



Red Hat Enterprise MRG 2 Realtime Reference Guide

Reference guide for the Realtime component of Red Hat Enterprise MRG

Lana Brindley

Alison Young

Cheryn Tan

Red Hat Enterprise MRG 2 Realtime Reference Guide

Reference guide 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

Legal Notice

Copyright 2013 Red Hat, Inc. The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at . In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version. Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law. Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries. Linux is the registered trademark of Linus Torvalds in the United States and other countries. Java is a registered trademark of Oracle and/or its affiliates. XFS is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries. MySQL is a registered trademark of MySQL AB in the United States, the European Union and other countries. All other trademarks are the property of their respective owners. 1801 Varsity Drive Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700 Phone: 888 733 4281 Fax: +1 919 754 3701

Keywords

Abstract

This book contains reference material for the MRG Realtime component of the Red Hat Enterprise MRG distributed computing platform. For installation instructions, see the MRG Realtime Installation Guide. For information on tuning, see the MRG Realtime Tuning Guide.

Table of Contents

Preface	5
1. Document Conventions	5
1.1. Typographic Conventions	5
1.2. Pull-quote Conventions	7
1.3. Notes and Warnings	7
2. Getting Help and Giving Feedback	8
2.1. Do You Need Help?	8
2.2. We Need Feedback!	8
Part I. Hardware	9
Chapter 1. Processor cores	10
1.1. Caches	10
1.2. Interconnects	10
Chapter 2. Memory allocation	11
2.1. Demand paging	11
2.2. Using mlock to avoid memory faults	13
Chapter 3. Hardware interrupts	15
3.1. Level-signalled interrupts	16
3.2. Message-signalled interrupts	16
3.3. Non-maskable interrupts	17
3.4. System management interrupts	17
3.5. Advanced programmable interrupt controller	18
Part II. Application architecture	19
Chapter 4. Threads and processes	20
Chapter 5. Priorities and policies	22
Chapter 6. Affinity	23
6.1. Using the taskset command to set processor affinity	24
6.2. Using the sched_getaffinity() system call to set processor affinity	24
Chapter 7. Thread synchronization	26
7.1. Mutexes	26
7.2. Barriers	26
7.3. Condvars	26
7.4. Other types of synchronization	26
Chapter 8. Sockets	27
8.1. Socket options	27
Chapter 9. Shared memory	30
Chapter 10. Shared libraries	31
Part III. Library services	32
Chapter 11. Setting the scheduler	33
11.1. Using chrt to set the scheduler	33
11.2. Preemption	34
11.3. Using library calls to set priority	34

11.3.1. <code>sched_getscheduler</code>	35
11.3.2. <code>sched_setscheduler</code>	36
11.3.3. <code>sched_getparam</code> and <code>sched_setparam</code>	36
11.3.4. <code>sched_get_priority_min</code> and <code>sched_get_priority_max</code>	37
11.3.5. <code>sched_rr_get_interval</code>	38
Chapter 12. Creating threads and processes	40
Chapter 13. Mmap	41
Chapter 14. System calls	42
14.1. <code>sched_yield</code>	42
14.2. <code>getrusage()</code>	42
Chapter 15. Timestamping	43
15.1. Hardware clocks	43
15.1.1. Reading hardware clock sources	44
15.2. POSIX clocks	46
15.2.1. <code>CLOCK_MONOTONIC_COARSE</code> and <code>CLOCK_REALTIME_COARSE</code>	47
15.2.2. Using <code>clock_getres()</code> to compare clock resolution	47
15.2.3. Using C code to compare clock resolution	48
15.2.4. Using the <code>time</code> command to compare cost of reading clocks	49
Chapter 16. More information	51
16.1. Reporting bugs	51
16.2. Further reading	51
Revision history	52

Preface

Red Hat Enterprise MRG

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

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

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

MRG Realtime

Many industries and organizations need extremely high performance computing and may require low and predictable latency, especially in the financial and telecommunications industries. Latency, or response time, is defined as the time between an event and system response and is generally measured in microseconds (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.

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_Reference_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.

Part I. Hardware

Selecting and configuring the right hardware is a critical part of setting up a realtime environment. Hardware impacts the way that the system operates. System Management Interrupts, CPU cache design, and NUMA utilization can all be handled in different ways. Hardware can vary from vendor to vendor, and not all hardware is suited to realtime environments. Check the [list of certified hardware for MRG Realtime](#).

It is important when setting up a MRG Realtime environment that the application is designed in such a way that it interacts well with the available hardware. This section contains information on the ways that MRG Realtime uses hardware, and the areas to look out for.

Chapter 1. Processor cores

A *processor core* is a physical Central Processing Unit (CPU) in a computer. Cores are responsible for executing machine code. A *socket* is the connection between the processor and the motherboard of the computer. The socket is the location on the motherboard that the processor is placed into. A single core processor physically occupies one socket, and has one core available. A quad-core processor physically occupies one socket and has four cores available.

When designing realtime applications, take the number of available cores into account. It is also important to note how caches are shared among cores, and how the cores are physically connected.

If multiple cores are available to the application, use threads or processes to take advantage of them. If a program is written without using these constructs, it will only run on one processor at a time. A multi-core platform allows advantages to be gained through using different cores for different types of operations.

1.1. Caches

Often, the various threads of an application will need to synchronize access to a shared resource, such as a data structure. Performance can be improved in this case by knowing the cache layout of the system. The Tuna tool can be used to help determine the cache layout. Try binding interacting threads to cores, so that they share the cache. *Cache sharing* reduces memory faults by ensuring that the mutual exclusion primitive (mutex, condvar, or similar) and the data structure itself use the same cache.

1.2. Interconnects

It is important to examine the interconnects that occur between cores. As the number of cores in a machine rise, the more difficult and expensive it becomes to provide uniform access to the memory for all of them. Many hardware vendors now provide a transparent network of interconnects between cores and memory, known as a *NUMA* (non-uniform memory access) architecture. On NUMA systems, knowing the interconnect topology allows threads that communicate frequently to be placed on adjacent cores.

Chapter 2. Memory allocation

Linux-based operating systems use a virtual memory system. Any address referenced by a user-space application must be translated into a physical address. This is achieved through a combination of page tables and address translation hardware in the underlying computer system.

One consequence of having the translation mechanism in between a program and the actual memory is that the operating system can steal pages when required. This is achieved by marking a previously used page table entry as invalid, so that even under normal memory pressure, the operating system might scavenge pages from one application to give to another. This can have adverse affects on systems that require deterministic behavior. Instructions that normally execute in a fixed amount of time can take longer than normal because a page fault has been triggered.

2.1. Demand paging

Under Linux, all memory addresses generated by a program get passed through an address translation mechanism in the processor. The addresses are converted from a process-specific virtual address to a physical memory address. This is referred to as *virtual memory*.

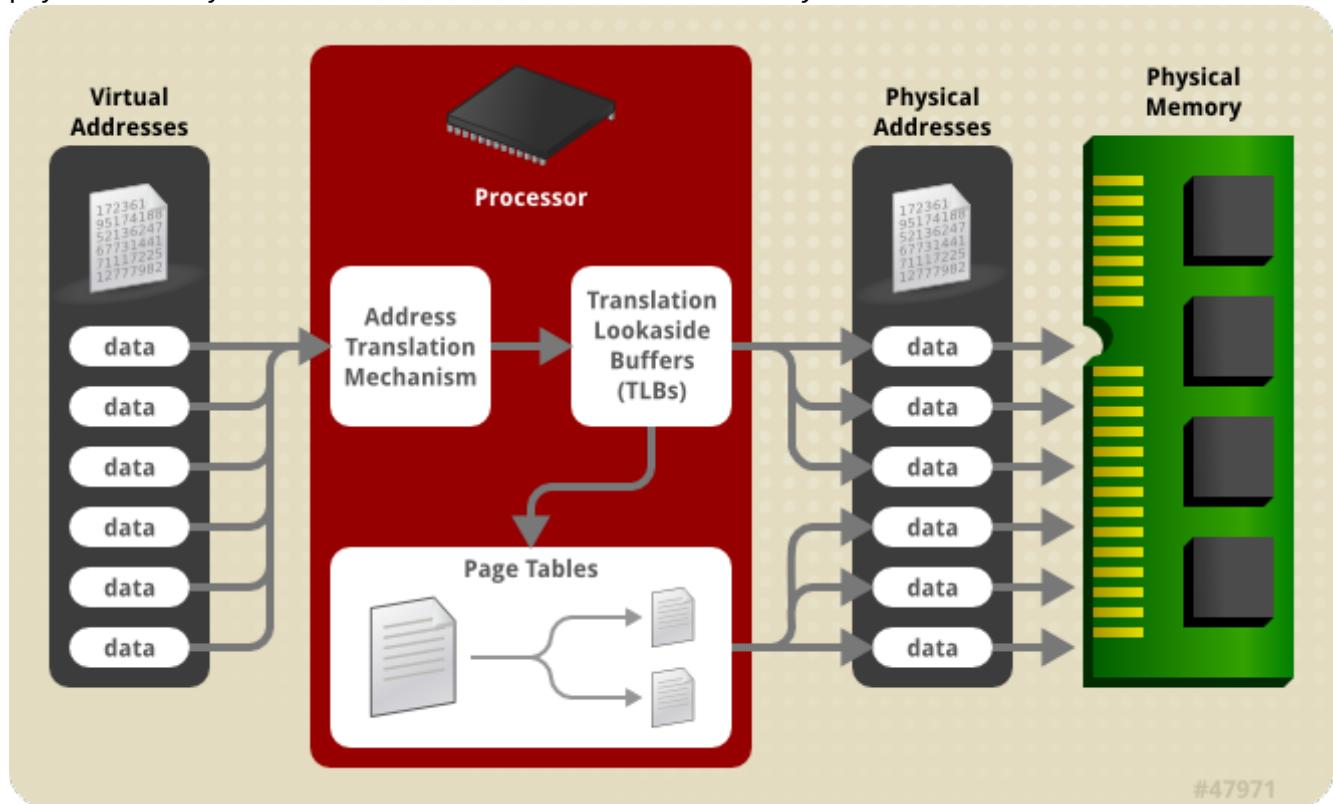


Figure 2.1. MRG Realtime virtual memory system

The two main components in the translation mechanism are *page tables* and *translation lookaside buffers (TLBs)*. Page tables are multi-level tables in physical memory that contain mappings for virtual to physical memory. These mappings are readable by the virtual memory translation hardware in the processor. TLBs are caches for page table translations.

When a page table entry has been assigned a physical address, it is referred to as the *resident working set*. When the operating system needs to free memory for other processes, it can remove pages from the working set. When this happens, any reference to a virtual address within that page will create a page fault, and the page will be reallocated. If the system is extremely low on physical memory, then this

process will start to *thrash*, constantly stealing pages from processes, and never allowing a process to complete. The virtual memory statistics can be monitored by looking for the **pgfault** value in the **/proc/vmstat** file.

TLBs are hardware caches of virtual memory translations. Any processor core with a TLB will check the TLB in parallel with initiating a memory read of a page table entry. If the TLB entry for a virtual address is valid, the memory read is aborted and the value in the TLB is used for the address translation.

TLBs operate on the principle of *locality of reference*. This means that if code stays in one region of memory for a significant period of time (such as loops or call-related functions) then the TLB references avoid the main memory for address translations. This can significantly speed up processing times. When writing deterministic and fast code, use functions that maintain locality of reference. This can mean using loops rather than recursion. If recursion cannot be avoided, place the recursion call at the end of the function. This is called *tail-recursion*, which makes the code work in a relatively small region of memory and avoid fetching table translations from main memory.

A potential source of memory latency is called a *minor page fault*. They are created when a process attempts to access a portion of memory before it has been initialized. In this case, the system will need to perform some operations to fill the memory maps or other management structures. The severity of a minor page fault can depend on system load and other factors, but they are usually short and have a negligible impact.

A more severe memory latency is a *major page fault*. These can occur when the system has to synchronize memory buffers with the disk, swap memory pages belonging to other processes, or undertake any other Input/Output activity to free memory. This occurs when the processor references a virtual memory address that has not had a physical page allocated to it. The reference to an empty page causes the processor to execute a fault, and instructs the kernel code to allocate a page and return, all of which increases latency dramatically.

When writing a multi-threaded application, it is important to consider the machine topology when designing the data decomposition. Topology is the memory hierarchy, and includes CPU caches and the NUMA node. Sharing data information very tightly on CPUs in different cache and NUMA domains can lead to traffic problems and bottlenecks.

Contention can create drastic performance problems. On some hardware, the traffic on the various memory buses are not subject to any fairness rules. Always check the hardware you are using in order to avoid this.

Memory allocation errors can not always be eliminated through the use of CPU affinity, scheduling policies, and priorities. When an application shows a performance drop, it can be beneficial to check if it is being affected by page faults. There are a number of ways of doing this, but a simple method is to look at the process information in the **/proc** directory. For a particular process PID, use the **cat** command to view the **/proc/PID/stat** file. The relevant entries in this file are:

- ▶ **Field 2** - filename of the executable
- ▶ **Field 10** - number of minor page faults
- ▶ **Field 12** - number of major page faults

When a process encounters a page fault all its threads will be frozen until the kernel handles the fault. There are several ways to address this problem, although the best solution is to adjust the source code to avoid page faults.

Example 2.1. Using the /proc file to check for page faults

This example uses the **/proc** file to check for page faults in a running process.

Use the **cat** command and a pipe function to return only the second, tenth, and twelfth lines of the **/proc/PID/stat** file:

```
# cat /proc/3366/stat | cut -d\ -f2,10,12
(bash) 5389 0
```

In the above output, PID 3366 is **bash**, and it has reported 5389 minor page faults, and no major page faults.

 **Note**

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

2.2. Using mlock to avoid memory faults

The **mlock** and **mlockall** system calls tell the system to lock to a specified memory range, and to not allow that memory to be paged. This means that once the physical page has been allocated to the page table entry, references to that page will not fault again.

There are two groups of mlock system calls available. The **mlock** and **munlock** calls lock and unlock a specific range of addresses. The **mlockall** and **munlockall** calls lock or unlock the entire program space.

Use of the **mlock** calls should be examined carefully and used with caution. If the application is large, or if it has a large data domain, the **mlock** calls can cause thrashing if the system cannot allocate memory for other tasks. If the application is entering a time sensitive region of code, an **mlockall** call prior to entering, followed by **munlockall** can reduce paging while in the critical section. Similarly, **mlock** can be used on a data region that is relatively static or that will grow slowly but needs to be accessed without page faulting.

Use of **mlock** will not guarantee that the program will experience no page faults. It is used to ensure that the data will stay in memory, but can not ensure that it will stay in the same page. Other functions such as **move_pages** and memory compactors can move data around despite the **mlock**.


Important

Always use **mlock** with care. Using it excessively can lead to an **out of memory** (OOM) error. Do not just put an **mlockall** call at the start of your application. It is recommended that only the data and text of the realtime portion of the application be locked.

Example 2.2. Using `mlock` in an application

This example uses the `mlock` call in a simple application.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>

char *
alloc_workbuf(int size)
{
    char *ptr;

    /* allocate some memory */
    ptr = malloc(size);

    /* return NULL on failure */
    if (ptr == NULL)
        return NULL;

    /* lock this buffer into RAM */
    if (mlock(ptr, size)) {
        free(ptr);
        return NULL;
    }
    return ptr;
}

void
free_workbuf(char *ptr, int size)
{
    /* unlock the address range */
    munlock(ptr, size);

    /* free the memory */
    free(ptr);
}
```

Note

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

- ▶ [mlock\(2\)](#)
- ▶ [mlock\(3\)](#)
- ▶ [mlockall\(2\)](#)
- ▶ [move_pages\(2\)](#)

Chapter 3. Hardware interrupts

Hardware interrupts are used by devices to communicate that they require attention from the operating system. Some common examples are a hard disk signaling that it has read a series of data blocks, or that a network device has processed a buffer containing network packets. Interrupts are also used for asynchronous events, such as the arrival of new data from an external network. Hardware interrupts are delivered directly to the CPU using a small network of interrupt management and routing devices. This chapter describes the different types of interrupt and how they are processed by the hardware and by the operating system. It also describes how the MRG Realtime kernel differs from the standard kernel in handling the types of interrupt.

A standard system receives many millions of interrupts over the course of its operation, including a semi-regular "timer" interrupt that periodically performs maintenance and system scheduling decisions. It may also receive special kinds of interrupts, such as NMI (Non-Maskable Interrupts) and SMI (System Management Interrupts).

Hardware interrupts are referenced by an *interrupt number*. These numbers are mapped back to the piece of hardware that created the interrupt. This enables the system to monitor which device created the interrupt and when it occurred.

In most computer systems, interrupts are handled as quickly as possible. When an interrupt is received, any current activity is stopped and an *interrupt handler* is executed. The handler will preempt any other running programs and system activities, which can slow the entire system down, and create latencies. MRG Realtime modifies the way interrupts are handled in order to improve performance, and decrease latency.

Example 3.1. Viewing interrupts on your system

To examine the type and quantity of hardware interrupts received by a Linux system, use the **cat** command to view **/proc/interrupts**:

```
$ cat /proc/interrupts
CPU0          CPU1
0:    13072311      0  IO-APIC-edge      timer
1:    18351        0  IO-APIC-edge      i8042
8:    190        0  IO-APIC-edge      rtc0
9:  118508      5415  IO-APIC-fasteoi  acpi
12:  747529     86120  IO-APIC-edge      i8042
14:  1163648      0  IO-APIC-edge      ata_piix
15:      0        0  IO-APIC-edge      ata_piix
16: 12681226    126932  IO-APIC-fasteoi ahci, uhci_hcd:usb2, radeon, yenta,
eth0
17:  3717841      0  IO-APIC-fasteoi uhci_hcd:usb3, HDA, iwl3945
18:      0        0  IO-APIC-fasteoi uhci_hcd:usb4
19:    577       68  IO-APIC-fasteoi ehci_hcd:usb1, uhci_hcd:usb5
NMI:      0        0 Non-maskable interrupts
LOC: 3755270    9388684 Local timer interrupts
RES: 1184857    2497600 Rescheduling interrupts
CAL: 12471      2914 function call interrupts
TLB: 14555     15567 TLB shootdowns
TRM:      0        0 Thermal event interrupts
SPU:      0        0 Spurious interrupts
ERR:      0
MIS:      0
```

The output shows the various types of hardware interrupt, how many have been received, which CPU was the target for the interrupt, and the device that generated the interrupt.

3.1. Level-signalled interrupts

Level-signalled interrupts, use a dedicated interrupt line to deliver voltage transitions.

The dedicated line can send one of two voltages to represent a binary 1 or binary 0. Once a signal has been sent by the line, it will remain in that state until the CPU specifically resets it. This is achieved by the CPU asking the generating device to stop *asserting* the line. This allows a number of devices to share a single interrupt line. If the CPU has instructed a device to stop asserting the line, and it remains asserted, there is another interrupt pending.

Although level-signalled interrupts require a high level of hardware logic in both the devices and the CPU, they also provide a number of benefits. Not only can they be used by more than one device, but they are almost completely unable to miss an interrupt.

3.2. Message-signalled interrupts

Many modern systems use *message-signalled interrupts*, which send the signal as a dedicated message on a packet or message-based electrical bus.

One common example of this type of bus is PCI Express (Peripheral Component Interconnect Express, or PCIe). These devices transmit a message as a type that the PCIe Host Controller interprets as an interrupt message. The host controller then sends the message on to the CPU.

Depending on the hardware, a PCIe system might send the signal using a dedicated interrupt line between the PCIe host controller and the CPU, or by sending the message over (for example) the CPU HyperTransport bus. Many PCIe systems can also operate in legacy mode, where legacy interrupt lines are implemented in order to support older operating systems, or Linux kernels booted with the option `pci=nomsi` on the kernel command line.

3.3. Non-maskable interrupts

An interrupt is said to be *masked* when it has been disabled, or when the CPU has been instructed to ignore it. A *non-maskable interrupt* (NMI) cannot be ignored, and is generally used only for critical hardware errors.

NMIs are normally delivered over a separate interrupt line. When an NMI is received by the CPU, it indicates that a critical error has occurred, and that the system is probably about to crash. The NMI is generally the best indication of what might have caused the problem.

Because NMIs are not able to be ignored, they are also used by some systems as a hardware monitor. The device sends a stream of NMIs, which are checked by an NMI handler in the processor. If certain conditions are met - such as an interrupt not being triggered after a specified length of time - the NMI handler can produce a warning and debugging information about the problem. This helps to identify and prevent system lockups.

3.4. System management interrupts

System management interrupts (SMIs) are used to offer extended functionality, such as legacy hardware device emulation. They can also be used for system management tasks. SMIs are similar to NMIs in that they use a special electrical signalling line directly into the CPU, and are generally not able to be masked.

When an SMI is received, the CPU will enter *System Management Mode* (SMM). In this mode, a very low-level handler routine is run to handle the SMIs. The SMM is typically provided directly from the system management firmware, often the BIOS or the EFI.

SMIs are most often used to provide legacy hardware emulation. A common example is to emulate a floppy disk drive. If there is no floppy disk device attached to the system, a virtualized network-managed emulation can be used instead. When the operating system attempts to access the floppy disk, an SMI is triggered and a handler provides the operating system with an emulated device instead. The operating system then treats the emulation as though it were the legacy device itself.

MRG Realtime can be adversely affected by SMIs because they take place without the direct involvement of the operating system. A poorly written SMI handling routine may consume many milliseconds of CPU time, and the operating system is not able to preempt the handler if it needs to. This situation creates periodic high latencies in an otherwise well-tuned, highly responsive system. Unfortunately, because SMI handlers can be used by a vendor to manage CPU temperature and fan control, it is not possible to disable them. Instead, it is recommended that you notify the vendor of the problem.



Note

You can attempt to isolate SMIs on a MRG Realtime system using the `hwlatdetect` utility, which is available in the `rt-tests` package. This utility is designed to measure periods of time during which the CPU has been stolen by an SMI handling routine.

3.5. Advanced programmable interrupt controller

The *advanced programmable interrupt controller* (APIC) was developed by Intel® to provide the ability to handle large amounts of interrupts, to allow each of these to be programmatically routed to a specific set of available CPUs (and for this to be changed accordingly), to support inter-CPU communication, and to remove the need for a large number of devices to share a single interrupt line.

APIC represents a series of devices and technologies that work together to generate, route, and handle a large number of hardware interrupts in a scalable and manageable way. It uses a combination of a local APIC built into each system CPU, and a number of Input/Output APICs that are connected directly to hardware devices. When a hardware device generates an interrupt, it is detected by the IO-APIC it is connected to, and then routed across the system APIC bus to a particular CPU. The operating system knows which IO-APIC is connected to which device, and to which particular interrupt line within that device because of a combination of information sources. Firstly, there is the ACPI DSDT (Advanced Configuration and Power Interface Differentiated System Description Table) that includes information about the specific wiring of the host system motherboard and peripheral components. Secondly, a device provides certain information about its available interrupt sources. Together, these two sets of data provide information about the overall interrupt hierarchy.

Complex APIC-based interrupt management strategies are possible, with the system APICs connected in hierarchies, and delivering interrupts to CPUs in a load-balanced fashion rather than targeting a specific CPU or set of CPUs.

Part II. Application architecture

The MRG Realtime kernel provides a number of constructs that are designed to help software developers build an application that performs to the highest possible standards. This section discusses those features and how to use them.

Throughout this and the next sections, instructions are given for tuning the MRG Realtime kernel directly. Most changes can also be performed using a tool called *Tuna*. It has a graphical interface, or can be run through the command shell.

Tuna can be used to change attributes of threads and interrupts, such as scheduling policy, scheduler priority and processor affinity. It 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.

Information on installing and using *Tuna* can be found in the *MRG Tuna User Guide*.

Chapter 4. Threads and processes

Although all programs use threads and processes, MRG Realtime handles them in a different way to standard Red Hat Enterprise Linux. This chapter explains the MRG Realtime approach to threads and processes.

Each CPU core is limited in the amount of work it can handle. To achieve greater efficiency, applications can execute different tasks simultaneously on multiple cores. This is called *parallelizing*.

Programs can be parallelized using *threads*. However, threads and processes are often confused, so it is important to understand the differences in the terms.

Process

A UNIX®-style process is an operating system construct that contains:

1. Address mappings for virtual memory
2. An execution context (PC, stack, registers)
3. State/Accounting information

Linux processes started as exactly this style of process. When the concept of more than one process running inside one address space was developed, Linux turned to a process structure that shares an address space with another process. This works well, as long as the process data structure is kept small. For the remainder of this document, the term *process* refers to an independent address space, potentially containing multiple threads.

Thread

Strictly, a thread is a schedulable entity that contains:

1. A program counter (PC)
2. A register context
3. A stack pointer

Multiple threads can exist within a process.

When programming on a MRG Realtime system, there are two potential ways to parallelize the programs.

1. Use the **fork** and **exec** functions to create new processes
2. Use the Posix Threads (pthreads) API to create new threads within an already running process

Note

Evaluate how the components will interact before deciding how to parallelize them. If the components are independent of one another and will not interact very much or at all then creating a new address space and running as a new process is usually the better option. If, however, the components will need to share data or communicate frequently, running them as threads within one address space will usually be more efficient.

 Note

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

- ▶ `fork(2)`
- ▶ `exec(2)`
- ▶ *Programming with POSIX Threads*, David R. Butenhof, Addison-Wesley, ISBN 0-201-63392-2
- ▶ *Advanced Programming in the UNIX Environment*, 2nd Ed., W. Richard Stevens and Stephen A. Rago, Addison-Wesley, ISBN 0-201-43307-9
- ▶ “*POSIX Threads Programming*”, Blaise Barney, Lawrence Livermore National Laboratory,
<http://www.llnl.gov/computing/tutorials/pthreads/>

Chapter 5. Priorities and policies

All Linux threads have one of the following *scheduling policies*:

- ▶ **SCHED_OTHER** or **SCHED_NORMAL**: The default policy
- ▶ **SCHED_BATCH**: Similar to **SCHED_OTHER**, but with a throughput orientation
- ▶ **SCHED_IDLE**: A lower priority than **SCHED_OTHER**
- ▶ **SCHED_FIFO**: A first in/first out realtime policy
- ▶ **SCHED_RR**: A round-robin realtime policy

The policies that are critical to MRG Realtime are **SCHED_OTHER**, **SCHED_FIFO**, and **SCHED_RR**.

SCHED_OTHER or **SCHED_NORMAL** is the default scheduling policy for Linux threads. It has a dynamic priority that is changed by the system based on the characteristics of the thread. Another thing that effects the priority of **SCHED_OTHER** threads is their *nice value*. The nice value is a number between -20 (highest priority) and 19 (lowest priority). By default, **SCHED_OTHER** threads have a nice value of 0. Adjusting the nice value will change the way the thread is handled.

Threads with a **SCHED_FIFO** policy will run ahead of **SCHED_OTHER** tasks. Instead of using nice values, **SCHED_FIFO** uses a fixed priority between 1 (lowest) and 99 (highest). A **SCHED_FIFO** thread with a priority of 1 will always be scheduled ahead of any **SCHED_OTHER** thread.

The **SCHED_RR** policy is very similar to the **SCHED_FIFO** policy. In the **SCHED_RR** policy, threads of equal priority are scheduled in a *round-robin* fashion. Generally, **SCHED_FIFO** is preferred over **SCHED_RR**.

SCHED_FIFO and **SCHED_RR** threads will run until one of the following events occurs:

- ▶ The thread goes to sleep or begins waiting for an event
- ▶ A higher-priority realtime thread becomes ready to run

If one of these events does not occur, the threads will run indefinitely on that processor, and lower-priority threads will not be given a chance to run. This can result in system service threads failing to run, and operations such as memory swapping and filesystem data flushing not occurring as expected.

Table 5.1. Policy priorities

Policy	Default priority value	Lowest priority value	Highest priority value
SCHED_FIFO		1	99
SCHED_RR		1	99
SCHED_OTHER	0	-20	19

Chapter 6. Affinity

Each thread and interrupt source in the system has a *processor affinity* property. The operating system scheduler uses this information to determine which threads and interrupts to run on which CPU.

Setting processor affinity, along with effective policy and priority settings, can help to achieve the maximum possible performance. Applications will always need to compete for resources, especially CPU time, with other processes. Depending on the application, related threads are often run on the same core. Alternatively, one application thread can be allocated to one core.

Systems that perform multitasking are naturally more prone to indeterminism. Even high priority applications may be delayed from executing while a lower priority application is in a critical section of code. Once the low priority application has exited the critical section, the kernel may safely preempt the low priority application and schedule the high priority application on the processor. Additionally, migrating processes from one CPU to another can be costly due to cache invalidation. MRG Realtime includes tools that address some of these issues and allow latencies to be better controlled.

Affinity is represented as a *bitmask*, where each bit in the mask represents a CPU core. If the bit is set to 1, then the thread or interrupt may run on that core; if 0 then the thread or interrupt is excluded from running on the core. The default value for an affinity bitmask is all ones, meaning the thread or interrupt may run on any core in the system.

By default, processes can run on any CPU. However, processes can be instructed to run on a predetermined selection of CPUs, by changing the affinity of the process. Child processes inherit the CPU affinities of their parents.

Some of the more typical affinity setups include:

- ▶ Reserve one CPU core for all system processes and allow the application to run on the remainder of the core, with one CPU core per application thread.
- ▶ Allow a thread application and a given kernel thread (such as the network softirq or a driver thread) on the same CPU.
- ▶ Pair producer and consumer threads on each CPU.

It is recommended that affinity settings are designed in conjunction with the program, to better match the expected behavior.

The usual practice for tuning affinities on a realtime system is to determine how many cores are needed to run the application and then isolate those cores. This can be achieved using the **Tuna** tool, or through the use of shell scripts to modify the bitmask value. The **taskset** command can be used to change the affinity of a process, while modifying the **/proc** filesystem entry changes the affinity of an interrupt.



Note

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

- ▶ *MRG Tuna User Guide*
- ▶ **taskset(1)**

6.1. Using the taskset command to set processor affinity

The **taskset** command sets and checks affinity information for a given process. These tasks can also be achieved using the Tuna tool.

Use the taskset command with the **-p**, or **--pid** option and the PID of the process to be checked. The **-c** or **--cpu-list** displays the information as a numerical list of cores, instead of as a bitmask.

The following command checks the affinity of the process with PID 1000. In this case, PID 1000 is permitted to use either CPU 0 or CPU 1:

```
# taskset -p -c 1000
pid 1000's current affinity list: 0,1
```

The affinity can be set by specifying the number of the CPU to which to bind the process. In this example, PID 1000 could previously run on either CPU 0 or CPU 1, and the affinity has been changed so that it can only run on CPU 1:

```
# taskset -p -c 1 1000
pid 1000's current affinity list: 0,1
pid 1000's new affinity list: 1
```

To define more than one CPU affinity, list both CPU numbers, separated by a comma:

```
# taskset -p -c 0,1 1000
pid 1000's current affinity list: 1
pid 1000's new affinity list: 0,1
```

The **taskset** command can also be used to start a new process with a particular affinity. This command will run the **/bin/my-app** application on CPU 4:

```
# taskset -c 4 /bin/my-app
```

For further granularity, the priority and policy can also be set. This command runs the **/bin/my-app** application on CPU 4, with a **SCHED_FIFO** policy and a priority of 78:

```
# taskset -c 5 chrt -f 78 /bin/my-app
```

6.2. Using the sched_getaffinity() system call to set processor affinity

In addition to the **taskset** command, processor affinity can also be set using the **sched_getaffinity()** system call.

The following code excerpt retrieves the CPU affinity information for a specified PID. If the PID passed to it is 0, it will return the affinity information for the current process:

```
int sched_getaffinity(pid_t pid, size_t setsiz, const cpu_set_t *set)
```



Note

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

- ▶ [sched_getaffinity\(2\)](#)
- ▶ [sched_setaffinity\(2\)](#)

Chapter 7. Thread synchronization

When threads require access to shared resources, it is coordinated using *thread synchronization*. The three thread synchronization mechanisms used on Linux are:

1. Mutexes
2. Barriers
3. Condvars

7.1. Mutexes

The word *mutex* is derived from the term *mutual exclusion*. A mutex is a POSIX threads construct, and is created using the `pthread_create_mutex` library call. A mutex serializes access to each section of code, so that only one thread of an application is running the code at any one time.

Similar to a mutex is a *futex*, or Fast User muTEX, which is an internal mechanism used to implement mutexes. Futexes use shared conventions between the kernel and the C library. This allows an uncontended mutex to be locked or freed without a context switch to kernel space.

7.2. Barriers

Barriers operate in a very different way to other thread synchronization methods. Instead of serializing access to code regions, barriers block all threads until a pre-determined number of them have accumulated. The barrier will then allow all threads to continue. Barriers are used in situations where a running application needs to be certain that all threads have completed their tasks before execution can continue.

7.3. Condvars

A *condvar*, or condition variable, is a POSIX thread construct that waits for a particular condition to be achieved before proceeding. In general the condition being signaled pertains to the state of data that the thread shares with another thread. For example, a condvar can be used to signal that a data entry has been put into a processing queue and a thread waiting to process data from the queue can now proceed.

7.4. Other types of synchronization

Prior to the advent of POSIX threads, thread synchronization occurred between processes. The most common mechanisms were the System V IPC calls for shared memory, message queues, and semaphores. The use of the System V IPC calls has now been deprecated in favor of POSIX thread calls.

Chapter 8. Sockets

A *socket* is a bi-directional data transfer mechanism. They are used to transfer data between two processes. The two processes can be running on the same system as Unix-domain or loopback sockets, or on different systems as network sockets.

There are no special options or restriction to using sockets on a MRG Realtime system.

8.1. Socket options

There are two socket options that are relevant to MRG Realtime applications: **TCP_NODELAY** and **TCP_CORK**.

TCP_NODELAY

TCP is the most common transport protocol, which means it is often used to solve many different needs. As new application and hardware features are developed, and kernel architecture optimizations are made, TCP has had to introduce new heuristics to handle the changes effectively.

These heuristics can result in a program becoming unstable. Because the behavior changes as the underlying operating system components change, they should be treated with care.

One example of heuristic behavior in TCP is that small buffers are delayed. This allows them to be sent as one network packet. This generally works well, but it can also create latencies. For MRG Realtime applications, **TCP_NODELAY** is a socket option that can be used to turn this behavior off. It can be enabled through the **setsockopt** sockets API, with the following function:

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_NODELAY, &one, sizeof(one));
```

For this option to be used effectively, the applications must avoid doing small buffer writes, as TCP will send these buffers as individual packets. **TCP_NODELAY** can also interact with other optimization heuristics to result in poor overall performance.

If applications have several buffers that are logically related and that should be sent as one packet it will achieve better latency and performance by building a contiguous packet before sending. The packet can then be sent as one using a socket with **TCP_NODELAY** enabled.

Alternatively, if the memory buffers are logically related but not already contiguous, use them to build an I/O vector. It can then be passed to the kernel using **writev** on a socket with **TCP_NODELAY** enabled.

TCP_CORK

Another TCP socket option that works in a similar way is **TCP_CORK**. When enabled, TCP will delay all packets until the application removes the cork, and allows the stored packets to be sent. This allows applications to build a packet in kernel space, which is useful when different libraries are being used to provide layer abstractions.

The **TCP_CORK** option can be enabled by using the following function:

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_CORK, &one, sizeof(one));
```

Enabling **TCP_CORK** is often referred to as **corking the socket**.

In a situation where the kernel is not able to identify when to remove the cork, it can be manually removed with the function:

```
int zero = 0;
setsockopt(descriptor, SOL_TCP, TCP_CORK, &zero, sizeof(zero));
```

Once the socket is uncorked, TCP will send the accumulated logical package immediately, without waiting for further packets from the application.

Example 8.1. Using **TCP_NODELAY** and **TCP_CORK**

This example demonstrates the performance impact that **TCP_NODELAY** and **TCP_CORK** can have on an application.

The server waits for packets of 30 bytes and then sends a 2 byte packet in response. To start with, define the TCP port and the number of packets it should process. In this example, it is 10,000 packets:

```
$ ./tcp_nodelay_server 5001 10000
```

The server does not need to have any socket options set.

If the client is run without any arguments, the default socket options will be used. Use the **no_delay** option to enable **TCP_NODELAY** socket options. Use the **cork** option to enable **TCP_CORK**. In all cases it will send 15 packets, each of two bytes, and wait for a response from the server.

This example uses a loopback interface to demonstrate three variations.

In the first variation, neither **TCP_NODELAY** nor **TCP_CORK** are in use. This is a baseline measurement. TCP coalesces writes and has to wait to check if the application has more data than can optimally fit in the network packet:

```
$ ./tcp_nodelay_client localhost 5001 10000
10000 packets of 30 bytes sent in 400129.781250 ms: 0.749757 bytes/ms
```

The second variation uses **TCP_NODELAY** only. TCP is instructed not to coalesce small packets, but to send buffers immediately. This improves performance significantly, but creates a large number of network packets for each logical packet.

```
$ ./tcp_nodelay_client localhost 5001 10000 no_delay
10000 packets of 30 bytes sent in 1649.771240 ms: 181.843399 bytes/ms using
TCP_NODELAY
```

The third variation uses **TCP_CORK** only. It halves the time required to send the same number of logical packets. This is because TCP coalesces full logical packets in its buffers, and sends fewer overall network packets.

```
$ ./tcp_nodelay_client localhost 5001 10000 cork
10000 packets of 30 bytes sent in 850.796448 ms: 352.610779 bytes/ms using
TCP_CORK
```

In this scenario, **TCP_CORK** is the best technique to use. It allows the application to precisely convey the information that a packet is finished and must be sent without delay. When developing programs, if they need to send bulk data from a file, consider using **TCP_CORK** with **sendfile**.



Note

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

- ▶ `sendfile(2)`
- ▶ “*TCP nagle sample applications*”, which are example applications of ghost protocols written in C. To download them, right-click and save from the following links:
 - http://oops.ghostprotocols.net:81/acme/tcp_nodelay_client.c
 - http://oops.ghostprotocols.net:81/acme/tcp_nodelay_server.c

Chapter 9. Shared memory

One of the main advantages of program threads is that all threads created in one process context share the same address space. This means that all data-structures become accessible to them. However, it is not always appropriate for applications to use threads that share address space. This could be because of a distributed development team, or the size of the application being developed, among other reasons. In this case, separate processes might need to share part of the address space. This can be achieved on both Red Hat Enterprise Linux and MRG Realtime Linux kernels by using *shared memory*.

The original mechanism for sharing a memory region between two processes was the System V IPC **shmem** set of calls. These calls are quite capable, but overly complicated and cumbersome for the vast majority of use cases. For this reason, they have been deprecated on the MRG Realtime kernel and should no longer be used.

MRG Realtime uses POSIX shared memory calls, such as **shm_open** and **mmap**.



Note

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

- ▶ [shm_open\(3\)](#)
- ▶ [shm_overview\(7\)](#)
- ▶ [mmap\(2\)](#)

Chapter 10. Shared libraries

Dynamic Shared Objects (DSOs) are commonly referred to as a *shared library*, and are used to share code between separate process address spaces. The DSO is loaded once by the **ld.so** system loader. From there, they are mapped into the address space of processes that require symbols from the library. Until the first reference to a symbol is encountered it cannot be evaluated. Evaluating the symbol only when it is referenced can be a source of latency. This is because memory pages can be on disk, and caches can become invalidated. Evaluating symbols in advance is a safe side procedure that can help to improve latency. .

Resolving symbols at program startup can slightly slow down program initialization. However, it also avoids non-deterministic latencies during program execution that can be caused by symbol lookup. Symbol resolution at application startup can be done using the **LD_BIND_NOW** environment variable. Setting **LD_BIND_NOW** to any value other than null will cause the system loader to lookup all unresolved symbols at program load time.



Note

For more information, or for further reading, the following man page is related to the information given in this section:

- » [ld.so\(8\)](#)

Part III. Library services

System commands are used to manipulate priorities, processor affinity and scheduling policies. It is also possible to manipulate these elements from within user applications using library functions.

This section explains how to select priorities, processor affinity, and scheduler policies using library functions, and how to observe the results of those changes.

Chapter 11. Setting the scheduler

There are two different ways to configure and observe process configurations: the command line utilities, and the Tuna graphical tool. This section uses the command line tools, but all actions can also be performed using Tuna. There is more information on using the Tuna tool in the *MRG Realtime Tuning Guide: Using the Tuna Interface* chapter.

11.1. Using chrt to set the scheduler

The **chrt** is used to check and adjust scheduler policies and priorities. It can start new processes with the desired properties, or change the properties of a running process.

To check the attributes of a particular process, use the **--pid** or **-p** option alone to specify the process ID (PID):

```
# chrt -p 468
pid 468's current scheduling policy: SCHED_FIFO
pid 468's current scheduling priority: 85

# chrt -p 476
pid 476's current scheduling policy: SCHED_OTHER
pid 476's current scheduling priority: 0
```

To set the scheduling policy of a process, use the appropriate command option:

Table 11.1. Policy options for the chrt command

Short option	Long option	Description
-f	--fifo	Set schedule to SCHED_FIFO
-o	--other	Set schedule to SCHED_OTHER
-r	--rr	Set schedule to SCHED_RR

To set the priority of a process, specify the value before the PID of the process that is being changed. The following command will set the process with PID 1000 to **SCHED_FIFO**, with a priority of 50:

```
# chrt -f -p 50 1000
```

The following command will set the same process (PID 1000) to **SCHED_OTHER**, with a priority of 0:

```
# chrt -o -p 0 1000
```

To start a new application with a given policy and priority, specify the name of the application (and the path, if necessary) along with the attributes. The following command will start **/bin/my-app**, with a policy of **SCHED_FIFO** and a priority of 36:

```
# chrt -f 36 /bin/my-app
```



Note

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

- ▶ [Tuna User Guide](#)
- ▶ [chrt\(1\)](#)

11.2. Preemption

A process can *voluntarily* yield the CPU either because it has completed, or because it is waiting for an event (such as data from a disk, a key press, or for a network packet).

A process can also *involuntarily* yield the CPU. This is referred to as *preemption*, and occurs when a higher priority process wants to use the CPU. Preemption can have a particularly negative impact on performance, and constant preemption can lead to a state known as *thrashing*. This problem occurs when processes are constantly preempted and no process ever gets to run completely.

To check voluntary and involuntary preemption occurring on a single process, check the contents of the **/proc/PID/status**, where **PID** is the PID of the process. The following command checks the preemption of the process with PID 1000:

```
# grep voluntary /proc/1000/status
voluntary_ctxt_switches: 194529
nonvoluntary_ctxt_switches: 195338
```

Changing the priority of a task can help reduce involuntary preemption.



Note

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

- ▶ [Tuna User Guide](#)
- ▶ [grep\(1\)](#)

11.3. Using library calls to set priority

Library calls are used to set the priority of non-realtime processes. These are:

- ▶ **nice**
- ▶ **getpriority**
- ▶ **setpriority**

These functions operate by retrieving and adjusting the *nice value* of the process. The *nicer* a process is, the lower its priority. Decreasing the nice value increments the priority.

Realtime processes use a different set of library calls to control policy and priority, which will be detailed in this section.



Important

The following functions all require the inclusion of a **sched.h** header file. Ensure you always check the return codes from functions. The appropriate man pages outline the various codes used.



Further reading

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

- ▶ [nice\(2\)](#)
- ▶ [getpriority\(2\)](#)
- ▶ [setpriority\(2\)](#)

11.3.1. **sched_getscheduler**

The **sched_getscheduler()** function retrieves the scheduler policy for a given PID:

```
#include <sched.h>

int policy;

policy = sched_getscheduler(pid_t pid);
```

The symbols **SCHED_OTHER**, **SCHED_RR** and **SCHED_FIFO** are also defined in **sched.h**. They can be used to check the defined policy or to set the policy:

```
#include <stdio.h>
#include <unistd.h>
#include <sched.h>

main(int argc, char *argv[])
{
    pid_t pid;
    int policy;

    if (argc < 2)
        pid = 0;
    else
        pid = atoi(argv[1]);

    printf("Scheduler Policy for PID: %d -> ", pid);

    policy = sched_getscheduler(pid);

    switch(policy) {
        case SCHED_OTHER: printf("SCHED_OTHER\n"); break;
        case SCHED_RR:    printf("SCHED_RR\n"); break;
        case SCHED_FIFO:  printf("SCHED_FIFO\n"); break;
        default:          printf("Unknown...\n");
    }
}
```

11.3.2. `sched_setscheduler`

The scheduler policy and other parameters can be set using the `sched_setscheduler()` function. Currently, realtime policies have one parameter, `sched_priority`. This parameter is used to adjust the priority of the process.

The `sched_setscheduler` function requires three parameters, in the form:

`sched_setscheduler(pid_t pid, int policy, const struct sched_param *sp);`



Note

The `sched_setscheduler(2)` man page lists all possible return values of `sched_setscheduler`, including the error codes.

If the `pid` is zero, the `sched_setscheduler()` function will act on the calling process.

The following code excerpt sets the scheduler policy of the current process to **SCHED_FIFO** and the priority to 50:

```
struct sched_param sp = { .sched_priority = 50 };
int ret;

ret = sched_setscheduler(0, SCHED_FIFO, &sp);
if (ret == -1) {
    perror("sched_setscheduler");
    return 1;
}
```

11.3.3. `sched_getparam` and `sched_setparam`

The **sched_setparam()** function is used to set the scheduling parameters of a particular process. This can then be verified using the **sched_getparam()** function.

Unlike the **sched_getscheduler()** function, which only returns the scheduling policy, the **sched_getparam()** function returns all scheduling parameters for the given process.

The following code excerpt reads the priority of a given realtime process and increments it by two:

```
struct sched_param sp;
int ret;

/* reads priority and increments it by 2 */
ret = sched_getparam(0, &sp);
sp.sched_priority += 2;

/* sets the new priority */
ret = sched_setparam(0, &sp);
```



Note

If the code above was used in a real application, it would also need to check the return values from the function, and handle any errors appropriately.



Important

Be careful with incrementing priorities. Continually adding two as in this example might eventually lead to an invalid priority.

11.3.4. **sched_get_priority_min** and **sched_get_priority_max**

The **sched_get_priority_min** and **sched_get_priority_max** functions are used to check the valid priority range for a given scheduler policy.

The only possible error in this call will occur if the specified scheduler policy is not known by the system. In this case, the function will return **-1** and **errno** will be set to **EINVAL**.

```
#include <stdio.h>
#include <unistd.h>
#include <sched.h>

main()
{
    printf("Valid priority range for SCHED_OTHER: %d - %d\n",
           sched_get_priority_min(SCHED_OTHER),
           sched_get_priority_max(SCHED_OTHER));

    printf("Valid priority range for SCHED_FIFO: %d - %d\n",
           sched_get_priority_min(SCHED_FIFO),
           sched_get_priority_max(SCHED_FIFO));

    printf("Valid priority range for SCHED_RR: %d - %d\n",
           sched_get_priority_min(SCHED_RR),
           sched_get_priority_max(SCHED_RR));
}
```

 **Note**

Both **SCHED_FIFO** and **SCHED_RR** can be any number within the range of 1 to 99. POSIX is not guaranteed to honor this range, however, and portable programs should use these calls.

11.3.5. `sched_rr_get_interval`

The **SCHED_RR** policy differs slightly from the **SCHED_FIFO** policy. **SCHED_RR** allocates concurrent processes that have the same priority in a round-robin rotation. In this way, each process is assigned a timeslice. The `sched_rr_get_interval()` function will report the timeslice that has been allocated to each process.

Even though POSIX requires that this function must work only with **SCHED_RR** processes, the `sched_rr_get_interval()` function is able to retrieve the timeslice length of any process on Linux.

The timeslice information is returned as a ***timespec***, or the number of seconds and nanoseconds since the base time of 00:00:00 GMT, 1 January 1970:

```
struct timespec {
    time_t tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */
}
```

The `sched_rr_get_interval` function requires the PID of the process, and a `struct timespec`:

```
#include <stdio.h>
#include <sched.h>

main()
{
    struct timespec ts;
    int ret;

    /* real apps must check return values */
    ret = sched_rr_get_interval(0, &ts);

    printf("Timeslice: %lu.%lu\n", ts.tv_sec, ts.tv_nsec);
}
```

The following commands run the test program **sched_03**, with varying policies and priorities. Processes with a **SCHED_FIFO** policy will return a timeslice of 0 seconds and 0 nanoseconds, indicating that it is infinite:

```
$ chrt -o 0 ./sched_03
Timeslice: 0.38994072
```

```
$ chrt -r 10 ./sched_03
Timeslice: 0.99984800
```

```
$ chrt -f 10 ./sched_03
Timeslice: 0.0
```

Chapter 12. Creating threads and processes

Process and thread creation are subject to system load, and integral to resource allocation and CPU time sharing. In some scenarios, a delay between an event occurring and it being handled is acceptable. In most situations, however, it creates unwanted and unnecessary latencies. To prevent this, the pool of processes or threads should always be created in advance, before they are called upon to service a request. For more information see [Chapter 4, Threads and processes](#).

Chapter 13. Mmap

The **mmap** system call allows a file (or parts of a file) to be mapped to memory. This allows the file content to be changed with a memory operation, avoiding system calls and input/output operations.

Always synchronize the changes to disk, and plan for a process hang that could result in data loss.



Note

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

- ▶ [mmap\(2\)](#)
- ▶ *Linux System Programming* by Robert Love

Chapter 14. System calls

14.1. `sched_yield`

The **`sched_yield`** function was originally designed to cause a processor to select a different process other than the running one. This type of request is prone to failure when issued from within a poorly-written application.

When the **`sched_yield()`** function is used within processes with realtime priorities, it can display unexpected behavior. The process that has called **`sched_yield`** gets moved to the tail of the queue of processes running at that priority. When this occurs in a situation where there are no other processes running at the same priority, the process that called **`sched_yield`** continues running. If the priority of that process is high, it can potentially create a busy loop, rendering the machine unusable.

In general, do not use **`sched_yield`** on realtime processes.

14.2. `getrusage()`

The **`getrusage`** function is used to retrieve important information from a given process or its threads. This will not provide all the information available, but will report on information such as context switches and page faults.

Chapter 15. Timestamping

15.1. Hardware clocks

Multiprocessor systems such as NUMA or SMP have multiple instances of clock sources. The way clocks interact among themselves and the way they react to system events, such as CPU frequency scaling or entering energy economy modes, determine whether they are suitable clock sources for the Realtime kernel.

During boot time the kernel discovers the available clock sources and selects one to use. The preferred clock source is the Time Stamp Counter (TSC), but if it is not available the High Precision Event Timer (HPET) is the second best option. However, not all systems have HPET clocks and some HPET clocks can be unreliable.

In the absence of TSC and HPET, other options include the ACPI Power Management Timer (ACPI_PM), the Programmable Interval Timer (PIT) and the Real Time Clock (RTC). The last two options are either costly to read or have a low resolution (time granularity), therefore they are sub-optimal for the Realtime kernel.

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 sample output 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
```

It is possible to select a different clock source, from the list presented in the

/sys/devices/system/clocksource/clocksource0/available_clocksource file. To do so, write the name of the clock source 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
```



Important

The kernel selects the best available clock source. Overriding the selected clock source is not recommended unless the implications are well understood.

While TSC is generally the preferred clock source, it also has several shortcomings. For example, the TSC clock can stop when the system goes to an idle state, or become out of sync when their CPUs enter deeper C-states (energy saving states) and perform speed- or frequency-scaling operations.

However, you can work around these TSC shortcomings by configuring additional kernel boot parameters. For instance, the **idle=poll** parameter forces the clock to avoid entering the idle state, and

the **`processor.max_cstate=1`** parameter prevents the clock from entering deeper C-states. Note however that in both cases there would be an increase on energy consumption, as the system would always run at top speed.



Note

For a comprehensive list of clock sources see the *Timing Measurements* chapter in *Understanding The Linux Kernel* by Daniel P. Bovet and Marco Cesati.

15.1.1. Reading hardware clock sources

Reading from the TSC is, basically, reading a register from the processor. Reading from the HPET clock means reading a memory area. Reading from the TSC is faster, which provides a significant performance advantage when timestamping hundreds of thousands of messages per second.

Using a simple program that reads the current clock source 10,000,000 times in a row, it is possible to observe the duration required to read the clock sources available:

Example 15.1. Comparing the cost of reading hardware clock sources

In this example, the clock source currently in use is TSC, as shown by the output of the **cat** command. The **time** command is used to view the duration required to read the clock source 10 million times.

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
# time ./clock_timing

real 0m0.601s
user 0m0.592s
sys 0m0.002s
```

The clock source is changed to HPET to compare the duration required to generate 10 million timestamps:

```
# echo hpet > /sys/devices/system/clocksource/clocksource0/current_clocksource
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
hpet
# time ./clock_timing

real 0m12.263s
user 0m12.197s
sys 0m0.001s
```

The steps are repeated with the ACPI_PM clock source:

```
# echo acpi_pm >
/sys/devices/system/clocksource/clocksource0/current_clocksource
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
acpi_pm
# time ./clock_timing

real 0m24.461s
user 0m0.504s
sys 0m23.776s
```

The **time(1)** man page provides detailed information on how to use the command and interpret its output. The example above uses the following categories:

- ▶ **real**: The total time spent beginning from program invocation until the process ends. **real** includes **user** and **sys** times, and will usually be larger than the sum of the latter two. If this process is interrupted by an application with higher priority, or by a system event such as a hardware interrupt (IRQ), this time spent waiting is also computed under **real**.
- ▶ **user**: The time the process spent in user space, performing tasks that did not require kernel intervention.
- ▶ **sys**: The time spent by the kernel while performing tasks required by the user process. These tasks include opening files, reading and writing to files or I/O ports, memory allocation, thread creation and network related activities.

As seen from the results of [Example 15.1, “Comparing the cost of reading hardware clock sources”](#), the efficiency of generating timestamps, in descending order, is: TSC, HPET, ACPI_PM. This is because of the increased overhead to access time values from the HPET and ACPI_PM timers.

15.2. POSIX clocks

POSIX is a standard for implementing and representing time sources. In contrast to the hardware clock, which is selected by the kernel and implemented across the system; the POSIX clock can be selected by each application, without affecting other applications in the system.

- ▶ **CLOCK_REALTIME**: it represents the time in the real world, also referred to as 'wall time' meaning the time as read from the clock on the wall. This clock is used to timestamp events, and when interfacing with the user. It can be modified by an user with the right privileges. However, user modification should be used with caution as it can lead to erroneous data if the clock has its value changed between two readings.
- ▶ **CLOCK_MONOTONIC**: represents the time monotonically increased since the system boot. This clock cannot be set by any process, and is the preferred clock for calculating the time difference between events. The following examples in this section use **CLOCK_MONOTONIC** as the POSIX clock.



Note

For more information on POSIX clocks see the following man page and book:

- ▶ `clock_gettime()`
- ▶ *Linux System Programming* by Robert Love

The function used to read a given POSIX clock is `clock_gettime()`, which is defined at `<time.h>`. The `clock_gettime()` command takes two parameters: the POSIX clock ID and a timespec structure which will be filled with the duration used to read the clock. The following example shows the function to measure the cost of reading the clock:

Example 15.2. Using `clock_gettime()` to measure the cost of reading POSIX clocks

```
#include <time.h>

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

    for(i=0; i<100000000; i++) {
        rc = clock_gettime(CLOCK_MONOTONIC, &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
```

15.2.1. **CLOCK_MONOTONIC_COARSE** and **CLOCK_REALTIME_COARSE**

The functions such as `clock_gettime()` and `gettimeofday()` have 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 user application to 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 the 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. The **_COARSE** variants are faster to read and have a precision (also known as resolution) of one millisecond (ms).

15.2.2. Using `clock_getres()` to compare clock resolution

Using the `clock_getres()` function you can check the resolution of a given POSIX clock. `clock_getres()` uses the same two parameters as `clock_gettime()`: the ID of the POSIX clock to be used, and a pointer to the timespec structure where the result is returned. The following function enables you to compare the precision between **CLOCK_MONOTONIC** and **CLOCK_MONOTONIC_COARSE**:

```
main()
{
int rc;
struct timespec res;

rc = clock_getres(CLOCK_MONOTONIC, &res);
if (!rc)
printf("CLOCK_MONOTONIC: %ldns\n", res.tv_nsec);
rc = clock_getres(CLOCK_MONOTONIC_COARSE, &res);
if (!rc)
printf("CLOCK_MONOTONIC_COARSE: %ldns\n", res.tv_nsec);
}
```

Example 15.3. Sample output of `clock_getres`

```
TSC:  
# ./clock_resolution  
CLOCK_MONOTONIC: 1ns  
CLOCK_MONOTONIC_COARSE: 999848ns (about 1ms)  
  
HPET:  
# ./clock_resolution  
CLOCK_MONOTONIC: 1ns  
CLOCK_MONOTONIC_COARSE: 999848ns (about 1ms)  
  
ACPI_PM:  
# ./clock_resolution  
CLOCK_MONOTONIC: 1ns  
CLOCK_MONOTONIC_COARSE: 999848ns (about 1ms)
```

15.2.3. Using C code to compare clock resolution

Using the following code snippet it is possible to observe the format of the data read from the **CLOCK_MONOTONIC** POSIX clock. All nine digits in the ***tv_nsec*** field of the **timespec** structure are meaningful as the clock has a nanosecond resolution. The example function, named **clock_test.c**, is as follows:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
  
main()  
{  
int i;  
struct timespec ts;  
  
for(i=0; i<5; i++) {  
clock_gettime(CLOCK_MONOTONIC, &ts);  
printf("%ld.%ld\n", ts.tv_sec, ts.tv_nsec);  
usleep(200);  
}  
}
```

Example 15.4. Sample output of `clock_test.c` and `clock_test_coarse.c`

As specified in the code above, the function reads the clock five times, with 200 microseconds between each reading:

```
# cc clock_test.c -o clock_test -lrt
# ./clock_test
218449.986980853
218449.987330908
218449.987590716
218449.987849549
218449.988108248
```

Using the same source code, renaming it to `clock_test_coarse.c` and replacing `CLOCK_MONOTONIC` with `CLOCK_MONOTONIC_COARSE`, the result would look something like:

```
# ./clock_test_coarse
218550.844862154
218550.844862154
218550.844862154
218550.845862154
218550.845862154
```

The `_COARSE` clocks have a one millisecond precision, therefore only the first three digits of the `tv_nsec` field of the timespec structure are significant. The result above could be read as:

```
# ./clock_test_coarse
218550.844
218550.844
218550.844
218550.845
218550.845
```

The `_COARSE` variants of the POSIX clocks are particularly useful in cases where timestamping can be performed with millisecond precision. The benefits are more evident on systems which use hardware clocks with high costs for the reading operations, such as ACPI_PM.

15.2.4. Using the `time` command to compare cost of reading clocks

Using the `time` command to read the clock source 10 million times in a row, you can compare the costs of reading `CLOCK_MONOTONIC` and `CLOCK_MONOTONIC_COARSE` representations of the hardware clocks available. The following example uses TSC, HPET and ACPI_PM hardware clocks. For more information on how to decipher the output of the `time` command see [Section 15.1.1, “Reading hardware clock sources”](#).

Example 15.5. Comparing the cost of reading POSIX clocks

```

TSC:
# time ./clock_timing_monotonic

real 0m0.567s
user 0m0.559s
sys 0m0.002s

# time ./clock_timing_monotonic_coarse

real 0m0.120s
user 0m0.118s
sys 0m0.001s

HPET:
# time ./clock_timing_monotonic

real 0m12.257s
user 0m12.179s
sys 0m0.002s

# time ./clock_timing_monotonic_coarse

real 0m0.119s
user 0m0.118s
sys 0m0.000s

ACPI_PM:
# time ./clock_timing_monotonic

real 0m25.524s
user 0m0.451s
sys 0m24.932s

# time ./clock_timing_monotonic_coarse

real 0m0.119s
user 0m0.117s
sys 0m0.001s

```

As seen from [Example 15.5. “Comparing the cost of reading POSIX clocks”](#), the **sys** time (the time spent by the kernel to perform tasks required by the user process) is greatly reduced when the **_COARSE** clocks are used. This is particularly evident in the **ACPI_PM** clock timings, which indicates that **_COARSE** variants of POSIX clocks yield high performance gains on clocks with high reading costs.

Chapter 16. More information

16.1. Reporting bugs

Diagnosing a bug

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

1. 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 with the standard kernel. 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:
 - a. The upstream kernel on which MRG Realtime is based. For example, Red Hat Enterprise Linux 6 is based on 2.6.32 and MRG Realtime is based on 3.6
 - b. 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.
4. Continue to enter the bug information by assigning an 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.

16.2. Further reading

- ▶ Red Hat Enterprise MRG product information
 - <http://www.redhat.com/mrg>
- ▶ MRG Realtime Tuning Guide and other Red Hat Enterprise MRG documentation
 - http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_MR/
- ▶ Mailing list
 - To post to the list, send mail to rhemrg-users-list@redhat.com
 - Subscribe to the mailing list at: <https://www.redhat.com/mailman/listinfo/rhemrg-users-list>

Revision history

Revision 4-0	Wed Feb 27 2013	Cheryn Tan
Prepared for publishing - MRG 2.3.		
Revision 3-1	Wed Dec 19 2012	Cheryn Tan
BZ#866858 - Kernel rebase to version 3.6.		
Revision 3-0	Thu May 31 2012	Cheryn Tan
Prepared for publishing (MRG 2.2).		
Revision 2-5	Tue May 15 2012	Cheryn Tan
BZ#813989 - Edited Timestamping chapter.		
Revision 2-4	Wed May 9 2012	Cheryn Tan
BZ#813989 - Added chapter 15 - Timestamping.		
Revision 2-2	Tue Apr 10 2012	Cheryn Tan
Fixed docs QE bugs, updated kernel references, removed references to RHEL 5.		
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 1-6	Wed Nov 16 2011	Alison Young
BZ#752406 - change RHEL versions		
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 0.1-3	Thu May 19 2011	Alison Young
Technical review updates		
Revision 0.1-2	Mon May 16 2011	Alison Young
BZ#666961 - Update for RHEL6		
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