Red Hat Enterprise MRG 2 Realtime Tuning Guide

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

Lana Brindley Alison Young Cheryn Tan

Red Hat Enterprise MRG 2 Realtime Tuning Guide

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

Lana Brindley
Red Hat Engineering Content Services

Alison Young
Red Hat Engineering Content Services

Cheryn Tan
Red Hat Engineering Content Services
cheryntan@redhat.com

Keywords

Abstract
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.

Table of Contents

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. *M*essaging — Cross platform, high performance, reliable messaging using the Advanced Message Queuing Protocol (AMQP) standard.
2. *R*ealtime — Consistent low-latency and predictable response times for applications that require microsecond latency.
3. *G*rid — 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 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 (µs). 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 6. MRG Realtime offers clients the opportunity to measure, configure and record latency times within 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 6 kernel and MRG Realtime specific tuning, which should be performed on a MRG Realtime kernel in addition to the standard Red Hat Enterprise Linux 6 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 Installation 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 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.

1.1. 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 keys 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, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from an individual key by the plus sign that connects each part of a key combination. For example:

> Press **Enter** to execute the command.
>
> Press **Ctrl**+**Alt**+**F2** to switch to a virtual terminal.

The first example highlights a particular key to press. The second example highlights a key combination: a set of three keys 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; dialog box text; labeled 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.

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:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings 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 add syntax highlighting 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

Notes are tips, shortcuts or alternative approaches 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 a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

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

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at http://access.redhat.com. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at https://www.redhat.com/mailman/listinfo. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. 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/ 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.

Chapter 1. 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 among all equal priority **SCHED_RR** threads. This policy is rarely used.

Chapter 2. General System Tuning

This chapter 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, “Setting BIOS parameters”
- Section 2.4, “Interrupt and process binding”
- Section 2.5, “File system determinism tips”

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

- Section 2.6, “Using hardware clocks for system timestamping”
- Section 2.7, “Avoid running extra applications”
- Section 2.8, “Swapping and out of memory tips”
- Section 2.9, “Network determinism tips”
- Section 2.10, “**`syslog`** tuning tips”
- Section 2.11, “The PC card daemon”
- Section 2.12, “Reduce TCP performance spikes”
- Section 2.13, “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.

Note

For instructions on installing and using Tuna, see the *Tuna User Guide*.

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, persist those parameters. The method you choose depends on the type of parameter you are setting.

Procedure 2.1. 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/vsyscall64`** will become **`kernel.vsyscall64`**.
3. Insert the new entry into the **`/etc/sysctl.conf`** file with the required parameter.

```
# Enable gettimeofday(2)
kernel.vsyscall64 = 2
```

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

```
# sysctl -p
...[output truncated]...
kernel.vsyscall64 = 2
```

Alternatively, check the *Red Hat Enterprise Linux Deployment Guide* available from the Red Hat Documentation website for information on the **`/etc/sysconfig/`** directory.

Procedure 2.2. Editing the `/etc/rc.d/rc.local` file

Warning

Use this alternative only as a last resort.

1. Adjust the command as per the Procedure 2.1, “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. Setting BIOS parameters

Because every system and BIOS vendor uses different terms and navigation methods, this section contains only general information about BIOS settings. If you have trouble locating the setting mentioned, contact the BIOS vendor.

Power Management

Anything that tries to save power by either changing the system clock frequency or by putting the CPU into various sleep states can affect how quickly the system responds to external events.

For best response times, disable power management options in the BIOS.

Error Detection and Correction (EDAC) units

EDAC units are devices used to detect and correct errors signaled from Error Correcting Code (ECC) memory. Usually EDAC options range from no ECC checking to a periodic scan of all memory nodes for errors. The higher the EDAC level, the more time is spent in BIOS, and the more likely that crucial event deadlines will be missed.

Turn EDAC off if possible. Otherwise, switch to the lowest functional level.

System Management Interrupts (SMI)

SMIs are a facility used by hardware vendors ensure the system is operating correctly. The SMI interrupt is usually not serviced by the running operating system, but by code in the BIOS. SMIs are typically used for thermal management, remote console management (IPMI), EDAC checks, and various other housekeeping tasks.

If the BIOS contains SMI options, check with the vendor and any relevant documentation to check to what extent it is safe to disable them.

Warning

While it is possible to completely disable SMIs, it is strongly recommended that you do not do this. Removing the ability for your system to generate and service SMIs can result in catastrophic hardware failure.

2.4. 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.

In practice, 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). It can also be expressed as a decimal or hexadecimal number, depending on the command you are using. 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.

Procedure 2.3. 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
```

Procedure 2.4. Partially Disabling the `irqbalance` daemon

An alternative approach to 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 has no effect on machines with only two processors, but will run effectively on a dual-core machine.

Procedure 2.5. Manually Assigning CPU Affinity to Individual IRQs

1. Check which IRQ is in use by each device 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 hexadecimal 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 will see that the number of interrupts on the chosen CPU have risen, while the numbers on the other CPUs have not changed.

Procedure 2.6. 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 a decimal or hexadecimal number. The mask argument is a bitmask that specifies which CPU cores are legal for the command or PID being modified.

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 3 (using the decimal version of the CPU mask).

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

2. It is also possible to specify more than one CPU in the bitmask. In this example, **my_embedded_process** is being instructed to execute on processors 4, 5, 6, and 7 (using the hexadecimal version of the CPU mask).

```
# taskset 0xF0 /usr/local/bin/my_embedded_process
```

3. 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
```

Important

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.

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.

2.5. File system determinism tips

The order in which journal changes arrive are sometimes not in the order that they are actually written to disk. The kernel I/O system has the option of reordering the journal changes, usually to try and make best use of available storage space. Journal activity can introduce latency through re-ordering journal changes and committing data and metadata. Often, journaling file systems can do things in such a way that they slow the system down.

The default filesystem used by Linux distributions including Red Hat Enterprise Linux 6 is a journaling file system called **ext4**. An earlier, mostly compatible implementation of the file system called **ext2** does not use journaling. Unless your organization specifically requires journaling, consider using **ext2**. In many of our best benchmark results, we utilize the **ext2** file system and consider it one of the top initial tuning recommendations.

Journaling file systems like **ext4** record the time a file was last accessed (**atime**). If using **ext2** is not a suitable solution for your system, consider disabling **atime** under **ext4** instead. Disabling **atime** increases performance and decreases power usage by limiting the number of writes to the filesystem journal.

Procedure 2.7. Disabling atime

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

```
LABEL=/        /        ext4    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=/       /       ext4    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.

- mkfs.ext2(8)
- mkfs.ext4(8)
- mount(8) - for information on **atime**, **nodiratime** and **noatime**
- chattr(1)

2.6. Using hardware clocks for system timestamping

Multiprocessor systems such as NUMA or SMP have multiple instances of hardware clocks. During boot time the kernel discovers the available clock sources and selects one to use. For the list of the available clock sources in your system, view the **/sys/devices/system/clocksource/clocksource0/available_clocksource** file:

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

In the example above, the TSC, HPET and ACPI_PM clock sources are available.

The clock source currently in use can be inspected by reading the **/sys/devices/system/clocksource/clocksource0/current_clocksource** file:

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

Changing clock sources

Sometimes the best-performing clock for a system's main application is not used due to known problems on the clock. After ruling out all problematic clocks, the system can be left with a hardware clock that is unable to satisfy the minimum requirements of a Realtime system.

Requirements for crucial applications vary on each system. Therefore, the best clock for each

application, and consequently each system, also varies. Some applications depend on clock resolution, and a clock that delivers reliable nanoseconds readings can be more suitable. Applications that read the clock too often can benefit from a clock with a smaller reading cost (the time between a read request and the result).

In all these cases it is possible to override the clock selected by the kernel, provided that you understand the side effects of this override and can create an environment which will not trigger the known shortcomings of the given hardware clock. To do so, select a clock source from the list presented in the **/sys/devices/system/clocksource/clocksource0/available_clocksource** file and write the clock's name into the **/sys/devices/system/clocksource/clocksource0/current_clocksource** file. For example, the following command sets HPET as the clock source in use:

```
# echo hpet > /sys/devices/system/clocksource/clocksource0/current_clocksource
```

Note

For a brief description of widely used hardware clocks, and to compare the performance between different hardware clocks, see the *MRG Realtime Reference Guide*.

Configuring additional boot parameters for the TSC clock

While there is no single clock which is ideal for all systems, TSC is generally the preferred clock source. To optimize the reliability of the TSC clock, you can configure additional parameters when booting the kernel, for example:

- ***idle=poll***: Forces the clock to avoid entering the idle state.
- ***processor.max_cstate=1***: Prevents the clock from entering deeper C-states (energy saving mode), so it does not become out of sync.

Note however that in both cases there will be an increase in energy consumption, as the system will always run at top speed.

Controlling power management transitions

Modern processors actively transition to higher power saving states (C-states) from lower states. Unfortunately, transitioning from a high power saving state back to a running state can consume more time than is optimal for a Realtime application. To prevent these transitions, an application can use the Power Management Quality of Service (PM QoS) interface.

With the PM QoS interface, the system can emulate the behaviour of the ***idle=poll*** and ***processor.max_cstate=1*** parameters (as listed in Configuring additional boot parameters for the TSC clock), but with a more fine-grained control of power saving states.

When an application holds the **/dev/cpu_dma_latency** file open, the PM QoS interface prevents the processor from entering deep sleep states and causing unexpected latencies when exiting deep sleep states. When the file is closed, the system returns to a power-saving state.

1. Open the **/dev/cpu_dma_latency** file. Keep the file descriptor open for the duration of the low-latency operation.
2. Write a 32-bit number to it. This number represents a maximum response time in microseconds. For the fastest possible response time, use ***0***.

An example **/dev/cpu_dma_latency** file is as follows:

```
static int pm_qos_fd = -1;

void start_low_latency(void)
{
 s32_t target = 0;

 if (pm_qos_fd >= 0)
  return;
 pm_qos_fd = open("/dev/cpu_dma_latency", O_RDWR);
 if (pm_qos_fd < 0) {
    fprintf(stderr, "Failed to open PM QOS file: %s",
            strerror(errno));
    exit(errno);
 }
 write(pm_qos_fd, &target, sizeof(target));
}

void stop_low_latency(void)
{
 if (pm_qos_fd >= 0)
    close(pm_qos_fd);
}
```

The application will first call **start_low_latency()**, perform the required latency-sensitive processing, then call **stop_low_latency()**.

Related Manual Pages

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

- *Linux System Programming* by Robert Love

2.7. Avoid running extra applications

These are common practices for improving performance, yet they are often overlooked. Here are some 'extra applications' to look for:

- Graphical desktop

  Do not run graphics where 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]...
  ```

- Mail Transfer Agents (MTA, such as Sendmail or Postfix)

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.

Important

Sendmail is used to send system-generated messages, which are executed by programs such as cron. This includes reports generated by logging functions like logwatch. You will not be able to receive these messages if sendmail is disabled.

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

  If you are not using a graphical interface like Gnome or KDE, then you probably won't need a mouse either. Remove the hardware and uninstall **gpm**.
- Automated tasks

  Check for automated **cron** or **at** jobs that could impact performance.

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)

2.8. 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. It is optimal that they remain on zero as much as possible.

Procedure 2.8. 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. The default setting is **0** which 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
0

#  echo 1 > /proc/sys/vm/panic_on_oom

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

2. It is also possible to prioritize which processes get killed by adjusting the **oom_killer** score. In **/proc/*PID*/** there are two tools labeled **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, among 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)

2.9. 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?

Transmitting packets more than once can cause 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.

In systems that transfer large amounts of data where bandwidth use is a priority, using the default value or increasing coalesce can increase bandwidth use and lower system use. For systems requiring a rapid network response, reducing or disabling coalesce is advised.

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)

2.10. syslog tuning tips

syslog can forward 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.

Procedure 2.9. 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 received 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 file system. 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 are locally logged.

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)

2.11. 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.

Procedure 2.10. 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
```

2.12. Reduce TCP performance spikes

To reduce a performance spike with relation to timestamp generation, change the values of the TCP related entries with the **sysctl** command. The timestamps kernel parameter is found at **/proc/sys/net/ipv4/tcp_timestamps**.

- Turn timestamps on with the following command:

```
# sysctl -w net.ipv4.tcp_timestamps=1
net.ipv4.tcp_timestamps = 1
```

- Turn timestamps off with the following command:

```
# sysctl -w net.ipv4.tcp_timestamps=0
net.ipv4.tcp_timestamps = 0
```

- Print the current value with the following command:

```
# sysctl net.ipv4.tcp_timestamps
net.ipv4.tcp_timestamps = 1
```

  The value **1** indicates that timestamps are on, the value **0** indicates they are off.

2.13. Reducing the TCP delayed ack timeout

Some applications that send small network packets can experience latencies due to the TCP delayed acknowledgment 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 acknowledgment systemwide.

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

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

Chapter 3. Realtime-Specific Tuning

Once you have completed the optimization 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, “Using **kdump** and **kexec** with the MRG Realtime kernel”
- Section 3.3, “TSC timer synchronization on Opteron CPUs”
- Section 3.4, “Infiniband”
- Section 3.6, “Non-Uniform Memory Access”

This chapter also includes information on performance monitoring tools:

- Section 3.8, “Using the **ftrace** utility for tracing latencies”
- Section 3.9, “Latency tracing using **trace-cmd**”

  Section 3.10, “Using **sched_nr_migrate** to limit **SCHED_OTHER** task migration.”

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 is useful but it can 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:

Table 3.1. Priority Map

Priority	Threads	Description
1	Low priority kernel threads	Priority 1 is usually reserved for those tasks that need to be just above **SCHED_OTHER**
2 - 69	Available for use	Range used for typical application priorities
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	*Only* for use by very high priority application threads
99	Watchdogs and migration	System threads that must run at the highest priority

Procedure 3.1. 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]...
   ```

   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).

2. The **`rtctl`** system startup script relies on the **`/etc/rtgroups`** file.

   To change this file, open **`/etc/rtgroups`** in your preferred text editor.

```
kthreads:*:1:*:\[.*\]
watchdog:f:99:*:\[watchdog.*\]
migration:f:99:*:\[migration\/.*\]
softirq:f:70:*:\[.*(softirq|sirq).*\]
softirq-net-tx:f:75:*:\[(softirq|sirq)-net-tx.*\]
softirq-net-rx:f:75:*:\[(softirq|sirq)-net-rx.*\]
softirq-sched:f:1:*:\[(softirq|sirq)-sched\/.*\]
rpciod:f:65:*:\[rpciod.*\]
lockd:f:65:*:\[lockd.*\]
nfsd:f:65:*:\[nfsd.*\]
hardirq:f:85:*:\[(irq|IRQ)[\-_/].*\]
```

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

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

4. 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. Using kdump and kexec with the MRG Realtime kernel

Kdump is a reliable kernel crash dumping mechanism because the crash dump is captured from the context of a freshly booted kernel and not from the context of the crashed kernel. **Kdump** uses a mechanism called **kexec** to boot into a second kernel whenever the system crashes. This second kernel, often called the crash kernel, boots with very little memory and captures the dump image.

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 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 kernel reboots, which resets the machine and brings the boot kernel back up.

Important

MRG Realtime uses the stock Red Hat Enterprise Linux 6 kernel as the kdump kernel.

There are three required procedures for enabling **kdump** under Red Hat Enterprise Linux 6. The first procedure ensures that the required RPM packages are installed on the system. The second creates the minimum configuration and modifies the grub command line using the **rt-setup-kdump** tool. The third uses a graphical system configuration tool called **system-config-kdump** to create and enable a detailed **kdump** configuration.

Procedure 3.2. Installing required kdump packages

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

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

2. Check that you have the **kexec-tools** and **system-config-kdump** packages installed.

```
# rpm -q kexec-tools system-config-kdump
kexec-tools-2.0.0-209.el6_2.5.x86_64
system-config-kdump-2.0.2.2-2.el6.noarch
```

Procedure 3.3. Creating a basic kdump kernel with rt-setup-kdump

1. Run the **rt-setup-kdump** tool by invoking it at the shell prompt as the root user. This will set the Red Hat Enterprise Linux 6 kernel to be the kdump kernel:

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

The ***--grub*** parameter adds the necessary changes to all the Realtime kernel entries listed on **/etc/grub.conf**.

2. 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  ]
```

Procedure 3.4. Enabling kdump with system-config-kdump

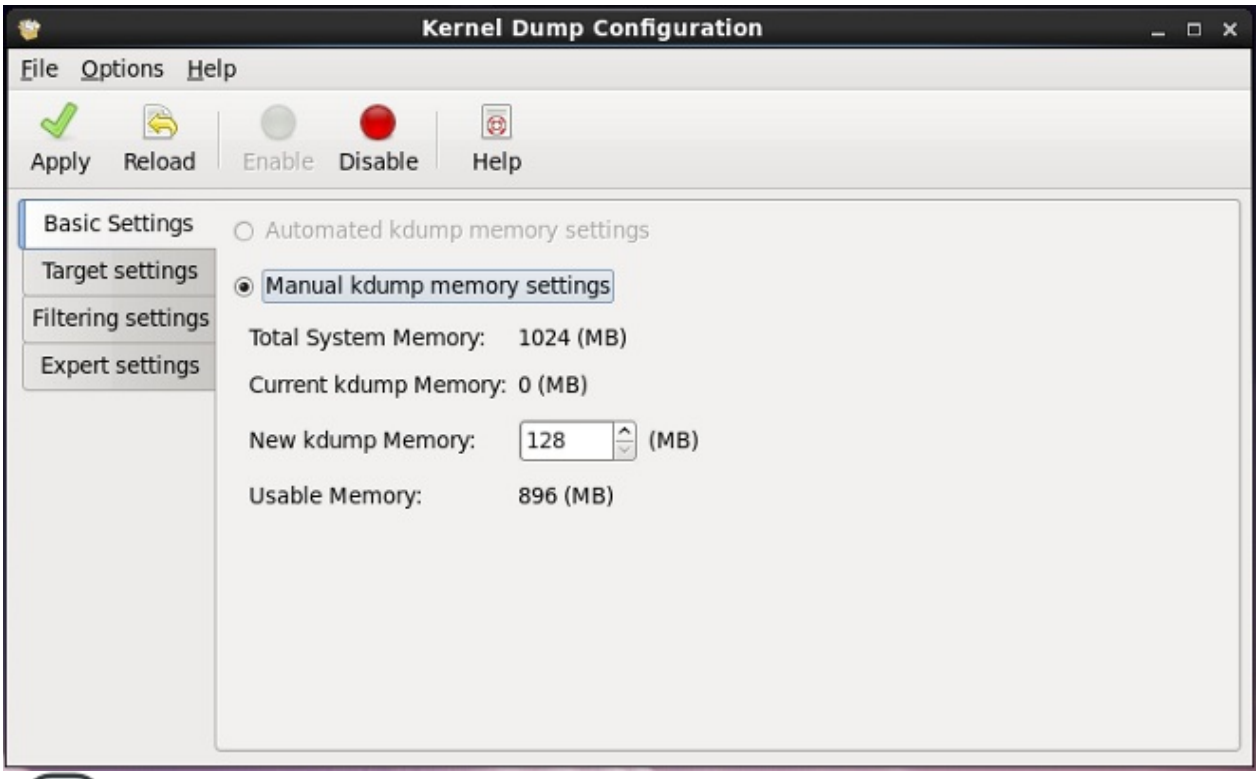
1. Select the **Kernel Crash Dumps** system tool from the **System** → **Administration** menu, or use the following command at the shell prompt:

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

2. The **Kernel Dump Configuration** window displays. On the toolbar, click the button labeled

Enable. The MRG Realtime kernel supports the ***crashkernel=auto*** parameter which automatically calculates the amount of memory necessary to accommodate the **kdump** kernel.

However, for Red Hat Enterprise Linux 6 systems with less than 4GB of RAM, the ***crashkernel=auto*** does not automatically reserve memory for the **kdump** kernel. In this case, it is necessary to manually set the amount of memory desired. You can do so by entering your required value in the **New kdump memory** field on the **Basic Settings** tab:



Note

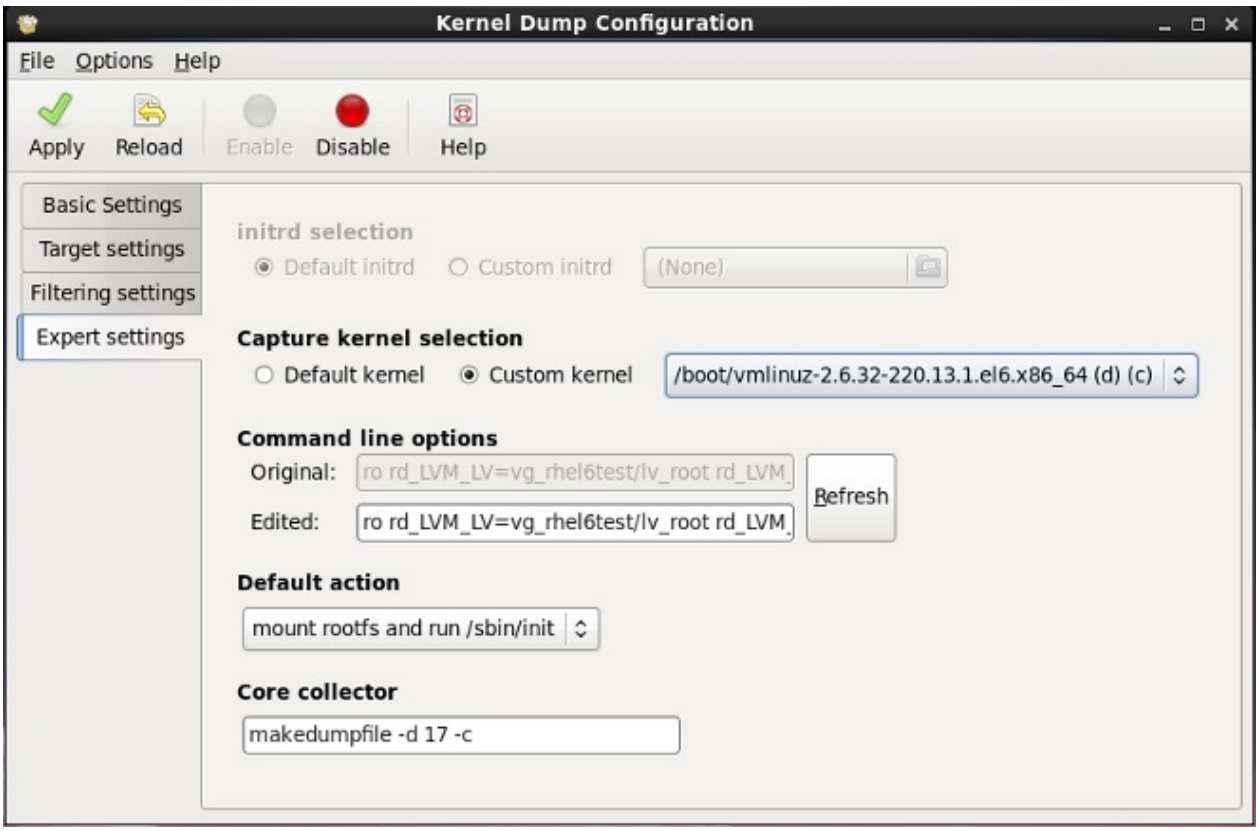
An alternative way of allocating memory for the **kdump** kernel is by manually setting the ***crashkernel=<value>*** parameter on **/etc/grub.conf**.

3. Click the **Target Settings** tab, and specify the target location for your dump file. It can be either stored as a file in a local file system, written directly to a device, or sent over a network using the NFS (Network File System) or SSH (Secure Shell) protocol.

Important

Always check the **/etc/grub.conf** file to ensure that the tool has adjusted the correct kernel. Use the MRG Realtime kernel as the default boot kernel, and use the Red Hat Enterprise Linux kernel as the crash kernel.

4. Click the **Expert Settings** tab. Under the **Capture kernel selection** field, select **Custom kernel** and specify the Red Hat Enterprise Linux 6 kernel as the **kdump** kernel.



To save your settings, click the **Apply** button on the toolbar.

5. Reboot the system to ensure that **kdump** is properly started. 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 can 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 can 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**

3.3. TSC timer synchronization on Opteron CPUs

The current generation of AMD64 Opteron processors can be susceptible to a large **gettimeofday** skew. This skew occurs when both **cpufreq** and the Time Stamp Counter (TSC) are in use. MRG Realtime provides a method to prevent this skew 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.

Procedure 3.5. Enabling TSC timer synchronization

1. Open the **/etc/grub.conf** file in your preferred text editor and add the parameter ***clocksource=tsc 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) (kernel-rtversion)
     root (hd0,0)
     kernel /vmlinuz-kernel-rtversion ro root=/dev/RHEL6/Root clocksource=tsc
   powernow-k8.tscsync=1
     initrd /initrd-kernel-rtversion.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)

3.4. 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 6.

Note

For more information see Douglas Ledford's article on Getting Started with Infiniband, however note that MRG Realtime does not support Red Hat Enterprise Linux 5.

3.5. RoCEE and High Performance Networking

RoCEE (RDMA over Converged Enhanced Ethernet) is a protocol that implements Remote Direct Memory Access (RDMA) over 10 Gigabit Ethernet networks. It allows you to maintain a consistent, high-speed environment in your datacenters while providing deterministic, low latency data transport for critical transactions.

High Performance Networking (HPN) is a set of shared libraries that provides RoCEE interfaces into the kernel. Instead of going through an independent network infrastructure, HPN places data directly into remote system memory using standard 10 Gigabit Ethernet infrastructure, resulting in less CPU overhead and reduced infrastructure costs.

Support for RoCEE and HPN under MRG Realtime does not differ from the support offered under Red Hat Enterprise Linux 6.

Note

For more information on how to set up ethernet networks, see the *Network Interfaces* chapter in the *Red Hat Enterprise Linux 6 Deployment Guide*.

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 Procedure 2.6, “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.

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)

3.7. Mount debugfs

debugfs is file system specially designed for debugging and making information available to users. Mount **debugfs** for use with MRG Realtime functions **ftrace** and **trace-cmd**.

1. Mount the kernel to **/sys/kernel/debug** and instruct it to use the **debugfs** file system.

```
# mount -t debugfs nodev /sys/kernel/debug
```

2. 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:

```
nodev /sys/kernel/debug    debugfs   defaults   0  0
```

3.8. 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 wake up, the length of time interrupts are disabled, 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 can be set up once the **trace** variant of the MRG Realtime kernel is installed and in use.

Procedure 3.6. Using the ftrace Utility

1. In the **/sys/kernel/debug/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 /sys/kernel/debug/tracing/available_tracers
wakeup preemptirqsoff preemptoff irqsoff ftrace sched_switch none
```

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 **preemptirqsoff** 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, except the **sched_switch** tracer.

sched_switch
Traces context switches between tasks.

none
Disables tracing.

2. 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** command to insert the name of the tracer into **/sys/kernel/debug/tracing/current_tracer**:

```
# echo preemptoff > /sys/kernel/debug/tracing/current_tracer
```

3. To check if the **ftrace** utility is enabled, use the **cat** command to view the **/proc/sys/kernel/ftrace_enabled** file. A value of **1** indicates that **ftrace** is enabled, and **0** indicates that it has been disabled.

```
# 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 **/proc/sys/kernel/ftrace_enabled** file.

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

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

Important

When using the **echo** command, 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 unexpected trace output.

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

The irqsoff, preemptoff, preempirqsoff, and wakeup tracers continuously monitor latencies. When they record a latency greater than the one recorded in **tracing_max_latency** the trace of that latency is recorded, and **tracing_max_latency** is updated to the new maximum time. In this way, **tracing_max_latency** will always shows the highest recorded latency since it was last reset.

To reset the maximum latency, **echo 0** into the **tracing_max_latency** file. To see only latencies greater than a set amount, **echo** the amount in microseconds:

```
# echo 0 > /sys/kernel/debug/tracing//tracing_max_latency
```

When the tracing threshold is set, it overrides the maximum latency setting. When a latency is recorded that is greater than the threshold, it will be recorded regardless of the maximum latency. When reviewing the trace file, only the last recorded latency is shown.

To set the threshold, **echo** the number of microseconds above which latencies must be recorded:

```
# echo 200 > /sys/kernel/debug/tracing//tracing_thresh
```

5. View the trace logs:

```
# cat /sys/kernel/debug/tracing/trace
```

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

```
# cat /sys/kernel/debug/tracing/trace > /tmp/lat_trace_log
```

7. The **ftrace** utility can be filtered by altering the settings in the **/sys/kernel/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 /sys/kernel/debug/tracing/set_ftrace_filter
```

8. To change the filters, **echo** the name of the function to be traced. The filter allows the use of a ***** wildcard at the beginning or end of a search term.

   The ***** wildcard can also be used at both the beginning *and* end of a word. For example: ***irq*** will select all functions that contain **irq** in the name.

   Encasing the search term and the wildcard character in double quotation marks ensures that that shell will not attempt to expand the search to the present working directory.

   Some examples of filters are:

   - Trace only the **schedule** process:

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

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

     ```
     # echo "*lock" > /sys/kernel/debug/tracing/set_ftrace_filter
     ```

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

     ```
     # echo "spin_*" > /sys/kernel/debug/tracing/set_ftrace_filter
     ```

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

     ```
     # echo "*cpu*" > /sys/kernel/debug/tracing/set_ftrace_filter
     ```

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.

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

```
# trace-cmd [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	Description
`record`	Record a trace into a trace.dat file.
`start`	Start tracing without recording into a file.
`extract`	Extract a trace from the kernel.
`stop`	Stops the kernel from recording trace data.
`reset`	Disable all kernel tracing and clear the trace buffers.
`report`	Read out the trace stored in a trace.dat file.
`split`	Parse a trace.dat file into smaller file(s).
`listen`	Listen on a network socket for trace clients.
`list`	List the available events, plugins or options.

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 *or* 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

Note

For further information about event tracing and function tracer refer to Appendix A, *Event Tracing* and Appendix B, *Function Tracer*.

3. In this example, the **trace-cmd** utility will trace a single trace point:

```
# ./trace-cmd record -e sched_wakeup ls /bin
```

3.10. Using sched_nr_migrate to limit SCHED_OTHER task migration.

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.

Procedure 3.7. 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, the value must 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
```

Chapter 4. Application Tuning and Deployment

This chapter contains tips related to enhancing and developing MRG Realtime applications.

Note

In general, try to use *POSIX* (Portable Operating System Interface) defined APIs. MRG Realtime is 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.

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 are taken when delivering a signal are not always optimal, and quite long latencies can 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 signal 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

Ulrich Drepper's Requirements of the POSIX Signal Model

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.

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

Procedure 4.1. 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;

pthread_mutex_init(&my_mutex, NULL);
```

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

Procedure 4.2. 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 futex.

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 must 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.

Procedure 4.3. Using TCP_NODELAY and TCP_CORK to improve network latency

1. Applications that require lower latency on every packet sent must 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 are to be sent as one packet, it is 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 can 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 in Procedure 3.1, "Using **rtctl** to Set Priorities". In the example given in that procedure, 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.

Use extreme caution when scheduling any application thread above priority 89. If any application threads are scheduled above priority 89, 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 can 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.

4.6. Loading dynamic libraries

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)

4.7. Using _COARSE POSIX clocks for application timestamping

Applications that frequently perform timestamps are affected by the cost of reading the clock. A high cost and amount of time used to read the clock can have a negative impact on the application's performance.

To illustrate that concept, imagine using a clock, inside a drawer, to time events being observed. If every time one has to open the drawer, get the clock and only then read the time, the cost of reading the clock is too high and can lead to missing events or incorrectly timestamping them.

Conversely, a clock on the wall would be faster to read, and timestamping would produce less interference to the observed events. Standing right in front of that wall clock would make it even faster to obtain time readings.

Likewise, this performance gain (in reducing the cost of reading the clock) can be obtained by selecting a hardware clock that has a faster reading mechanism. In MRG Realtime, a further performance gain can be acquired by using POSIX clocks with the **clock_gettime()** function to produce clock readings with the lowest cost possible.

POSIX clocks

POSIX is a standard for implementing and representing time sources. The POSIX clock can be selected by each application, without affecting other applications in the system. This is in contrast to the hardware clock as described in Section 2.6, “Using hardware clocks for system timestamping”, which is selected by the kernel and implemented across the system.

The function used to read a given POSIX clock is **clock_gettime()**, which is defined at **<time.h>**. **clock_gettime()** has a counterpart in the kernel, in the form of a system call. When the user process calls **clock_gettime()**, the corresponding C library (**glibc**) calls the **sys_clock_gettime()** system call which performs the requested operation and then returns the result to the user program.

However, this context switch from the user application to the kernel has a cost. Even though this cost is very low, if the operation is repeated thousands of times, the accumulated cost can have an impact on the overall performance of the application. To avoid that context switch to the kernel, thus making it faster to read the clock, support for the **CLOCK_MONOTONIC_COARSE** and **CLOCK_REALTIME_COARSE** POSIX clocks was created in the form of a VDSO library function.

Time readings performed by **clock_gettime()**, using one of the ***_COARSE*** clock variants, do not require kernel intervention and are executed entirely in user space, which yields a significant performance gain. Time readings for ***_COARSE*** clocks have a millisecond (ms) resolution, meaning that time intervals smaller than 1ms will not be recorded. The ***_COARSE*** variants of the POSIX clocks are suitable for any application that can accommodate millisecond clock resolution, and the benefits are more evident on systems which use hardware clocks with high reading costs.

Note

To compare the cost and resolution of reading POSIX clocks with and without the ***_COARSE*** prefix, see the *MRG Realtime Reference Guide*.

Older MRG Realtime kernels provided a system-wide timestamp function. Although it did not require

changes in the applications, it affected all applications on the system. The current MRG Realtime kernel offers a more granular solution, allowing any application to select a POSIX clock for optimal performance gain without affecting other applications. Usually the only required change is to replace **CLOCK_MONOTONIC** with **CLOCK_MONOTONIC_COARSE** on the **clock_gettime()** calls in the source code, for example:

Example 4.1. Using the *_COARSE* clock variant in clock_gettime

```
#include <time.h>

main()
{
 int rc;
 long i;
 struct timespec ts;

 for(i=0; i<10000000; i++) {
  rc = clock_gettime(CLOCK_MONOTONIC_COARSE, &ts);
 }
}
```

You can improve upon the example above, for example by using more strings to verify the return code of **clock_gettime()**, to verify the value of the ***rc*** variable, or to ensure the content of the ***ts*** structure is to be trusted. The **clock_gettime()** manpage provides more information to help you write more reliable applications.

Important

Programs using the **clock_gettime()** function must be linked with the **rt** library by adding ***'-lrt'*** to the **gcc** command line.

```
cc clock_timing.c -o clock_timing -lrt
```

Related Manual Pages

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

- clock_gettime()
- *Linux System Programming* by Robert Love
- *Understanding The Linux Kernel* by Daniel P. Bovet and Marco Cesati

Chapter 5. More Information

5.1. Reporting Bugs

Diagnosing a Bug

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

1. Check that you have the latest version of the Red Hat Enterprise Linux 6 kernel, then boot into it from the grub menu. Try reproducing the problem. If the problem still occurs, report a bug against Red Hat Enterprise Linux 6 *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:
   - The upstream kernel on which MRG Realtime is based. For example, the Red Hat Enterprise Linux 6 kernel is based on version 2.6.32 and MRG Realtime is based on version 3.6
   - MRG Realtime specific enhancements Red Hat has applied on top of the baseline (3.6) kernel

   To determine the problem, try to reproduce the problem on an unmodified upstream 3.6 kernel. For this reason, in addition to providing the MRG Realtime kernel, we also provide a **`vanilla`** kernel variant. The **`vanilla`** kernel is the 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. Create a Bugzilla account.
2. Log in and click on Enter A New Bug Report.
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 6 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
  - https://access.redhat.com/knowledge/docs/Red_Hat_Enterprise_MRG/
- Red Hat Knowledgebase
  - https://access.redhat.com/knowledge/search

Event Tracing

```
    Event Tracing

  Documentation written by Theodore Ts'o
  Updated by Li Zefan and Tom Zanussi

1. Introduction
===============

Tracepoints (see Documentation/trace/tracepoints.txt) can be used
without creating custom kernel modules to register probe functions
using the event tracing infrastructure.

Not all tracepoints can be traced using the event tracing system;
the kernel developer must provide code snippets which define how the
tracing information is saved into the tracing buffer, and how the
tracing information should be printed.

2. Using Event Tracing
======================

2.1 Via the 'set_event' interface
---------------------------------

The events which are available for tracing can be found in the file
/sys/kernel/debug/tracing/available_events.

To enable a particular event, such as 'sched_wakeup', simply echo it
to /sys/kernel/debug/tracing/set_event. For example:

 # echo sched_wakeup >> /sys/kernel/debug/tracing/set_event

[ Note: '>>' is necessary, otherwise it will firstly disable
  all the events. ]

To disable an event, echo the event name to the set_event file prefixed
with an exclamation point:

 # echo '!sched_wakeup' >> /sys/kernel/debug/tracing/set_event

To disable all events, echo an empty line to the set_event file:

 # echo > /sys/kernel/debug/tracing/set_event

To enable all events, echo '*:*' or '*:' to the set_event file:

 # echo *:* > /sys/kernel/debug/tracing/set_event

The events are organized into subsystems, such as ext4, irq, sched,
etc., and a full event name looks like this: <subsystem>:<event>.  The
subsystem name is optional, but it is displayed in the available_events
file.  All of the events in a subsystem can be specified via the syntax
"<subsystem>:*"; for example, to enable all irq events, you can use the
command:

 # echo 'irq:*' > /sys/kernel/debug/tracing/set_event

2.2 Via the 'enable' toggle
---------------------------

The events available are also listed in /sys/kernel/debug/tracing/events/ hierarchy
```

```
of directories.

To enable event 'sched_wakeup':

 # echo 1 > /sys/kernel/debug/tracing/events/sched/sched_wakeup/enable

To disable it:

 # echo 0 > /sys/kernel/debug/tracing/events/sched/sched_wakeup/enable

To enable all events in sched subsystem:

 # echo 1 > /sys/kernel/debug/tracing/events/sched/enable

To enable all events:

 # echo 1 > /sys/kernel/debug/tracing/events/enable

When reading one of these enable files, there are four results:

 0 - all events this file affects are disabled
 1 - all events this file affects are enabled
 X - there is a mixture of events enabled and disabled
 ? - this file does not affect any event

2.3 Boot option
---------------

In order to facilitate early boot debugging, use boot option:

 trace_event=[event-list]

The format of this boot option is the same as described in section 2.1.

3. Defining an event-enabled tracepoint
=======================================

See The example provided in samples/trace_events

4. Event formats
================

Each trace event has a 'format' file associated with it that contains
a description of each field in a logged event.  This information can
be used to parse the binary trace stream, and is also the place to
find the field names that can be used in event filters (see section 5).

It also displays the format string that will be used to print the
event in text mode, along with the event name and ID used for
profiling.

Every event has a set of 'common' fields associated with it; these are
the fields prefixed with 'common_'.  The other fields vary between
events and correspond to the fields defined in the TRACE_EVENT
definition for that event.

Each field in the format has the form:

     field:field-type field-name; offset:N; size:N; signed:N;
```

```
where offset is the offset of the field in the trace record and size
is the size of the data item, in bytes, signed will be 0 or 1 denoting
if the type of field is signed or not.

For example, here's the information displayed for the 'sched_wakeup'
event:

# cat /sys/kernel/debug/tracing/events/sched/sched_wakeup/format
name: sched_wakeup
ID: 62
format:
 field:unsigned short common_type; offset:0; size:2; signed:0;
 field:unsigned char common_flags; offset:2; size:1; signed:0;
 field:unsigned char common_preempt_count; offset:3; size:1; signed:0;
 field:int common_pid; offset:4; size:4; signed:1;
 field:int common_lock_depth; offset:8; size:4; signed:1;

 field:char comm[TASK_COMM_LEN]; offset:12; size:16; signed:1;
 field:pid_t pid; offset:28; size:4; signed:1;
 field:int prio; offset:32; size:4; signed:1;
 field:int success; offset:36; size:4; signed:1;
 field:int target_cpu; offset:40; size:4; signed:1;

print fmt: "comm=%s pid=%d prio=%d success=%d target_cpu=%03d", REC->comm, REC-
>pid, REC->prio, REC->success, REC->target_cpu

This event contains 10 fields, the first 5 common and the remaining 5
event-specific.  All the fields for this event are numeric, except for
'comm' which is a string, a distinction important for event filtering.

5. Event filtering
==================

Trace events can be filtered in the kernel by associating boolean
'filter expressions' with them.  As soon as an event is logged into
the trace buffer, its fields are checked against the filter expression
associated with that event type.  An event with field values that
'match' the filter will appear in the trace output, and an event whose
values don't match will be discarded.  An event with no filter
associated with it matches everything, and is the default when no
filter has been set for an event.

5.1 Expression syntax
---------------------

A filter expression consists of one or more 'predicates' that can be
combined using the logical operators '&&' and '||'.  A predicate is
simply a clause that compares the value of a field contained within a
logged event with a constant value and returns either 0 or 1 depending
on whether the field value matched (1) or didn't match (0):

   field-name relational-operator value

Parentheses can be used to provide arbitrary logical groupings and
double-quotes can be used to prevent the shell from interpreting
operators as shell meta characters.

The field-names available for use in filters can be found in the
'format' files for trace events (see section 4).
```

```
The relational-operators depend on the type of the field being tested:

The operators available for numeric fields are:

==, !=, <, <=, >, >=

And for string fields they are:

==, !=

Currently, only exact string matches are supported.

Currently, the maximum number of predicates in a filter is 16.

5.2 Setting filters
-------------------

A filter for an individual event is set by writing a filter expression
to the 'filter' file for the given event.

For example:

# cd /sys/kernel/debug/tracing/events/sched/sched_wakeup
# echo "common_preempt_count > 4" > filter

A slightly more involved example:

# cd /sys/kernel/debug/tracing/events/signal/signal_generate
# echo "((sig >= 10 && sig < 15) || sig == 17) && comm != bash" > filter

If there is an error in the expression, you'll get an 'Invalid
argument' error when setting it, and the erroneous string along with
an error message can be seen by looking at the filter e.g.:

# cd /sys/kernel/debug/tracing/events/signal/signal_generate
# echo "((sig >= 10 && sig < 15) || dsig == 17) && comm != bash" > filter
-bash: echo: write error: Invalid argument
# cat filter
((sig >= 10 && sig < 15) || dsig == 17) && comm != bash
^
parse_error: Field not found

Currently the caret ('^') for an error always appears at the beginning of
the filter string; the error message should still be useful though
even without more accurate position info.

5.3 Clearing filters
--------------------

To clear the filter for an event, write a '0' to the event's filter
file.

To clear the filters for all events in a subsystem, write a '0' to the
subsystem's filter file.

5.3 Subsystem filters
---------------------

For convenience, filters for every event in a subsystem can be set or
cleared as a group by writing a filter expression into the filter file
```

```
at the root of the subsystem.  Note however, that if a filter for any
event within the subsystem lacks a field specified in the subsystem
filter, or if the filter can't be applied for any other reason, the
filter for that event will retain its previous setting.  This can
result in an unintended mixture of filters which could lead to
confusing (to the user who might think different filters are in
effect) trace output.  Only filters that reference just the common
fields can be guaranteed to propagate successfully to all events.

Here are a few subsystem filter examples that also illustrate the
above points:

Clear the filters on all events in the sched subsytem:

# cd /sys/kernel/debug/tracing/events/sched
# echo 0 > filter
# cat sched_switch/filter
none
# cat sched_wakeup/filter
none

Set a filter using only common fields for all events in the sched
subsytem (all events end up with the same filter):

# cd /sys/kernel/debug/tracing/events/sched
# echo common_pid == 0 > filter
# cat sched_switch/filter
common_pid == 0
# cat sched_wakeup/filter
common_pid == 0

Attempt to set a filter using a non-common field for all events in the
sched subsytem (all events but those that have a prev_pid field retain
their old filters):

# cd /sys/kernel/debug/tracing/events/sched
# echo prev_pid == 0 > filter
# cat sched_switch/filter
prev_pid == 0
# cat sched_wakeup/filter
common_pid == 0
```

Function Tracer

```
  ftrace - Function Tracer
=======================

Copyright 2012 Red Hat Inc.
   Author:   Steven Rostedt <srostedt@redhat.com>
  License:   The GNU Free Documentation License, Version 1.2
               (dual licensed under the GPL v2)
Reviewers:   Elias Oltmanns, Randy Dunlap, Andrew Morton,
      John Kacur, and David Teigland.
Written for: 3.2.16-rt27-mrg

Introduction
------------

Ftrace is an internal tracer designed to help out developers and
designers of systems to find what is going on inside the kernel.
It can be used for debugging or analyzing latencies and
performance issues that take place outside of user-space.

Although ftrace is the function tracer, it also includes an
infrastructure that allows for other types of tracing. Some of
the tracers that are currently in ftrace include a tracer to
trace the time it takes for a high priority task to run after it
was woken up, the time interrupts are disabled.

Static trace event points are also spread throughout the kernel.
These trace events can be enabled to show specific information about
a part of the kernel (like context switches, system calls, interrupts,
memory, etc). The nice thing about the trace events is that they show
up in all tracers and even the nop (off) tracer.


Implementation Details
----------------------

See ftrace-design.txt for details for arch porters and such.


The File System
---------------

Ftrace uses the debugfs file system to hold the control files as
well as the files to display output.

When debugfs is configured into the kernel (which selecting any ftrace
option will do) the directory /sys/kernel/debug will be created. To mount
this directory, you can add to your /etc/fstab file:

 debugfs       /sys/kernel/debug          debugfs defaults        0       0

Or you can mount it at run time with:

 mount -t debugfs nodev /sys/kernel/debug

For quicker access to that directory you may want to make a soft link to
it:

 ln -s /sys/kernel/debug /debug

Any selected ftrace option will also create a directory called tracing
```

```
within the debugfs. The rest of the document will assume that you are in
the ftrace directory (cd /sys/kernel/debug/tracing) and will only concentrate
on the files within that directory and not distract from the content with
the extended "/sys/kernel/debug/tracing" path name.

That's it! (assuming that you have ftrace configured into your kernel)

After mounting the debugfs, you can see a directory called
"tracing".  This directory contains the control and output files
of ftrace. Here is a list of some of the key files:


 Note: all time values are in microseconds.

  current_tracer:

 This is used to set or display the current tracer
 that is configured.

  available_tracers:

 This holds the different types of tracers that
 have been compiled into the kernel. The
 tracers listed here can be configured by
 echoing their name into current_tracer.

  trace:

 This file holds the output of the trace in a human
 readable format (described below). Note, tracing will
 be temporarily disabled while this file is read. The
 "trace" file is static, and if the tracer is not
 adding more data, it will display the same
 information every time it is read.

  trace_pipe:

 The output is the same as the "trace" file but this
 file is meant to be streamed with live tracing.
 Reads from this file will block until new data is
 retrieved.  Unlike the "trace" file, this file is a
 consumer. This means reading from this file causes
 sequential reads to display more current data. Once
 data is read from this file, it is consumed, and
 will not be read again with a sequential read.

        This file will not disable tracing when read, like the
        "trace" file does.

  trace_options:

 This file lets the user control the amount of data
 that is displayed in one of the above output
 files. Some of the options even modify the behavior
 of the trace.

  tracing_max_latency:

 Some of the tracers record the max latency.
 For example, the time interrupts are disabled.
```

```
This time is saved in this file. The max trace
will also be stored, and displayed by "trace".
A new max trace will only be recorded if the
latency is greater than the value in this
file. (in microseconds)

By writing an ASCII '0' into this file, it will
reset the max latency and the next latency will be
recorded. Writing a ASCII number other than zero
(ie. "123") will set the max latency to that number
and the next latency to be recorded will have to be
greater than the number in tracing_max_latency.

 buffer_size_kb:

This sets or displays the number of kilobytes each CPU
buffer can hold. The tracer buffers are the same size
for each CPU. The displayed number is the size of the
CPU buffer and not the total size of all buffers. The
trace buffers are allocated in pages (blocks of memory
that the kernel uses for allocation, usually 4 KB in size).
If the last page allocated has room for more bytes
than requested, the rest of the page will be used,
making the actual allocation bigger than requested.
( Note, the size may not be a multiple of the page size
  due to buffer management overhead. )

 buffer_total_size_kb

       This shows the total size of all allocated buffers.
       The buffer_size_kb shows the size of each individual CPU
       buffer. To know the full buffer size of all the individual
       CPU buffers combined, view this file.

 tracing_cpumask:

This is a mask that lets the user only trace
on specified CPUS. The format is a hex string
representing the CPUS.

 set_ftrace_filter:

When dynamic ftrace is configured in (see the
section below "dynamic ftrace"), the code is dynamically
modified (code text rewrite) to disable calling of the
function profiler (mcount). This lets tracing be configured
in with practically no overhead in performance.  This also
has a side effect of enabling or disabling specific functions
to be traced. Echoing names of functions into this file
will limit the trace to only those functions.

 set_ftrace_notrace:

This has an effect opposite to that of
set_ftrace_filter. Any function that is added here will not
be traced. If a function exists in both set_ftrace_filter
and set_ftrace_notrace, the function will _not_ be traced.

 set_ftrace_pid:
```

```
 Have the function tracer only trace a single thread.

  set_graph_function:

 Set a "trigger" function where tracing should start
 with the function graph tracer (See the section
 "dynamic ftrace" for more details). The functions here
        will make the function_graph tracer show just what these
        functions call (max of 32 functions can be added here).

  available_filter_functions:

 This lists the functions that ftrace
 has processed and can trace. These are the function
 names that you can pass to "set_ftrace_filter" or
 "set_ftrace_notrace". (See the section "dynamic ftrace"
 below for more details.)


The Tracers
-----------

Here is the list of current tracers that may be configured.

  "function"

 Function call tracer to trace all kernel functions.

  "function_graph"

 Similar to the function tracer except that the
 function graph tracer traces both the entry and exit
 of each function. It then provides the ability to draw
 a graph of the function calls similar to what C code
 source would look like, as well as the time spent in
 that particular function. Note, the time of the function
 will include the overhead of the tracer if that function
 called other functions that were traced.

  "irqsoff"

 Traces the areas that disable interrupts and saves
 the trace with the longest max latency.
 See tracing_max_latency. When a new max is recorded,
 it replaces the old trace. It is best to view this
 trace with the latency-format option enabled.

 (Note latency-format is automatically enabled when
  "irqsoff" tracer is enabled.)

  "wakeup_rt"

        Traces and records the max latency that it takes for
        the highest priority task to get scheduled after
        it has been woken up. This differs from the "wakeup" tracer
        as it only considers tasks with a real-time priority.
        As non-real-time tasks may take longer to wake up due
        to fair balance scheduling, they can hide a long latency
        of a real-time task. Use this tracer if you are only
        concerned about the wake up latency of real-time tasks.
```

```
        (Note latency-format is automatically enabled when
         "wakeup_rt" tracer is enabled.)

  "preemptoff"

 Similar to irqsoff but traces and records the amount of
 time for which preemption is disabled.

 (Note latency-format is automatically enabled when
  "preemptsoff" tracer is enabled.)

  "preemptirqsoff"

 Similar to irqsoff and preemptoff, but traces and
 records the largest time for which irqs and/or preemption
 is disabled.

 (Note latency-format is automatically enabled when
  "preemptirqsoff" tracer is enabled.)

  "wakeup"

 Traces and records the max latency that it takes for
 the highest priority task to get scheduled after
 it has been woken up.

 (Note latency-format is automatically enabled when
  "wakeup" tracer is enabled.)

  "nop"

 This is the "trace nothing" tracer. To remove all
 tracers from tracing simply echo "nop" into
 current_tracer.

 Note, this is also useful to view only trace events.

Trace Events
------------

Along with the tracers, there are trace events that are static points
within the kernel that can be enabled or disabled to trace. When
an event is enabled, it will be recorded within the recording
of the tracer. All tracers can view trace events.

For more information about trace events, see:

  Documentation/trace/events.txt


Examples of using the tracer
----------------------------

Here are typical examples of using the tracers when controlling
them only with the debugfs interface (without using any
user-land utilities).

Output format:
--------------
```

```
Here is an example of the output format of the file "trace"

                                --------
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
            bash-2030  [001] 14035.027994: prefetchw <-__kmalloc
            bash-2030  [001] 14035.027997: alloc_fdmem <-alloc_fdtable
            bash-2030  [001] 14035.027997: __kmalloc <-alloc_fdmem
            bash-2030  [001] 14035.027998: __find_general_cachep <-__kmalloc
                                --------

A header is printed with the tracer name that is represented by
the trace. In this case the tracer is "function". Then a header
showing the format. Task name "bash", the task PID "2030", the
CPU that it was running on "001", the timestamp in <secs>.<usecs>
format, the function name that was traced "prefetchw" and the
parent function that called this function "__kmalloc". The
timestamp is the time at which the function was entered.

The prio is the internal kernel priority, which is the inverse
of the priority that is usually displayed by user-space tools.
Zero represents the highest priority (99). Prio 100 starts the
"nice" priorities with 100 being equal to nice -20 and 139 being
nice 19.


Latency trace format
--------------------

When the latency-format option is enabled, the trace file gives
somewhat more information to see why a latency happened.
Some tracers automatically enable the latency format, but you can enable
or disable it for any tracer:

Enabling:
  # echo latency-format > trace_options
 or
  # echo 1 > options/latency-format

Disabling:
  # echo nolatency-format > trace_options
 or
  # echo 0 > options/latency-format

Here is a typical trace.

# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.2.16-rt27
# --------------------------------------------------------------------
# latency: 49 us, #130/130, CPU#2 | (Mreempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: swapper/2-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: __schedule
#  => ended at:   finish_task_switch
#
```

```
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| / _--=> migrate-disable
#                ||||| /     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      |||||  \    |   /
ksoftirq-16      2d..1.    0us : _raw_spin_lock_irq <__schedule
ksoftirq-16      2d..1.    0us : add_preempt_count <_raw_spin_lock_irq
ksoftirq-16      2d..2.    0us : do_raw_spin_lock <_raw_spin_lock_irq
ksoftirq-16      2d..2.    1us : signal_pending_state <__schedule


This shows that the current tracer is "irqsoff" tracing the time
for which interrupts were disabled. It gives the trace version
and the version of the kernel upon which this was executed on
(3.2.16-rt27). Then it displays the max latency in microsecs (97
us). The number of trace entries displayed and the total number
recorded (both are three: #130/130). The type of preemption that was
used (preempt). VP, KP, SP, and HP are always zero and are
reserved for later use. #P is the number of online CPUS (#P:4).

The task is the process that was running when the latency
occurred. (swapper/2 pid: 0).

The start and stop (the functions in which the interrupts were
disabled and enabled respectively) that caused the latencies:

  __schedule is where the interrupts were disabled.
  finish_task_switch is where they were enabled again.

The next lines after the header are the trace itself. The header
explains which is which.

  cmd: The name of the process in the trace.

  pid: The PID of that process.

  CPU#: The CPU which the process was running on.

  irqs-off: 'd' interrupts are disabled. '.' otherwise.
     Note: If the architecture does not support a way to
    read the irq flags variable, an 'X' will always
    be printed here.

  need-resched: 'N' task need_resched is set, '.' otherwise.

  hardirq/softirq:
 'H' - hard irq occurred inside a softirq.
 'h' - hard irq is running
 's' - soft irq is running
 '.' - normal context.

  preempt-depth: The level of preempt_disabled

  migrate-disable: for the real-time kernel, tasks can be temporarially
            bound to a CPU. When this occurs, the tasks migrate-disable
```

```
             count is incremented. This will show a number if migrate-disable
             is set to something other than zero, and a '.' if it is zero.

The above is mostly meaningful for kernel developers.

  time: When the latency-format option is enabled, the trace file
 output includes a timestamp relative to the start of the
 trace. This differs from the output when latency-format
 is disabled, which includes an absolute timestamp from boot up.

  delay: This is just to help catch your eye a bit better. And
  needs to be fixed to be only relative to the same CPU
  (but doesn't matter for (preempt)(irqs)off tracers as
  they are for single CPUs anyway).
  The marks are determined by the difference between this
  current trace event and the next trace event.
   '!' - greater than preempt_mark_thresh (default 100)
   '+' - greater than 1 microsecond
   ' ' - less than or equal to 1 microsecond.

  The rest is the same as the 'trace' file.


trace_options
-------------

The trace_options file is used to control what gets printed in
the trace output. To see what is available, simply cat the file:

  cat trace_options
   print-parent
   nosym-offset
   nosym-addr
   noverbose
   noraw
   nohex
   nobin
   noblock
   nostacktrace
   trace_printk
   noftrace_preempt
   nobranch
   annotate
   nouserstacktrace
   nosym-userobj
   noprintk-msg-only
   context-info
   latency-format
   sleep-time
   graph-time
   record-cmd
   overwrite
   nodisable_on_free

Some options appear only when a tracer is active:

  [blk]
   noblk_classic

  [function]
```

```
    nofunc_stack_trace

  [function_graph]
    nofuncgraph-overrun
    funcgraph-cpu
    funcgraph-overhead
    nofuncgraph-proc
    funcgraph-duration
    nofuncgraph-abstime
    funcgraph-irqs

  [wakeup, wakeup_rt, irqsoff, preemptoff, preemptirqsoff]
    nodisplay-graph

To disable one of the options, echo in the option prepended with
"no".

  echo noprint-parent > trace_options

To enable an option, leave off the "no".

  echo sym-offset > trace_options

Each of these options also exist in the options directory, and can
be enabled and disabled by writing in an ASCII '1' or '0' respectively.

    echo 0 > options/print-parent
    echo 1 > options/sym-offset

Here are the available options:

  print-parent - On function traces, display the calling (parent)
    function as well as the function being traced.

  print-parent:
    bash-4000  [01]  1477.606694: simple_strtoul <-strict_strtoul

  noprint-parent:
    bash-4000  [01]  1477.606694: simple_strtoul


  sym-offset - Display not only the function name, but also the
        offset in the function. For example, instead of
        seeing just "ktime_get", you will see
        "ktime_get+0xb/0x20".

  sym-offset:
    bash-4000  [01]  1477.606694: simple_strtoul+0x6/0xa0

  sym-addr - this will also display the function address as well
      as the function name.

  sym-addr:
    bash-4000  [01]  1477.606694: simple_strtoul <c0339346>

  verbose - This deals with the trace file when the
            latency-format option is enabled.

     bash  4000 1 0 00000000 00010a95 [58127d26] 1720.415ms \
     (+0.000ms): simple_strtoul (strict_strtoul)
```

```
  raw - This will display raw numbers. This option is best for
 use with user applications that can translate the raw
 numbers better than having it done in the kernel.

  hex - Similar to raw, but the numbers will be in a hexadecimal
 format.

  bin - This will print out the formats in raw binary numbers
        It is still ASCII, just not human readable.

  block - deprecated

  stacktrace - This is one of the options that changes the trace
        itself. When a trace is recorded, so is the stack
        of functions. This allows for back traces of
        trace sites. This does not affect the function or
        function graph events.

  userstacktrace - This option changes the trace. It records a
     stacktrace of the current userspace thread.
     Note, this will go to the user space function
     and depending if the user space application
     was compiled with frame pointers, it may or
     may not go deeper into the user space call stack.

  sym-userobj - when user stacktrace are enabled, look up which
  object the address belongs to, and print a
  relative address. This is especially useful when
  ASLR is on, otherwise you don't get a chance to
  resolve the address to object/file/line after
  the app is no longer running

  The lookup is performed when you read
  trace,trace_pipe. Example:

  a.out-1623  [000] 40874.465068: /root/a.out[+0x480] <-/root/a.out[+0
x494] <- /root/a.out[+0x4a8] <- /lib/libc-2.7.so[+0x1e1a6]

  context-info - This prints out the prefix to most events. When disabled
            only the content of the event is shown.

     content-info:

    <idle>-0  [000] 63974.137819:    0:120:R   + [000]     5: 50:S sirq-timer/0

     nocontent-info:

     0:120:R   + [000]     5: 50:S sirq-timer/0


  trace_printk - When the kernel has trace_printk()s used, this option
                 can disable them. Otherwise they always write into the
   ring buffer.

  printk-msg-only - When trace_printk()s are used in the kernel,
        sometimes only the printk message is desired.
      Like nocontext-info, enabling printk-msg-only
      will remove all context from trace_printks only.
```

```
For a - trace_printk("jiffies are %ld\n", jiffies);

  noprintk-msg-only:

<...>-2866  [003] 24152.494117: ftrace_print_test: jiffies are 4318859691

  printk-msg-only:

 jiffies are 4318859691


ftrace_preempt - When the function tracer is running, it disables
      interrupts while it records its trace. Enabling
   ftrace_preempt, will make the function trace
   only disable preemption. But because function
   tracing must disable tracing to prevent recursion
   this may miss tracing interrupts that happen while
   a function was being traced.

branch - When the branch tracer is configured, by enabling the
    branch option, all locations that have likely() and
  unlikely() branch annotations in the kernel, will be
  traced. Note, this has very high overhead.

annotate - Because the ring buffer is split into per cpu buffers,
      to prevent confusion about when a CPU buffer starts
    compared to the other CPU buffers, an annotation is
    displayed. This option is can disable the annotation.

      ##### CPU 1 buffer started ####

sched-tree - trace all tasks that are on the runqueue, at
      every scheduling event. Will add overhead if
      there's a lot of tasks running at once.

latency-format - This option changes the trace. When
                 it is enabled, the trace displays
                 additional information about the
                 latencies, as described in "Latency
                 trace format".

sleep-time - When the function graph tracer is running, the time
        it schedules out is also recorded. When the task
      schedules back in, the time it scheduled out is
      also included in the time of the function.
      When "sleep-time" is disabled, the time a task is
      scheduled out is not included.

   sleep-time:

        0) ! 388.106 us  |  }

   nosleep-time:

        2) + 13.116 us   |  }

graph-time - This is used with the function profiler when function
             graph tracing is enabled. The timing for functions by
             default will be the entire time a function is running
             including all the functions it calls. If you are more
```

```
                interested in only the actual function time, not counting
                the time spent in other functions that it may call, then
                disable the graph-time option.

  record-cmd - The task names when traced are recorded into a small buffer
               during task switches. For tracers this is automatic, and by
               default, event tracing will do the same. But if you do not
               care about the name of the task, disable "record-cmd".

    record-cmd:

            bash-5288  [003] 10196.848377: irq_handler_entry: irq=21 name=eth0

    norecord-cmd:

            <...>-5288  [003] 10196.848377: irq_handler_entry: irq=21 name=eth0

  overwrite - By default, when the ring buffer fills up on a CPU, it will start
              overwriting the older data to make room for the newer data.
              If you prefer a producer/consumer approach where the writer must
wait
              for the reader, then disable "overwrite".

  disable_on_free - By writing any value into the file free_buffer, will cause the
                    ring buffer to shrink to a minimum, and all allocated pages
                    will be freed. If this option is set, writing into the ring
buffer
                    will be disabled as well. The ring buffer still maintains a
few
                    pages when set to its minimum, but it may not make sense to
keep
                    writing to it.

Some options only appear when a tracer is set in current_tracer:

 [function]
   func_stack_trace - This is similar to the stacktrace option for events. When
                      enabled, each function that is traced will have its stack
dump
                      as well.

                   CAUTION: Do not enable this for all functions, it may cause the
system
                   to live lock. Think about it, every kernel function called is not
only
                   being traced, but having its stack traced as well.

                   Only use it when filtering a few functions. See set_ftrace_filter
below.

 [function_graph]
   funcgraph-overrun - Function graph tracing only traces a finite depth within a
                       function. If it exceeds this depth, then it is considered
an
                       overrun. If you are interested to see if any overruns
happened,
                       enable "funcgraph-overrun".

   funcgraph-cpu - By default the output shows the CPU number for each trace entry.
                   To suppress this, disable "funcgraph-cpu".
```

```
   funcgraph-overhead - By default, if a function took over a certain amount of
time,
                        a character is displayed by the duration.
                        ('!' - greater than 100us, '+' greater than 1us)
                        To suppress this, disable "funcgraph-overhead".

   funcgraph-proc - By default (to save room), the task and pid are not shown in
output
                    of the trace. To display these, enable "funcgraph-proc".

   funcgraph-duration - By default, the time each function took is displayed in the
                        output. To suppress this, disable "funcgraph-duration".

   funcgraph-abstime - By default (to save room), the timestamp is not displayed.
                       To display the absolute timestamp, enable "funcgraph-
abstime".

   funcgraph-irqs - By default, when an interrupt is detected, it is annotated with
                    "==========>" on entry, and "<==========" on exit of the
                    interrupt. To suppress this, disable "funcgraph-irqs".

  [wakeup, wakeup_rt, irqsoff, preemptoff, preemptirqsoff]

  display-graph: By default, the latency tracers (irqsoff, preempoff,
preempirqsoff,
                 wakeup, and wakeup_rt) will show a function trace where the
latency
                 was detected. By enabling "display-graph" and having function
graph
                 tracer configured in, the latency tracers will use the function
                 graph tracer instead. Note, the function graph tracer produces
                 much more overhead than the function tracer, which will make the
latencies
                 even larger. Requires that ftrace_enabled is set (see next
section).

ftrace_enabled
--------------

The following tracers (listed below) give different output
depending on whether or not the sysctl ftrace_enabled is set. To
set ftrace_enabled, one can either use the sysctl function or
set it via the proc file system interface.

  sysctl kernel.ftrace_enabled=1

 or

  echo 1 > /proc/sys/kernel/ftrace_enabled

To disable ftrace_enabled simply replace the '1' with '0' in the
above commands.

When ftrace_enabled is set the latency tracers will also record the
functions that are within the trace. The following descriptions of the
tracers will also show an example with ftrace enabled.


irqsoff
```

```
-------

When interrupts are disabled, the CPU can not react to any other
external event (besides NMIs and SMIs). This prevents the timer
interrupt from triggering or the mouse interrupt from letting
the kernel know of a new mouse event. The result is a latency
with the reaction time.

The irqsoff tracer tracks the time for which interrupts are
disabled. When a new maximum latency is hit, the tracer saves
the trace leading up to that latency point so that every time a
new maximum is reached, the old saved trace is discarded and the
new trace is saved.

To reset the maximum, echo 0 into tracing_max_latency. Here is
an example:

 # echo irqsoff > current_tracer
 # echo 0 > tracing_max_latency
 # ls -ltr
 [...]
 # cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 88 us, #4/4, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: sirq-timer/3-48 (uid:0 nice:0 policy:1 rt_prio:49)
#    -----------------
#  => started at: save_args
#  => ended at:   run_ksoftirqd
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
   <...>-3571    3d....    0us+: trace_hardirqs_off_thunk <-save_args
sirq-tim-48      3d....   87us : schedule <-run_ksoftirqd
sirq-tim-48      3d....   89us : trace_hardirqs_on <-run_ksoftirqd
sirq-tim-48      3d....   90us : <stack trace>
 => schedule
 => run_ksoftirqd
 => kthread
 => kernel_thread_helper

Here we see that that we had a latency of 88 microsecs.
The trace_hardirqs_off_thunk is a helper routine in the assembly
code of save_args that disabled interrupts. The difference
between the 88 and the displayed timestamp 90us occurred because the
clock was incremented between the time of recording the max latency
and the time of recording the function that had that latency.

At the end of the trace, a stack dump is given to help find the
```

```
full call graph of the location that had the irqsoff latency.

Note the above example had ftrace_enabled not set. If we set the
ftrace_enabled, we get a much larger output:

# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 300 us, #301/301, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: sirq-sched/3-54 (uid:0 nice:0 policy:1 rt_prio:49)
#    -----------------
#  => started at: save_args
#  => ended at:   run_ksoftirqd
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
   <...>-3593    3d....    1us : trace_hardirqs_off_thunk <-save_args
   <...>-3593    3d....    2us : smp_apic_timer_interrupt <-apic_timer_interrupt
   <...>-3593    3d....    3us : ack_APIC_irq <-smp_apic_timer_interrupt
   <...>-3593    3d....    4us : apic_write <-ack_APIC_irq
   <...>-3593    3d....    5us : native_apic_mem_write <-apic_write
   <...>-3593    3d....    6us : exit_idle <-smp_apic_timer_interrupt
   <...>-3593    3d....    7us : irq_enter <-smp_apic_timer_interrupt
   <...>-3593    3d....    8us : rcu_irq_enter <-irq_enter
   <...>-3593    3d....    9us : idle_cpu <-irq_enter
   <...>-3593    3d.h..   10us : hrtimer_interrupt <-smp_apic_timer_interrupt
   <...>-3593    3d.h..   11us+: ktime_get <-hrtimer_interrupt
   <...>-3593    3d.h..   13us : timekeeping_get_ns <-ktime_get
   <...>-3593    3d.h..   13us : ktime_set <-ktime_get
   <...>-3593    3d.h..   15us : _raw_spin_lock <-hrtimer_interrupt
[...]
   <...>-3593    3d..2.  286us : account_group_exec_runtime <-update_curr
   <...>-3593    3d..2.  287us : check_spread.clone.63 <-put_prev_task_fair
   <...>-3593    3d..2.  288us : __enqueue_entity <-put_prev_task_fair
   <...>-3593    3d..2.  288us : pick_next_task <-__schedule
   <...>-3593    3d..2.  289us : pick_next_task_rt <-pick_next_task
   <...>-3593    3d..2.  290us : sched_find_first_bit <-pick_next_task_rt
   <...>-3593    3d..2.  291us : sched_info_queued <-__schedule
   <...>-3593    3d..2.  292us : atomic_inc <-__schedule
   <...>-3593    3d..2.  293us : enter_lazy_tlb.clone.15 <-__schedule
   <...>-3593    3d..2.  294us : native_load_tls <-__switch_to
   <...>-3593    3d..2.  295us+: __unlazy_fpu <-__switch_to
sirq-sch-54      3d..2.  296us : finish_task_switch <-__schedule
sirq-sch-54      3d..2.  297us : _raw_spin_unlock <-finish_task_switch
sirq-sch-54      3d..1.  298us : test_ti_thread_flag.clone.2 <-_raw_spin_unlock
sirq-sch-54      3d....  299us : post_schedule <-__schedule
sirq-sch-54      3d....  300us : schedule <-run_ksoftirqd
sirq-sch-54      3d....  301us : trace_hardirqs_on <-run_ksoftirqd
sirq-sch-54      3d....  302us : <stack trace>
 => schedule
```

```
 => run_ksoftirqd
 => kthread
 => kernel_thread_helper


Here we traced a 300 microsecond latency. But we also see all the
functions that were called during that time. Note that by
enabling function tracing, we incur an added overhead. This
overhead causes the latency times to be greatly exaggerated.
The previous largest time was 88us has grown to 300us for the
same latency. But nevertheless, this trace has provided some very
helpful debugging information.

If the option "display-graph" is enabled, the following output would appear.

# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.2.16-test-rt27
# --------------------------------------------------------------------
# latency: 122 us, #259/259, CPU#3 | (Mreempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: swapper/3-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: __schedule
#  => ended at:   return_to_handler
#
#
#                                      _-----=> irqs-off
#                                     / _----=> need-resched
#                                    | / _---=> hardirq/softirq
#                                    || / _--=> preempt-depth
#                                    ||| /
#     TIME         CPU  TASK/PID       ||||  DURATION                  FUNCTION
CALLS
#      |           |     |    |        ||||   |   |                     |   |   |
|
  145.917252 |   3)   ksoftir-20   |  d..1.  0.000 us    |  _raw_spin_lock_irq();
  145.917252 |   3)   ksoftir-20   |  d..1.  0.084 us    |  add_preempt_count();
  145.917253 |   3)   ksoftir-20   |  d..2.  0.097 us    |  do_raw_spin_lock();
  145.917253 |   3)   ksoftir-20   |  d..2.              |  signal_pending_state()
{
  145.917254 |   3)   ksoftir-20   |  d..2.  0.078 us    |
test_ti_thread_flag();
  145.917254 |   3)   ksoftir-20   |  d..2.  0.644 us    |  }
  145.917254 |   3)   ksoftir-20   |  d..2.              |  deactivate_task() {
  145.917255 |   3)   ksoftir-20   |  d..2.              |    dequeue_task() {
  145.917255 |   3)   ksoftir-20   |  d..2.  0.142 us    |      update_rq_clock();
  145.917256 |   3)   ksoftir-20   |  d..2.              |      dequeue_task_rt()
{
  145.917256 |   3)   ksoftir-20   |  d..2.              |        update_curr_rt()
{
  145.917256 |   3)   ksoftir-20   |  d..2.  0.085 us    |
account_group_exec_runtime();
  145.917257 |   3)   ksoftir-20   |  d..2.              |
cpuacct_charge() {
  145.917257 |   3)   ksoftir-20   |  d..2.  0.069 us    |
__rcu_read_lock();
  145.917258 |   3)   ksoftir-20   |  d..2.  0.066 us    |
__rcu_read_unlock();
  145.917258 |   3)   ksoftir-20   |  d..2.  1.140 us    |          }
```

```
[...]

  145.917371 |   3)   ksoftir-20   |  d..2.  0.087 us    |  atomic_inc();
  145.917372 |   3)   ksoftir-20   |  d..2.  0.072 us    |  native_load_tls();
 ----------------------------------------
 3)   ksoftir-20   =>    <idle>0
 ----------------------------------------

  145.917373 |   3)    <idle>0    |  d..2.              |  finish_task_switch() {
  145.917373 |   3)    <idle>0    |  d..2.              |
_raw_spin_unlock_irq() {
  145.917373 |   3)    <idle>0    |  d..2.  0.000 us    |  _raw_spin_unlock_irq();
  145.917374 |   3)    <idle>0    |  d..2.  0.000 us    |  trace_hardirqs_on();
  <idle>0       3d..2.  158us : <stack trace>
 => trace_hardirqs_on
 => _raw_spin_unlock_irq
 => return_to_handler
 => __schedule
 => return_to_handler
 => schedule
 => schedule_preempt_disabled
 => cpu_idle
 => start_secondary


preemptoff
----------

When preemption is disabled, we may be able to receive
interrupts but the task cannot be preempted and a higher
priority task must wait for preemption to be enabled again
before it can preempt a lower priority task.

The preemptoff tracer traces the places that disable preemption.
Like the irqsoff tracer, it records the maximum latency for
which preemption was disabled. The control of preemptoff tracer
is much like the irqsoff tracer.

 # echo preemptoff > current_tracer
 # echo 0 > tracing_max_latency
 # ls -ltr
 [...]
 # cat trace
# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 28 us, #4/4, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: irqbalance-1460 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: smp_apic_timer_interrupt
#  => ended at:   smp_apic_timer_interrupt
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
```

```
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
irqbalan-1460    0d.h..    1us+: irq_enter <-smp_apic_timer_interrupt
irqbalan-1460    0dN.1.   28us : irq_exit <-smp_apic_timer_interrupt
irqbalan-1460    0dN.1.   29us : trace_preempt_on <-smp_apic_timer_interrupt
irqbalan-1460    0dN.1.   29us : <stack trace>
 => sub_preempt_count
 => irq_exit
 => smp_apic_timer_interrupt
 => apic_timer_interrupt
 => show_stat
 => seq_read
 => proc_reg_read
 => vfs_read

This has some more changes. Preemption was disabled when an
interrupt came in (notice the 'h'), and was enabled when returning
from the softirq. The 'N' flag tells us that the NEED_RESCHED flag
of the task has been set. We also see that interrupts
have been disabled when entering the preempt off section and
leaving it (the 'd'). We do not know if interrupts were enabled
in the mean time.

# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 148 us, #167/167, CPU#2 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: bash-2040 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: default_wake_function
#  => ended at:   default_wake_function
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
    bash-2040    2...1.    0us+: try_to_wake_up <-default_wake_function
    bash-2040    2d..1.    2us : _raw_spin_lock <-task_rq_lock
    bash-2040    2d..2.    3us : do_raw_spin_lock <-_raw_spin_lock
    bash-2040    2d..2.    3us : update_rq_clock <-try_to_wake_up
    bash-2040    2d..2.    5us : task_waking_fair <-try_to_wake_up
    bash-2040    2d..2.    6us : cfs_rq_of <-task_waking_fair
    bash-2040    2d..2.    6us : select_task_rq <-try_to_wake_up
    bash-2040    2d..2.    7us : select_task_rq_fair <-select_task_rq
    bash-2040    2d..2.    8us : _raw_spin_unlock <-select_task_rq_fair
    bash-2040    2d..1.    9us : test_ti_thread_flag.clone.2 <-_raw_spin_unlock
    bash-2040    2d..1.   10us : _raw_spin_lock <-select_task_rq_fair
[...]
    bash-2040    2d..2.   53us : resched_task <-check_preempt_curr_idle
```

```
    bash-2040    2d..2.   53us : test_ti_thread_flag <-resched_task
    bash-2040    2d..2.   54us : set_tsk_need_resched <-resched_task
    bash-2040    2d..2.   55us : _raw_spin_unlock_irqrestore <-try_to_wake_up
    bash-2040    2d..2.   57us : smp_apic_timer_interrupt <-apic_timer_interrupt
    bash-2040    2d..2.   57us : ack_APIC_irq <-smp_apic_timer_interrupt
    bash-2040    2d..2.   58us : apic_write <-ack_APIC_irq
    bash-2040    2d..2.   59us : native_apic_mem_write <-apic_write
    bash-2040    2d..2.   59us : exit_idle <-smp_apic_timer_interrupt
    bash-2040    2d..2.   60us : irq_enter <-smp_apic_timer_interrupt
    bash-2040    2d..2.   61us : rcu_irq_enter <-irq_enter
    bash-2040    2d..2.   61us : idle_cpu <-irq_enter
    bash-2040    2d.h2.   62us : hrtimer_interrupt <-smp_apic_timer_interrupt
[...]
    bash-2040    2dNh2.  142us : apic_write <-lapic_next_event
    bash-2040    2dNh2.  143us : native_apic_mem_write <-apic_write
    bash-2040    2dNh2.  143us : irq_exit <-smp_apic_timer_interrupt
    bash-2040    2dN.3.  144us : do_softirq <-irq_exit
    bash-2040    2dN.3.  145us : __do_softirq <-call_softirq
    bash-2040    2dN.3.  145us : trigger_softirqs <-__do_softirq
    bash-2040    2dN.3.  146us : wakeup_softirqd <-trigger_softirqs
    bash-2040    2dN.3.  146us : rcu_irq_exit <-irq_exit
    bash-2040    2dN.3.  147us : idle_cpu <-irq_exit
    bash-2040    2.N.1.  148us : try_to_wake_up <-default_wake_function
    bash-2040    2.N.1.  149us : trace_preempt_on <-default_wake_function
    bash-2040    2.N.1.  150us : <stack trace>
 => sub_preempt_count
 => try_to_wake_up
 => default_wake_function
 => autoremove_wake_function
 => __wake_up_common
 => __wake_up_sync_key
 => __wake_up_sync
 => pipe_release

The above is an example of the preemptoff trace with
ftrace_enabled set. Here we see that interrupts were enabled just
before preemption was enabled. Also, interrupts were enabled
at the _raw_spin_unlock_irqrestore() call, and at that moment
the timer interrupt (apic_timer_interrupt) came in. But because
no function was traced between those two events, the 'd' flag
was never shown to be off there.

The irq_enter code lets us know that we entered an interrupt 'h'.
Before that, the functions being traced still show that it is not
in an interrupt, but we can see from the functions themselves that
this is not the case.


preemptirqsoff
--------------

Knowing the locations that have interrupts disabled or
preemption disabled for the longest times is helpful. But
sometimes we would like to know when either preemption and/or
interrupts are disabled.

Consider the following code:

    local_irq_disable();
    call_function_with_irqs_off();
```

```
    preempt_disable();
    call_function_with_irqs_and_preemption_off();
    local_irq_enable();
    call_function_with_preemption_off();
    preempt_enable();

The irqsoff tracer will record the total length of
call_function_with_irqs_off() and
call_function_with_irqs_and_preemption_off().

The preemptoff tracer will record the total length of
call_function_with_irqs_and_preemption_off() and
call_function_with_preemption_off().

But neither will trace the time that interrupts and/or
preemption is disabled. This total time is the time that we can
not schedule. To record this time, use the preemptirqsoff
tracer.

Again, using this trace is much like the irqsoff and preemptoff
tracers.

 # echo preemptirqsoff > current_tracer
 # echo 0 > tracing_max_latency
 # ls -ltr
 [...]
 # cat trace
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 52 us, #4/4, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: hackbench-11587 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: rt_spin_lock_slowunlock
#  => ended at:   rt_spin_lock_slowunlock
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
hackbenc-11587   0d....    0us+: _raw_spin_lock_irqsave <-rt_spin_lock_slowunlock
hackbenc-11587   0.N.1.   52us : _raw_spin_unlock_irqrestore <-
rt_spin_lock_slowunlock
hackbenc-11587   0.N.1.   53us : trace_preempt_on <-rt_spin_lock_slowunlock
hackbenc-11587   0.N.1.   54us : <stack trace>
 => sub_preempt_count
 => _raw_spin_unlock_irqrestore
 => rt_spin_lock_slowunlock
 => rt_spin_lock_fastunlock.clone.13
 => rt_spin_unlock
 => slab_irq_enable
 => kfree
```

```
 => skb_release_data

Interrupts and preemption was disabled at the _raw_spin_lock_irqsave.
Although the interrupts are shown disabled and the preemption was not,
is just the placement of where the recording takes place (it happens
after interrupts were disabled, and before the preemption was disabled).
Both interrupts and preemption is re-enabled at the rt_spin_lock_slowunlock.
This time due to the placement of the disabling, the interrupts are
shown enabled while preemption is still disabled.

Here is a trace with ftrace_enabled set:

# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 350 us, #457/457, CPU#2 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: hackbench-4755 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: rt_spin_lock_slowunlock
#  => ended at:   rt_spin_lock_slowunlock
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
hackbenc-4755    2d....    1us : _raw_spin_lock_irqsave <-rt_spin_lock_slowunlock
hackbenc-4755    2d..1.    2us : wakeup_next_waiter <-rt_spin_lock_slowunlock
hackbenc-4755    2d..1.    3us : rt_mutex_top_waiter <-wakeup_next_waiter
hackbenc-4755    2d..1.    3us : _raw_spin_lock <-wakeup_next_waiter
hackbenc-4755    2d..2.    4us : do_raw_spin_lock <-_raw_spin_lock
hackbenc-4755    2d..2.    5us : _raw_spin_unlock <-wakeup_next_waiter
hackbenc-4755    2d..1.    6us : test_ti_thread_flag.clone.2 <-_raw_spin_unlock
hackbenc-4755    2d..1.    6us : wake_up_process_mutex <-wakeup_next_waiter
hackbenc-4755    2d..1.    7us : try_to_wake_up <-wake_up_process_mutex
hackbenc-4755    2d..2.    8us : task_rq_lock <-try_to_wake_up
hackbenc-4755    2d..2.    8us : __raw_local_irq_save <-task_rq_lock
hackbenc-4755    2d..2.    9us : __raw_local_save_flags <-__raw_local_irq_save
[...]
hackbenc-4755    2d..3.   20us : wakeup_preempt_entity <-check_preempt_wakeup
hackbenc-4755    2d..3.   21us : _raw_spin_unlock_irqrestore <-try_to_wake_up
hackbenc-4755    2d..2.   21us : test_ti_thread_flag.clone.2 <-
_raw_spin_unlock_irqrestore
hackbenc-4755    2d..1.   22us : test_ti_thread_flag <-try_to_wake_up
hackbenc-4755    2d..1.   23us+: _raw_spin_unlock_irqrestore <-
rt_spin_lock_slowunlock
hackbenc-4755    2d..1.   25us : do_IRQ <-ret_from_intr
hackbenc-4755    2d..1.   26us : exit_idle <-do_IRQ
hackbenc-4755    2d..1.   26us : irq_enter <-do_IRQ
hackbenc-4755    2d..1.   27us : rcu_irq_enter <-irq_enter
hackbenc-4755    2d..1.   28us : idle_cpu <-irq_enter
hackbenc-4755    2d.h1.   29us : handle_irq <-do_IRQ
hackbenc-4755    2d.h1.   30us : irq_to_desc <-handle_irq
```

```
[...]
hackbenc-4755    2dNh1.  271us : timekeeping_get_ns <-ktime_get
hackbenc-4755    2dNh1.  272us : clockevents_program_event <-
tick_dev_program_event
hackbenc-4755    2dNh1.  272us : lapic_next_event <-clockevents_program_event
hackbenc-4755    2dNh1.  273us : apic_write <-lapic_next_event
hackbenc-4755    2dNh1.  274us : native_apic_mem_write <-apic_write
hackbenc-4755    2dNh1.  274us : irq_exit <-smp_apic_timer_interrupt
hackbenc-4755    2dN.2.  275us : do_softirq <-irq_exit
hackbenc-4755    2dN.2.  276us : __do_softirq <-call_softirq
hackbenc-4755    2dN.2.  277us : trigger_softirqs <-__do_softirq
hackbenc-4755    2dN.2.  277us : wakeup_softirqd <-trigger_softirqs
hackbenc-4755    2dN.2.  278us : rcu_irq_exit <-irq_exit
hackbenc-4755    2dN.2.  279us+: idle_cpu <-irq_exit
[...]
hackbenc-4755    2dNh1.  343us : irq_exit <-do_IRQ
hackbenc-4755    2dN.2.  344us : do_softirq <-irq_exit
hackbenc-4755    2dN.2.  345us : __do_softirq <-call_softirq
hackbenc-4755    2dN.2.  346us : trigger_softirqs <-__do_softirq
hackbenc-4755    2dN.2.  346us : wakeup_softirqd <-trigger_softirqs
hackbenc-4755    2dN.2.  348us : rcu_irq_exit <-irq_exit
hackbenc-4755    2dN.2.  349us : idle_cpu <-irq_exit
hackbenc-4755    2.N.1.  350us : _raw_spin_unlock_irqrestore <-
rt_spin_lock_slowunlock
hackbenc-4755    2.N.1.  351us : trace_preempt_on <-rt_spin_lock_slowunlock
hackbenc-4755    2.N.1.  352us : <stack trace>
 => sub_preempt_count
 => _raw_spin_unlock_irqrestore
 => rt_spin_lock_slowunlock
 => rt_spin_lock_fastunlock.clone.13
 => rt_spin_unlock
 => slab_irq_enable
 => kmem_cache_alloc_node
 => __alloc_skb


This is a very interesting trace. It started again with the irq disabling
of _raw_spin_lock_irqsave which also disabled preemption later.
But we can also see here that when it enabled interrupts before
disabling preemption, the time interrupt triggered. As the interrupt exited,
it enabled softirqs. Finally when the interrupt returned, the
_raw_spin_unlock_irqrestore was able to disable preemption. If we
did not have the function tracer running, we would not have noticed
that an interrupt arrived. (But we can if we enabled events,
see events.txt for more info.)


wakeup and wakeup_rt
--------------------

In a Real-Time environment it is very important to know the
wakeup time it takes for the highest priority task that is woken
up to the time that it executes. This is also known as "schedule
latency".

Real-Time environments are interested in the worst case latency.
That is the longest latency it takes for something to happen,
and not the average. We can have a very fast scheduler that may
only have a large latency once in a while, but that would not
work well with Real-Time tasks. The wakeup_rt tracer was designed
```

```
to record the worst case wakeups of RT tasks. Non-RT tasks are
not recorded because the tracer only records one worst case and
tracing non-RT tasks that are unpredictable will overwrite the
worst case latency of RT tasks. If you are still interested in non-RT
tasks, then use the wakeup tracer.

Since the wakeup_rt tracer only deals with RT tasks, we will run this
slightly differently than we did with the previous tracers.
Instead of performing an 'ls', we will run 'sleep 1' under
'chrt' which changes the priority of the task.

 # echo wakeup_rt > current_tracer
 # echo 0 > tracing_max_latency
 # chrt -f 90 sleep 1
 # cat trace
# tracer: wakeup_rt
#
# wakeup_rt latency trace v1.1.5 on 3.2.16-test-rt27
# --------------------------------------------------------------------
# latency: 5 us, #4/4, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: sleep-2903 (uid:0 nice:0 policy:1 rt_prio:90)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> migrate-disable
#                ||||| /     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>0       3d.h4.    0us :      0:120:R   + [003]  2903: 49:R sleep
  <idle>0       3d.h4.    0us+: ttwu_do_activate.constprop.175 <try_to_wake_up
  <idle>0       3d..3.    6us : __schedule <schedule
  <idle>0       3d..3.    6us :      0:120:R ==> [003]  2903: 9:R sleep

Running this on an idle system, we see that it took 5
microseconds to perform the task switch.

The header shows the task PID of 2903 that was recorded.
The rt_prio is 90 which is the user space priority. The prio shown
in the wake up and schedule events is 9 which is the kernel version
of that priority.  The policy is 1 for SCHED_FIFO and 2 for SCHED_RR.

Remember that the KERNEL-PRIO is the inverse of the actual
priority with zero (0) being the highest priority and the nice
values starting at 100 (nice -20). Below is a quick chart to map
the kernel priority to user land priorities.

   Kernel Space                     User Space
 ===============================================================
   0(high) to  98(low)     user RT priority 99(high) to 1(low)
                            with SCHED_RR or SCHED_FIFO
 ---------------------------------------------------------------
  99                        sched_priority is not used in scheduling
                            decisions(it must be specified as 0)
 ---------------------------------------------------------------
 100(high) to 139(low)     user nice -20(high) to 19(low)
```

```
 -----------------------------------------------------------------

The task states, like in the final event:

 R - running : wants to run, may not actually be running
 S - sleep   : process is waiting to be woken up (handles signals)
 D - disk sleep (uninterruptible sleep) : process must be woken up
                                          (ignores signals)
 T - stopped : process suspended
 t - traced  : process is being traced (with something like gdb)
 Z - zombie  : process waiting to be cleaned up
 X - unknown


Doing the same with chrt -r 90 and ftrace_enabled set.

# tracer: wakeup
#
# wakeup latency trace v1.1.5 on 3.2.16-test
# --------------------------------------------------------------------
# latency: 95 us, #179/179, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: -6 (uid:0 nice:0 policy:1 rt_prio:99)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
   sleep-2461    1d..3.    1us+:   2461:120:R   + [001]     6:  0:R migration/1
   sleep-2461    1d..3.    5us : wake_up_process <-sched_exec
   sleep-2461    1d..2.    6us : test_ti_thread_flag <-
rcu_read_unlock_sched_notrace
   sleep-2461    1d..2.    6us : check_preempt_curr <-try_to_wake_up
   sleep-2461    1d..2.    7us : check_preempt_wakeup <-check_preempt_curr
   sleep-2461    1d..2.    7us : resched_task <-check_preempt_wakeup
   sleep-2461    1d..2.    8us : test_ti_thread_flag <-resched_task
   sleep-2461    1d..2.    8us : set_tsk_need_resched <-resched_task
   sleep-2461    1dN.2.    9us : task_woken_rt <-try_to_wake_up
   sleep-2461    1dN.2.    9us : test_ti_thread_flag <-task_woken_rt
   sleep-2461    1dN.2.   10us : _raw_spin_unlock_irqrestore <-try_to_wake_up
   sleep-2461    1dN.2.   11us : smp_apic_timer_interrupt <-apic_timer_interrupt
   sleep-2461    1dN.2.   11us : ack_APIC_irq <-smp_apic_timer_interrupt
   sleep-2461    1dN.2.   12us : apic_write <-ack_APIC_irq
   sleep-2461    1dN.2.   12us : native_apic_mem_write <-apic_write
   sleep-2461    1dN.2.   13us : exit_idle <-smp_apic_timer_interrupt
   sleep-2461    1dN.2.   13us : irq_enter <-smp_apic_timer_interrupt
   sleep-2461    1dN.2.   14us : rcu_irq_enter <-irq_enter
   sleep-2461    1dN.2.   14us : idle_cpu <-irq_enter
   sleep-2461    1dNh2.   15us : hrtimer_interrupt <-smp_apic_timer_interrupt
[...]
   sleep-2461    1dNh2.   56us : native_apic_mem_write <-apic_write
   sleep-2461    1dNh2.   57us : irq_exit <-smp_apic_timer_interrupt
   sleep-2461    1dN.3.   57us : do_softirq <-irq_exit
   sleep-2461    1dN.3.   58us : __do_softirq <-call_softirq
```

```
   sleep-2461    1dN.3.   58us : __local_bh_disable <-__do_softirq
   sleep-2461    1dN.3.   58us : __raw_local_irq_save <-__local_bh_disable
   sleep-2461    1dN.3.   59us : __raw_local_save_flags <-__raw_local_irq_save
   sleep-2461    1.Ns3.   60us : run_timer_softirq <-__do_softirq
   sleep-2461    1.Ns3.   61us : hrtimer_run_pending <-run_timer_softirq
[...]
   sleep-2461    1.Ns3.   80us : rcu_bh_qs <-__do_softirq
   sleep-2461    1dNs3.   80us : _local_bh_enable <-__do_softirq
   sleep-2461    1dNs3.   80us : __raw_local_save_flags <-_local_bh_enable
   sleep-2461    1dN.3.   81us : rcu_irq_exit <-irq_exit
   sleep-2461    1dN.3.   82us : idle_cpu <-irq_exit
   sleep-2461    1.N.1.   82us : test_ti_thread_flag.clone.2 <-
_raw_spin_unlock_irqrestore
   sleep-2461    1.N.1.   83us : preempt_schedule <-_raw_spin_unlock_irqrestore
   sleep-2461    1.N...   83us : test_ti_thread_flag <-try_to_wake_up
   sleep-2461    1.N...   84us : preempt_schedule <-try_to_wake_up
   sleep-2461    1.N...   84us : __raw_local_save_flags <-preempt_schedule
   sleep-2461    1.N...   85us : schedule <-preempt_schedule
   sleep-2461    1.N.1.   85us : rcu_sched_qs <-schedule
[...]
   sleep-2461    1d..2.   92us : pick_next_task_rt <-pick_next_task
   sleep-2461    1d..2.   93us : sched_find_first_bit <-pick_next_task_rt
   sleep-2461    1d..3.   94us : schedule <-preempt_schedule
   sleep-2461    1d..3.   95us :   2461:120:R ==> [001]     6:  0:R migration/1

This time instead of tracing the wakeup of our sleep task, the trace
captured the migration task. It may have caught the sleep task, but
then the migration task took longer to wake up, and only the maximum
trace is stored. Shortly after the migration thread was worken up on
the same CPU our sleep task was running "[001]", the sleep task
need resched flag was set ("N"). After "_raw_spin_unlock_irqrestore()"
enabled interrupts, a timer interrupt triggered (also disabling
interrupts leaving the 'd' set). The irq_entry() has a hook to cause
the 'h' flag to be set to show that the event happened in interrupt
context. The timer interrupt queued the timer softirq and then started
executing that. The 's' flag shows the softirqs are disabled or is
running. Finally, the softirq returns back to the original place
the code was interrupted and the migration thread is scheduled.

function
--------

This tracer is the function tracer. Enabling the function tracer
can be done from the debug file system. Make sure the
ftrace_enabled is set; otherwise this tracer is a nop. On boot up
the ftrace_enabled sysctl is set, but the bootup scripts or a user
could have cleared it.

 # sysctl kernel.ftrace_enabled=1
 # echo function > current_tracer
 # usleep 1
 # cat trace
 # echo 0 > tracing_on
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
          <idle>-0     [003] 15774.015440: test_ti_thread_flag <-cpu_idle
          <idle>-0     [003] 15774.015441: enter_idle <-cpu_idle
          <idle>-0     [003] 15774.015442: mwait_idle <-cpu_idle
```

```
          <idle>-0     [003] 15774.015442: need_resched <-mwait_idle
          <idle>-0     [003] 15774.015443: test_ti_thread_flag <-need_resched
          <idle>-0     [003] 15774.015444: trace_power_start.clone.5 <-mwait_idle
          <idle>-0     [003] 15774.015445: need_resched <-mwait_idle
          <idle>-0     [003] 15774.015446: test_ti_thread_flag <-need_resched
          <idle>-0     [003] 15774.015447: __exit_idle <-cpu_idle
          <idle>-0     [003] 15774.015448: test_ti_thread_flag <-cpu_idle
          <idle>-0     [003] 15774.015449: enter_idle <-cpu_idle
          <idle>-0     [003] 15774.015450: mwait_idle <-cpu_idle
          <idle>-0     [003] 15774.015450: need_resched <-mwait_idle
          <idle>-0     [003] 15774.015451: test_ti_thread_flag <-need_resched
          <idle>-0     [003] 15774.015452: trace_power_start.clone.5 <-mwait_idle
          <idle>-0     [003] 15774.015453: need_resched <-mwait_idle
          <idle>-0     [003] 15774.015454: test_ti_thread_flag <-need_resched
          <idle>-0     [003] 15774.015455: __exit_idle <-cpu_idle
[...]


Note: function tracer uses ring buffers to store the above
entries. The newest data may overwrite the oldest data (unless the
overwrite option is off) Sometimes using echo to stop the trace is not
sufficient because the tracing could have overwritten the data that
you wanted to record. For this reason, it is sometimes better to disable
tracing directly from a program. This allows you to stop the
tracing at the point that you hit the part that you are
interested in. To disable the tracing directly from a C program,
something like following code snippet can be used:

int trace_fd;
[...]
int main(int argc, char *argv[]) {
 [...]
 trace_fd = open(tracing_file("tracing_on"), O_WRONLY);
 [...]
 if (condition_hit()) {
  write(trace_fd, "0", 1);
 }
 [...]
}


Single thread tracing
---------------------

By writing into set_ftrace_pid you can trace a
single thread. For example:

# cat set_ftrace_pid
no pid
# echo 3111 > set_ftrace_pid
# cat set_ftrace_pid
3111
# echo function > current_tracer
# cat trace | head
 # tracer: function
 #
 #           TASK-PID    CPU#    TIMESTAMP  FUNCTION
 #              | |       |          |         |
     yum-updatesd-3111  [003]  1637.254676: finish_task_switch <-thread_return
     yum-updatesd-3111  [003]  1637.254681: hrtimer_cancel <-
```

```
schedule_hrtimeout_range
     yum-updatesd-3111  [003]  1637.254682: hrtimer_try_to_cancel <-hrtimer_cancel
     yum-updatesd-3111  [003]  1637.254683: lock_hrtimer_base <-
hrtimer_try_to_cancel
     yum-updatesd-3111  [003]  1637.254685: fget_light <-do_sys_poll
     yum-updatesd-3111  [003]  1637.254686: pipe_poll <-do_sys_poll
# echo -1 > set_ftrace_pid
# cat trace |head
 # tracer: function
 #
 #            TASK-PID    CPU#    TIMESTAMP  FUNCTION
 #               | |       |          |         |
 ##### CPU 3 buffer started ####
     yum-updatesd-3111  [003]  1701.957688: free_poll_entry <-poll_freewait
     yum-updatesd-3111  [003]  1701.957689: remove_wait_queue <-free_poll_entry
     yum-updatesd-3111  [003]  1701.957691: fput <-free_poll_entry
     yum-updatesd-3111  [003]  1701.957692: audit_syscall_exit <-sysret_audit
     yum-updatesd-3111  [003]  1701.957693: path_put <-audit_syscall_exit

If you want to trace a function when executing, you could use a
simple shell script:

   --------
#!/bin/bash
echo $$ > /sys/kernel/debug/tracing/set_ftrace_pid
echo function > /sys/kernel/debug/tracing/current_tracer
exec $*
something like this simple program:
   --------

Then just run the script followed by a program and its arguments.

For including this in a C program:

   --------
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define _STR(x) #x
#define STR(x) _STR(x)
#define MAX_PATH 256

const char *find_debugfs(void)
{
       static char debugfs[MAX_PATH+1];
       static int debugfs_found;
       char type[100];
       FILE *fp;

       if (debugfs_found)
               return debugfs;

       if ((fp = fopen("/proc/mounts","r")) == NULL) {
               perror("/proc/mounts");
               return NULL;
```

```
        }

        while (fscanf(fp, "%*s %"
                      STR(MAX_PATH)
                      "s %99s %*s %*d %*d\n",
                      debugfs, type) == 2) {
                if (strcmp(type, "debugfs") == 0)
                        break;
        }
        fclose(fp);

        if (strcmp(type, "debugfs") != 0) {
                fprintf(stderr, "debugfs not mounted");
                return NULL;
        }

        strcat(debugfs, "/tracing/");
        debugfs_found = 1;

        return debugfs;
}

const char *tracing_file(const char *file_name)
{
        static char trace_file[MAX_PATH+1];
        snprintf(trace_file, MAX_PATH, "%s/%s", find_debugfs(), file_name);
        return trace_file;
}

int main (int argc, char **argv)
{
         if (argc < 1)
                 exit(-1);

         if (fork() > 0) {
                 int fd, ffd;
                 char line[64];
                 int s;

                 ffd = open(tracing_file("current_tracer"), O_WRONLY);
                 if (ffd < 0)
                         exit(-1);
                 write(ffd, "nop", 3);

                 fd = open(tracing_file("set_ftrace_pid"), O_WRONLY);
                 s = sprintf(line, "%d\n", getpid());
                 write(fd, line, s);

                 write(ffd, "function", 8);

                 close(fd);
                 close(ffd);

                 execvp(argv[1], argv+1);
         }

         return 0;
}
   --------
```

```
function graph tracer
---------------------------

This tracer is similar to the function tracer except that it
probes a function on its entry and its exit. This is done by
using a dynamically allocated stack of return addresses in each
task_struct. On function entry the tracer overwrites the return
address of each function traced to set a custom probe. Thus the
original return address is stored on the stack of return address
in the task_struct.

Probing on both ends of a function leads to special features
such as:

- measure of a function's time execution
- having a reliable call stack to draw function calls graph

This tracer is useful in several situations:

- you want to find the reason of a strange kernel behavior and
  need to see what happens in detail on any areas (or specific
  ones).

- you are experiencing weird latencies but it's difficult to
  find its origin.

- you want to find quickly which path is taken by a specific
  function

- you just want to peek inside a working kernel and want to see
  what happens there.

# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |

 0)               |  sys_open() {
 0)               |    do_sys_open() {
 0)               |      getname() {
 0)               |        kmem_cache_alloc() {
 0)   1.382 us    |          __might_sleep();
 0)   2.478 us    |        }
 0)               |        strncpy_from_user() {
 0)               |          might_fault() {
 0)   1.389 us    |            __might_sleep();
 0)   2.553 us    |          }
 0)   3.807 us    |        }
 0)   7.876 us    |      }
 0)               |      alloc_fd() {
 0)   0.668 us    |        _spin_lock();
 0)   0.570 us    |        expand_files();
 0)   0.586 us    |        _spin_unlock();


There are several columns that can be dynamically
enabled/disabled. You can use every combination of options you
want, depending on your needs.
```

```
- The cpu number on which the function executed is default
  enabled.  It is sometimes better to only trace one cpu (see
  tracing_cpu_mask file) or you might sometimes see unordered
  function calls while cpu tracing switch.

 hide: echo nofuncgraph-cpu > trace_options
 show: echo funcgraph-cpu > trace_options

- The duration (function's time of execution) is displayed on
  the closing bracket line of a function or on the same line
  than the current function in case of a leaf one. It is default
  enabled.

 hide: echo nofuncgraph-duration > trace_options
 show: echo funcgraph-duration > trace_options

- The overhead field precedes the duration field in case of
  reached duration thresholds.

 hide: echo nofuncgraph-overhead > trace_options
 show: echo funcgraph-overhead > trace_options
 depends on: funcgraph-duration

  ie:

  0)               |    up_write() {
  0)   0.646 us    |      _spin_lock_irqsave();
  0)   0.684 us    |      _spin_unlock_irqrestore();
  0)   3.123 us    |    }
  0)   0.548 us    |    fput();
  0) + 58.628 us   |  }

  [...]

  0)               |      putname() {
  0)               |        kmem_cache_free() {
  0)   0.518 us    |          __phys_addr();
  0)   1.757 us    |        }
  0)   2.861 us    |      }
  0) ! 115.305 us  |    }
  0) ! 116.402 us  |  }

  + means that the function exceeded 10 usecs.
  ! means that the function exceeded 100 usecs.


- The task/pid field displays the thread cmdline and pid which
  executed the function. It is default disabled.

 hide: echo nofuncgraph-proc > trace_options
 show: echo funcgraph-proc > trace_options

  ie:
  # tracer: function_graph
  #
  # CPU  TASK/PID        DURATION                  FUNCTION CALLS
  # |    |    |           |   |                     |   |   |   |
  0)    sh-4802     |               |                  d_free() {
  0)    sh-4802     |               |                    call_rcu() {
  0)    sh-4802     |               |                      __call_rcu() {
```

```
  0)    sh-4802     |   0.616 us    |                          rcu_process_gp_end();
  0)    sh-4802     |   0.586 us    |
check_for_new_grace_period();
  0)    sh-4802     |   2.899 us    |                        }
  0)    sh-4802     |   4.040 us    |                      }
  0)    sh-4802     |   5.151 us    |                    }
  0)    sh-4802     | + 49.370 us   |                  }


- The absolute time field is an absolute timestamp given by the
  system clock since it started. A snapshot of this time is
  given on each entry/exit of functions

 hide: echo nofuncgraph-abstime > trace_options
 show: echo funcgraph-abstime > trace_options

  ie:

  #
  #      TIME        CPU  DURATION                  FUNCTION CALLS
  #       |          |     |   |                     |   |   |   |
  360.774522 |   1)   0.541 us    |                                      }
  360.774522 |   1)   4.663 us    |                                    }
  360.774523 |   1)   0.541 us    |
__wake_up_bit();
  360.774524 |   1)   6.796 us    |                                  }
  360.774524 |   1)   7.952 us    |                                }
  360.774525 |   1)   9.063 us    |                              }
  360.774525 |   1)   0.615 us    |
journal_mark_dirty();
  360.774527 |   1)   0.578 us    |                              __brelse();
  360.774528 |   1)               |
reiserfs_prepare_for_journal() {
  360.774528 |   1)               |
unlock_buffer() {
  360.774529 |   1)               |
wake_up_bit() {
  360.774529 |   1)               |
bit_waitqueue() {
  360.774530 |   1)   0.594 us    |
__phys_addr();


You can put some comments on specific functions by using
trace_printk() For example, if you want to put a comment inside
the __might_sleep() function, you just have to call trace_printk()
inside __might_sleep()

trace_printk("I'm a comment!\n")

will produce:

 1)               |             __might_sleep() {
 1)               |                /* I'm a comment! */
 1)   1.449 us    |             }


You might find other useful features for this tracer in the
following "dynamic ftrace" section such as tracing only specific
functions or tasks.
```

```
dynamic ftrace
--------------

how it works
------------

(skip this section if you do not care how dynamic ftrace
 is implemented)

If CONFIG_DYNAMIC_FTRACE is set, the system will run with
virtually no overhead when function tracing is disabled. The way
this works is the mcount function call (placed at the start of
every kernel function, produced by the -pg switch in gcc),
starts of pointing to a simple return. (Enabling FTRACE will
include the -pg switch in the compiling of the kernel.)

At compile time every C file object is run through the
the recordmcount program (located in the scripts directory). This
program will parse the ELF data within the object file and
create a new .text section that will hold all the mcount locations.

A new section called "__mcount_loc" is created that holds
references to all the mcount call sites in the .text section.
The final linker will add all these references into a single table

On boot up, before SMP is initialized, the dynamic ftrace code
scans this table and updates all the locations into nops. It
also records the locations, which are added to the
available_filter_functions list.  Modules are processed as they
are loaded and before they are executed.  When a module is
unloaded, it also removes its functions from the ftrace function
list. This is automatic in the module unload code, and the
module author does not need to worry about it.

When tracing is enabled, stop_machine is called to prevent
races with the CPUS executing code being modified (which can
cause the CPU to do undesirable things), and the nops are
patched back to calls. But this time, they do not call mcount
(which is just a function stub). They now call into the ftrace
infrastructure.

One special side-effect to the recording of the functions being
traced is that we can now selectively choose which functions we
wish to trace and which ones we want the mcount calls to remain
as nops.


Picking specific functions to trace
-----------------------------------

Two files are used, one for enabling and one for disabling the
tracing of specified functions. They are:

  set_ftrace_filter

and

  set_ftrace_notrace
```

```
A list of available functions that you can add to these files is
listed in:

   available_filter_functions

 # cat available_filter_functions
put_prev_task_idle
kmem_cache_create
pick_next_task_rt
get_online_cpus
pick_next_task_fair
mutex_lock
[...]

If I am only interested in sys_nanosleep and hrtimer_interrupt:

 # echo sys_nanosleep hrtimer_interrupt > set_ftrace_filter
 # echo function > current_tracer
 # echo 1 > tracing_on
 # usleep 1
 # echo 0 > tracing_on
 # cat trace
# tracer: ftrace
#
#  TASK-PID   CPU#    TIMESTAMP  FUNCTION
#      | |      |          |         |
  <idle>-0     [001] 33979.796281: hrtimer_interrupt <-smp_apic_timer_interrupt
  <idle>-0     [000] 33979.797217: hrtimer_interrupt <-smp_apic_timer_interrupt
  <idle>-0     [000] 33979.804207: hrtimer_interrupt <-smp_apic_timer_interrupt
  usleep-2672  [002] 33979.804330: hrtimer_interrupt <-smp_apic_timer_interrupt
  usleep-2672  [002] 33979.804785: sys_nanosleep <-system_call_fastpath
  <idle>-0     [002] 33979.804841: hrtimer_interrupt <-smp_apic_timer_interrupt

To see which functions are being traced, you can cat the file:

 # cat set_ftrace_filter
hrtimer_interrupt
sys_nanosleep


Perhaps this is not enough. The filters also allow simple wild
cards. Only the following are currently available

  <match>*  - will match functions that begin with <match>
  *<match>  - will match functions that end with <match>
  *<match>* - will match functions that have <match> in it

These are the only wild cards which are supported.

  <match>*<match> will not work.

Note: It is better to use quotes to enclose the wild cards,
      otherwise the shell may expand the parameters into names
      of files in the local directory.

 # echo 'hrtimer_*' > set_ftrace_filter

Produces:

  <idle>-0     [002] 68988.813277: hrtimer_hres_active <-hrtimer_run_pending
```

```
  <idle>-0     [002] 68988.813286: hrtimer_get_next_event <-
get_next_timer_interrupt
  <idle>-0     [002] 68988.813286: hrtimer_hres_active <-hrtimer_get_next_event
  <idle>-0     [002] 68988.813287: hrtimer_start <-tick_nohz_stop_sched_tick
  <idle>-0     [002] 68988.813288: hrtimer_hres_active <-__remove_hrtimer
  <idle>-0     [002] 68988.813288: hrtimer_force_reprogram <-__remove_hrtimer
  <idle>-0     [003] 68988.881182: hrtimer_interrupt <-smp_apic_timer_interrupt
  <idle>-0     [003] 68988.881185: hrtimer_run_queues <-run_local_timers
  <idle>-0     [003] 68988.881186: hrtimer_hres_active <-hrtimer_run_queues
  <idle>-0     [003] 68988.881189: hrtimer_forward <-tick_sched_timer
  <idle>-0     [003] 68988.881190: hrtimer_run_pending <-run_timer_softirq
  <idle>-0     [003] 68988.881191: hrtimer_hres_active <-hrtimer_run_pending


Notice that we lost the sys_nanosleep.

 # cat set_ftrace_filter
hrtimer_restart
hrtimer_start_expires
hrtimer_hres_active
hrtimer_init_sleeper
hrtimer_forward
hrtimer_force_reprogram
hrtimer_get_res
hrtimer_wakeup
hrtimer_init
hrtimer_get_remaining
hrtimer_try_to_cancel
hrtimer_cancel
hrtimer_start
hrtimer_start_range_ns
hrtimer_start_expires
hrtimer_get_next_event
hrtimer_interrupt
hrtimer_peek_ahead_timers
hrtimer_run_pending
hrtimer_run_queues
hrtimer_nanosleep
hrtimer_nanosleep_restart
hrtimer_start_expires.clone.5
hrtimer_forward_now
hrtimer_restart


This is because the '>' and '>>' act just like they do in bash.
To rewrite the filters, use '>'
To append to the filters, use '>>'

To clear out a filter so that all functions will be recorded
again:

 # echo > set_ftrace_filter
 # cat set_ftrace_filter
 #

Again, now we want to append.

 # echo sys_nanosleep > set_ftrace_filter
 # cat set_ftrace_filter
sys_nanosleep
```

```
 # echo 'hrtimer_*' >> set_ftrace_filter
 # cat set_ftrace_filter
hrtimer_restart
hrtimer_start_expires
hrtimer_hres_active
hrtimer_init_sleeper
hrtimer_forward
hrtimer_force_reprogram
hrtimer_get_res
hrtimer_wakeup
hrtimer_init
hrtimer_get_remaining
hrtimer_try_to_cancel
hrtimer_cancel
hrtimer_start
hrtimer_start_range_ns
hrtimer_start_expires
hrtimer_get_next_event
hrtimer_interrupt
hrtimer_peek_ahead_timers
hrtimer_run_pending
hrtimer_run_queues
hrtimer_nanosleep
sys_nanosleep
hrtimer_nanosleep_restart
hrtimer_start_expires.clone.5
hrtimer_forward_now
hrtimer_restart


The set_ftrace_notrace prevents those functions from being
traced.

 # echo '*preempt*' '*lock*' > set_ftrace_notrace

Produces:

# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
          <idle>-0     [002] 69247.262737: need_resched <-mwait_idle
          <idle>-0     [002] 69247.262738: test_ti_thread_flag <-need_resched
          <idle>-0     [002] 69247.262739: __exit_idle <-cpu_idle
          <idle>-0     [002] 69247.262740: test_ti_thread_flag <-cpu_idle
          <idle>-0     [002] 69247.262741: enter_idle <-cpu_idle
          <idle>-0     [002] 69247.262742: mwait_idle <-cpu_idle
##### CPU 0 buffer started ####
          <idle>-0     [000] 69247.262742: need_resched <-mwait_idle
          <idle>-0     [002] 69247.262742: need_resched <-mwait_idle
          <idle>-0     [000] 69247.262743: test_ti_thread_flag <-need_resched
          <idle>-0     [002] 69247.262743: test_ti_thread_flag <-need_resched
          <idle>-0     [000] 69247.262744: __exit_idle <-cpu_idle
          <idle>-0     [002] 69247.262744: trace_power_start.clone.5 <-mwait_idle
          <idle>-0     [000] 69247.262745: test_ti_thread_flag <-cpu_idle
          <idle>-0     [002] 69247.262745: need_resched <-mwait_idle
          <idle>-0     [000] 69247.262746: enter_idle <-cpu_idle

We can see that there's no more lock or preempt tracing.
```

```
Dynamic ftrace with the function graph tracer
---------------------------------------------

Although what has been explained above concerns both the
function tracer and the function-graph-tracer, there are some
special features only available in the function-graph tracer.

If you want to trace only one function and all of its children,
you just have to echo its name into set_graph_function:

 echo __do_fault > set_graph_function

will produce the following "expanded" trace of the __do_fault()
function:

 0)               |  __do_fault() {
 0)               |    filemap_fault() {
 0)               |      find_lock_page() {
 0)   0.804 us    |        find_get_page();
 0)               |        __might_sleep() {
 0)   1.329 us    |        }
 0)   3.904 us    |      }
 0)   4.979 us    |    }
 0)   0.653 us    |    _spin_lock();
 0)   0.578 us    |    page_add_file_rmap();
 0)   0.525 us    |    native_set_pte_at();
 0)   0.585 us    |    _spin_unlock();
 0)               |    unlock_page() {
 0)   0.541 us    |      page_waitqueue();
 0)   0.639 us    |      __wake_up_bit();
 0)   2.786 us    |    }
 0) + 14.237 us   |  }
 0)               |  __do_fault() {
 0)               |    filemap_fault() {
 0)               |      find_lock_page() {
 0)   0.698 us    |        find_get_page();
 0)               |        __might_sleep() {
 0)   1.412 us    |        }
 0)   3.950 us    |      }
 0)   5.098 us    |    }
 0)   0.631 us    |    _spin_lock();
 0)   0.571 us    |    page_add_file_rmap();
 0)   0.526 us    |    native_set_pte_at();
 0)   0.586 us    |    _spin_unlock();
 0)               |    unlock_page() {
 0)   0.533 us    |      page_waitqueue();
 0)   0.638 us    |      __wake_up_bit();
 0)   2.793 us    |    }
 0) + 14.012 us   |  }

You can also expand several functions at once:

 echo sys_open > set_graph_function
 echo sys_close >> set_graph_function

Now if you want to go back to trace all functions you can clear
this special filter via:

 echo > set_graph_function
```

```
Outputting the trace on panic or oops
-------------------------------------

The tracer may be used to dump the trace for the oops'ing cpu on
a kernel oops into the system log. To enable this,
ftrace_dump_on_oops must be set. To set ftrace_dump_on_oops, one
can either add "ftrace_dump_on_oops" on the kernel command line
or use the sysctl function or set it via the proc system
interface.

  sysctl kernel.ftrace_dump_on_oops=1

or

  echo 1 > /proc/sys/kernel/ftrace_dump_on_oops


Here's an example of such a dump after a null pointer dereference.

BUG: unable to handle kernel NULL pointer dereference at (null)
IP: [<ffffffff8125022f>] sysrq_handle_crash+0x16/0x20
PGD 3ed76067 PUD 373c5067 PMD 0
Oops: 0002 [#1] PREEMPT SMP
last sysfs file: /sys/devices/system/cpu/cpu3/cache/index1/shared_cpu_map
Dumping ftrace buffer:
---------------------------------
    bash-1570    2..... 22115712us : test_ti_thread_flag <-mnt_want_write
    bash-1570    2..... 22115713us : file_move <-__dentry_open
    bash-1570    2..... 22115714us : _raw_spin_lock <-file_move
    bash-1570    2...1. 22115715us : do_raw_spin_lock <-_raw_spin_lock
    bash-1570    2...1. 22115716us : _raw_spin_unlock <-file_move
    bash-1570    2..... 22115717us : test_ti_thread_flag.clone.2 <-_raw_spin_unlock
    bash-1570    2..... 22115718us : security_dentry_open <-__dentry_open
[...]
  <idle>-0       0d..1. 22118642us : need_resched <-mwait_idle
    bash-1570    2d..1. 22118642us : test_ti_thread_flag <-pagefault_enable
  <idle>-0       3d..1. 22118642us : need_resched <-mwait_idle
  <idle>-0       1...1. 22118642us : __exit_idle <-cpu_idle
  <idle>-0       0d..1. 22118642us : test_ti_thread_flag <-need_resched
    bash-1570    2d..1. 22118643us : oops_enter <-oops_begin
  <idle>-0       3d..1. 22118643us : test_ti_thread_flag <-need_resched
  <idle>-0       0d..1. 22118643us : trace_power_start.clone.5 <-mwait_idle
  <idle>-0       1...1. 22118643us : test_ti_thread_flag <-cpu_idle
---------------------------------
CPU 2
Pid: 1570, comm: bash Not tainted 3.2.16-test #2 0C9316/Precision WorkStation 470
RIP: 0010:[<ffffffff8125022f>]  [<ffffffff8125022f>] sysrq_handle_crash+0x16/0x20
RSP: 0018:ffff880037027e38  EFLAGS: 00010096
RAX: 0000000000000010 RBX: 0000000000000063 RCX: 00000000ffffffe5
RDX: 0000000000000000 RSI: 0000000000000000 RDI: 0000000000000063
RBP: ffff880037027e38 R08: 0000000000000001 R09: ffffffffffffffff
R10: ffff880037027c48 R11: ffffffff819438bc R12: 0000000000000000
R13: ffffffff816ec1c0 R14: 0000000000000003 R15: 0000000000000296
FS:  00007f19e62f7700(0000) GS:ffff880001a80000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 0000000000000000 CR3: 000000003eda8000 CR4: 00000000000006e0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
```

```
Process bash (pid: 1570, threadinfo ffff880037026000, task ffff88003c036700)
Stack:
 ffff880037027e88 ffffffff81250662 ffff880037027ea8 ffffffff00000000
<0> fffffffffffffffb 0000000000000002 ffffffff81250701 ffff88003cfc8440
<0> 00007f19e6308000 0000000000000002 ffff880037027ea8 ffffffff81250738
Call Trace:
 [<ffffffff81250662>] __handle_sysrq+0xa3/0x142
 [<ffffffff81250701>] ? write_sysrq_trigger+0x0/0x3e
 [<ffffffff81250738>] write_sysrq_trigger+0x37/0x3e
 [<ffffffff8113d853>] proc_reg_write+0x90/0xaf
 [<ffffffff810f4685>] vfs_write+0xac/0x100
 [<ffffffff810f5591>] ? fget_light+0x40/0x8a
 [<ffffffff810f488e>] sys_write+0x4a/0x6e
 [<ffffffff81002cdb>] system_call_fastpath+0x16/0x1b


trace_pipe
----------

The trace_pipe outputs the same content as the trace file, but
the effect on the tracing is different. Every read from
trace_pipe is consumed. This means that subsequent reads will be
different. The trace is live.

 # echo function > current_tracer
 # cat trace_pipe > /tmp/trace.out &
[1] 4153
 # echo 1 > tracing_on
 # usleep 1
 # echo 0 > tracing_on
 # cat trace
# tracer: function
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |

 #
 # cat /tmp/trace.out
       <idle>-0     [003]   392.260222: trace_power_start.clone.5 <-mwait_idle
       <idle>-0     [003]   392.260223: need_resched <-mwait_idle
       <idle>-0     [003]   392.260224: test_ti_thread_flag <-need_resched
       <idle>-0     [003]   392.260225: __exit_idle <-cpu_idle
       <idle>-0     [003]   392.260226: test_ti_thread_flag <-cpu_idle
       <idle>-0     [003]   392.260227: enter_idle <-cpu_idle
       <idle>-0     [003]   392.260228: mwait_idle <-cpu_idle
       <idle>-0     [003]   392.260229: need_resched <-mwait_idle
       <idle>-0     [003]   392.260229: test_ti_thread_flag <-need_resched
       <idle>-0     [003]   392.260230: trace_power_start.clone.5 <-mwait_idle
       <idle>-0     [003]   392.260231: need_resched <-mwait_idle
       <idle>-0     [003]   392.260232: test_ti_thread_flag <-need_resched
       <idle>-0     [003]   392.260233: __exit_idle <-cpu_idle


Note, reading the trace_pipe file will block until more input is
added. By changing the tracer, trace_pipe will issue an EOF. We
needed to set the function tracer _before_ we "cat" the
trace_pipe file.


trace entries
```

```
-------------

Having too much or not enough data can be troublesome in
diagnosing an issue in the kernel. The file buffer_size_kb is
used to modify the size of the internal trace buffers. The
number listed is the number of kilobytes each CPU ring buffer has.
To know the full size, multiply the number of possible CPUS with
the size in buffer_size_kb.

 # cat buffer_size_kb
1408

Note, when modifying the ring buffer size, tracing will stop
while the buffer size is being updated, and then will continue
after the update.

 # echo 10000 > buffer_size_kb
 # cat buffer_size_kb
10000

-----------

More details can be found in the source code, in the kernel/trace/*.c files.
```

Revision History

Revision 4-0 **Wed Feb 27 2013** **Cheryn Tan**
Prepared for publishing (MRG 2.3)

Revision 3-3 **Wed Dec 19 2012** **Cheryn Tan**
BZ#866858 - Kernel rebase to version 3.6.

Revision 3-2 **Wed Dec 5 2012** **Cheryn Tan**
Docs QE review fixes.

Revision 3-0 **Mon Jun 11 2012** **Cheryn Tan**
Prepared for publishing (MRG 2.2).

Revision 2-8 **Fri Jun 1 2012** **Cheryn Tan**
BZ#813890 - Added documentation on /dev/cpu_dma_latency.

Revision 2-6 **Wed May 16 2012** **Cheryn Tan**
BZ#809309 - Further edits to kdump instructions.
BZ#821697 - Removed obsolete reference to bdflush.

Revision 2-5 **Tue May 15 2012** **Cheryn Tan**
BZ#809309 - Edited kdump instructions according to tech review.
BZ#813890 - Added section on using _COARSE clocks in Application Tuning chapter.
Removed "MRG Realtime specific gettimeofday speedup" section.

Revision 2-4 **Thu May 10 2012** **Cheryn Tan**
BZ#804847 - Added link to RHEL networking documentation.
BZ#805746 - Added link to Infiniband instructions.
BZ#813890 - Removed gettimeofday setup, added clocks and timestamping section.
BZ#800737 - Updated ftrace appendix with kernel changes.

Revision 2-3 **Thu May 3 2012** **Cheryn Tan**
BZ#804853 - Brief overview of HPN.
BZ#804847 - Brief overview of RoCEE.
BZ#809309 - Updated kdump instructions for RHEL6.
BZ#800737 - Updated references of MRG RT kernel to 3.2.

Revision 2-1 **Tue Feb 28 2012** **Tim Hildred**
Updated configuration file for new publication tool.

Revision 2-0 **Wed Dec 7 2011** **Alison Young**
Prepared for publishing

Revision 1-7 **Wed Nov 16 2011** **Alison Young**
BZ#752406 - change RHEL versions

Revision 1-5 **Tue Oct 12 2011** **Alison Young**
BZ#716559 - Event and Function trace updates

Revision 1-3 **Tue Oct 11 2011** **Alison Young**

BZ#717261 - Incorrect data
BZ#725667 - trace-cmd in RHEL 6

Revision 1-2 **Wed Oct 5 2011** **Alison Young**

BZ#712267 - Link to non-existent mailing list

Revision 1-1 **Thu Sep 22 2011** **Alison Young**

Version numbering change

Revision 1-0 **Thu Jun 23 2011** **Alison Young**

Prepared for publishing

Revision 0.1-5 **Thu June 02 2011** **Alison Young**

Rebuilt following brand package update

Revision 0.1-4 **Mon May 23 2011** **Alison Young**

Technical Review updates

Revision 0.1-3 **Mon May 16 2011** **Alison Young**

BZ#584297 - Reorganise the latency tracing sections
BZ#666962 - Update for RHEL6

Revision 0.1-2 **Thu Apr 05 2011** **Alison Young**

Minor Update

Revision 0.1-1 **Tue Apr 05 2011** **Alison Young**

BZ#683586 - Update Further Reading section
Minor XML updates

Revision 0.1-0 **Wed Feb 23 2011** **Alison Young**

Fork from 1.3