest** which is useful for demonstrating MRG Realtime system behavior. A whitepaper written by Arnaldo Carvalho de Melo explains this in more detail: [signaltest: Using the RT priorities](#)⁵

4.6. Dynamic Libraries Loading

When developing your MRG Realtime program, consider resolving symbols at startup. Although it can slow down program initialization, it is one way to avoid non-deterministic latencies during program execution.

Dynamic Libraries can be instructed to load at system startup by setting the **LD_BIND_NOW** variable with **ld.so**, the dynamic linker/loader.

The following is an example shell script. This script exports the **LD_BIND_NOW** variable with a non-null value of **1**, then runs a program with a scheduler policy of FIFO and a priority of **1**.

```
#!/bin/sh

LD_BIND_NOW=1
export LD_BIND_NOW
```

```
chrt --fifo 1 /opt/myapp/myapp-server &
```

Related Manual Pages

For more information, or for further reading, the following man pages are related to the information given in this section.

- [ld.so\(8\)](#)

More Information

5.1. Reporting Bugs



Important

An up-to-date listing of known issues can be found on the [MRG Realtime Wiki: Known Bugs¹](#). Always check this list before reporting a new bug.

Diagnosing a Bug

Before you file a bug report, follow these steps to diagnose where the problem has been introduced. This will greatly assist in rectifying the problem.

1. Try reproducing the problem with the standard kernel. Check that you have the latest version of the Red Hat Enterprise Linux 5.2 kernel, then boot into it from the grub menu. Try reproducing the problem. If the problem still occurs with the standard kernel, report a bug against Red Hat Enterprise Linux 5.2 *NOT* MRG Realtime.
2. If the problem does not occur when using the standard kernel, then the bug is probably the result of changes introduced in either:
 - a. The upstream kernel on which MRG Realtime is based. For example, Red Hat Enterprise Linux 5.2 is based on kernel version 2.6.18 and MRG Realtime is based on version 2.6.24.7
 - b. MRG Realtime specific enhancements Red Hat has applied on top of the baseline (2.6.24.7) kernel

To determine the problem, it is helpful to see if you can reproduce the problem on an unmodified upstream 2.6.24.7 kernel. For this reason, in addition to providing the MRG Realtime kernel, we also provide a **vanilla** kernel variant. The **vanilla** kernel is the unmodified upstream kernel build without the MRG Realtime additions.

Reporting a Bug

If you have determined that the bug is specific to MRG Realtime follow these instructions to enter a bug report:

1. You will need a [Bugzilla²](#) account. You can create one at [Create Bugzilla Account³](#).
2. Once you have a Bugzilla account, log in and click on [Enter A New Bug Report⁴](#).
3. You will need to identify the product the bug occurs in. MRG Realtime appears under Red Hat Enterprise MRG in the Red Hat products list. It is important that you choose the correct product that the bug occurs in.
4. Continue to enter the bug information by designating the appropriate component and giving a detailed problem description. When entering the problem description be sure to include details of whether you were able to reproduce the problem on the standard Red Hat Enterprise Linux 5.2 or the supplied **vanilla** kernel.

5.2. Further Reading

- Red Hat Enterprise MRG Product Information
 - <http://www.redhat.com/mrg>
- MRG Realtime Installation Guide and other Red Hat Enterprise MRG documentation
 - http://redhat.com/docs/en-US/Red_Hat_Enterprise_MRG
- Mailing List
 - To post to the list, send mail to rhemrg-users-list@redhat.com
 - Subscribe to the mailing list at: <http://post-office.corp.redhat.com/mailman/listinfo/rhemrg-users-list>

Appendix A. Revision History

Revision 1.5	Mon Jan 19 2009	Lana Brindley lbrindle@redhat.com
Added links to product page		
Revision 1.4	Mon Dec 8 2008	Lana Brindley lbrindle
BZ #472478		
Revision 1.3	Fri Nov 21 2008	Lana Brindley lbrindle
Minor updates prior to releasing document to Quality Engineering		
Revision 1.2	Thu 30 Oct 2008	Lana Brindley lbrindle@redhat.com
BZ #444837		
BZ #450647		
BZ #455259		
BZ #456028		
Revision 1.1	Thu 30 Oct 2008	Lana Brindley lbrindle@redhat.com
Updated for new build system		
Revision 1.0	Thu Jun 5 2008	Lana Brindley lbrindle@redhat.com
Completed Revision for 1.0 Release		
Revision 0.1	Mon Feb 18 2008	Lana Brindley lbrindle@redhat.com
Initial draft		

