Red Hat Enterprise Linux for Real Time 8

Optimizing RHEL 8 for Real Time for low latency operation

Configuring the Linux real-time kernel on Red Hat Enterprise Linux 8

Last Updated: 2021-12-02

Red Hat Enterprise Linux for Real Time 8 Optimizing RHEL 8 for Real Time for low latency operation

Configuring the Linux real-time kernel on Red Hat Enterprise Linux 8

Legal Notice

Abstract

As an administrator, you can configure your workstations on the Real-Time RHEL kernel. Such adjustments bring performance enhancements, easier troubleshooting, or an optimized system.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your input on our documentation. Please let us know how we could make it better. To do so:

- For simple comments on specific passages:
  1. Make sure you are viewing the documentation in the *Multi-page HTML* format. In addition, ensure you see the **Feedback** button in the upper right corner of the document.
  2. Use your mouse cursor to highlight the part of text that you want to comment on.
  3. Click the **Add Feedback** pop-up that appears below the highlighted text.
  4. Follow the displayed instructions.
- For submitting more complex feedback, create a Bugzilla ticket:
  1. Go to the Bugzilla website.
  2. As the Component, use **Documentation**.
  3. Fill in the **Description** field with your suggestion for improvement. Include a link to the relevant part(s) of documentation.
  4. Click **Submit Bug**.

CHAPTER 1. REAL-TIME KERNEL TUNING IN RHEL 8

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 must be low, accountable, and predictable, Red Hat has a kernel replacement that can be tuned so that latency meets those needs. **RHEL for Real Time 8** provides seamless integration with **RHEL 8** and offers clients the opportunity to measure, configure, and record latency times within their organization.

RHEL for Real Time 8 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.

Before you begin, perform general system tuning of the standard **RHEL 8** system before using **RHEL for Real Time 8**. For more information on performing general **RHEL 8** system tuning, refer to the *RHEL 8 Tuning Guide*.

WARNING

Failure to perform these tasks may prevent getting consistent performance from a RHEL Real Time deployment.

1.1. TUNING GUIDELINES

- Real-time 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 tuning configurations 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 configuration 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 highest latency or resource exhaustion.

- 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 (for example, "The mouse moves more smoothly.") is usually wrong and varies from person to person. Do hard measurements and record them for later analysis.

- 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.

- 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.

- Use the available tools. The **tuna** 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** and **perf** utilities can help locate latency issues.

- Rather than hard-coding values into your application, use external tools to change policy, priority and affinity. Using external tools 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 startup logic to implement the settings when the application starts.

1.2. THREAD SCHEDULING POLICIES

Linux uses three main thread 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 real-time priority of 0 (zero).

- **SCHED_FIFO** (First in, first out)
  A real-time 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_FIFO** 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 a modification 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. SPECIFYING THE RHEL KERNEL TO RUN

You can boot any installed kernel, standard or Real Time. You can select the required kernel manually in the GRUB menu during booting. You can also configure which kernel boot by default.

When the RHEL for Real Time kernel is installed, it is automatically set to be the default kernel and is used on the next boot.

2.1. DISPLAYING THE DEFAULT KERNEL

You can display the kernel configured to boot by default.

Procedure

- To view the default kernel:

```
~]# grubby --default-kernel
/boot/vmlinuz-4.18.0-80.rt9.138.el8.x86_64
```

The **rt** in the output of the command shows that the default kernel is a real time kernel.

2.2. DISPLAYING THE RUNNING KERNEL

You can display the currently running kernel

Procedure

- To show which kernel the system is currently running.

```
~]# uname -a
Linux rt-server.example.com 4.18.0-80.rt9.138.el8.x86_64 …
```

NOTE

When the system receives a minor update, for example, from 8.3 to 8.4, the default kernel might automatically change from the Real Time kernel back to the standard kernel.

2.3. CONFIGURING THE DEFAULT KERNEL

You can configure the default boot kernel.

Procedure

1. List the installed Real Time kernels.

```
~]# ls /boot/vmlinuz*rt*
/boot/vmlinuz-4.18.0-80.rt9.138.el8.x86_64
```

2. Set the default kernel to the listed Real Time kernel.

```
~]# grubby --set-default real-time-kernel
```

-

Replace ***real-time-kernel*** with the Real Time kernel version. For example:

```
~]# grubby --set-default /boot/vmlinuz-4.18.0-80.rt9.138.el8.x86_64
```

Verification steps

- Display the default kernel:

```
~]# grubby --default-kernel
/boot/vmlinuz-4.18.0-80.rt9.138.el8.x86_64
```

CHAPTER 3. RUNNING AND INTERPRETING HARDWARE AND FIRMWARE LATENCY TESTS

You can test and verify that a potential hardware platform is suitable for real-time operations by running the **hwlatdetect** program with the RHEL Real Time kernel.

Prerequisites

- Ensure that the **RHEL-RT** (RHEL for Real Time) and **rt-tests** packages are installed.
- Check the vendor documentation for any tuning steps required for low latency operation. The vendor documentation can provide instructions to reduce or remove any System Management Interrupts (SMIs) that would transition the system into System Management Mode (SMM). While a system is in SMM, it runs firmware and not operating system code. This means that any timers that expire while in SMM wait until the system transitions back to normal operation. This can cause unexplained latencies, because SMIs cannot be blocked by Linux, and the only indication that we actually took an SMI can be found in vendor-specific performance counter registers.

WARNING

Red Hat strongly recommends that you do not completely disable SMIs, as it can result in catastrophic hardware failure.

3.1. RUNNING HARDWARE AND FIRMWARE LATENCY TESTS

You do not need to run any load on the system while running the **hwlatdetect** program, because the test is looking for latencies introduced by the hardware architecture or BIOS/EFI firmware. The default values for **hwlatdetect** are to poll for 0.5 seconds each second, and report any gaps greater than 10 microseconds between consecutive calls to fetch the time. **hwlatdetect** returns the **best** maximum latency possible on the system.

Therefore, if you have an application that requires maximum latency values of less than 10us and **hwlatdetect** reports one of the gaps as 20us, then the system can only guarantee latency of 20us.

NOTE

If **hwlatdetect** shows that the system cannot meet the latency requirements of the application, try changing the BIOS settings or working with the system vendor to get new firmware that meets the latency requirements of the application.

Prerequisites

- Ensure that the **RHEL-RT** and **rt-tests** packages are installed.

Procedure

- Run **hwlatdetect**, specifying the test duration in seconds.

hwlatdetect looks for hardware and firmware-induced latencies by polling the clock-source and looking for unexplained gaps.

```
# hwlatdetect --duration=60s
hwlatdetect:  test duration 60 seconds
 detector: tracer
 parameters:
  Latency threshold:    10us
  Sample window:        1000000us
  Sample width:         500000us
  Non-sampling period:  500000us
  Output File:          None

Starting test
test finished
Max Latency: Below threshold
Samples recorded: 0
Samples exceeding threshold: 0
```

Additional resources

- **hwlatdetect** man page.
- Interpreting hardware and firmware latency tests

3.2. INTERPRETING HARDWARE AND FIRMWARE LATENCY TEST RESULTS

This provides information about the output from the **hwlatdetect** utility.

Examples

- The following result represents a system that was tuned to minimize system interruptions from firmware. In this situation, the output of **hwlatdetect** looks like this:

```
# hwlatdetect --duration=60s
hwlatdetect:  test duration 60 seconds
 detector: tracer
 parameters:
  Latency threshold: 10us
  Sample window:     1000000us
  Sample width:      500000us
  Non-sampling period:  500000us
  Output File:       None

Starting test
test finished
Max Latency: Below threshold
Samples recorded: 0
Samples exceeding threshold: 0
```

- The following result represents a system that could not be tuned to minimize system interruptions from firmware. In this situation, the output of **hwlatdetect** looks like this:

```
# hwlatdetect --duration=10s
hwlatdetect:  test duration 10 seconds
 detector: tracer
 parameters:
  Latency threshold: 10us
  Sample window:     1000000us
  Sample width:      500000us
  Non-sampling period:  500000us
  Output File:       None

Starting test
test finished
Max Latency: 18us
Samples recorded: 10
Samples exceeding threshold: 10
SMIs during run: 0
ts: 1519674281.220664736, inner:17, outer:15
ts: 1519674282.721666674, inner:18, outer:17
ts: 1519674283.722667966, inner:16, outer:17
ts: 1519674284.723669259, inner:17, outer:18
ts: 1519674285.724670551, inner:16, outer:17
ts: 1519674286.725671843, inner:17, outer:17
ts: 1519674287.726673136, inner:17, outer:16
ts: 1519674288.727674428, inner:16, outer:18
ts: 1519674289.728675721, inner:17, outer:17
ts: 1519674290.729677013, inner:18, outer:17----
```

This result shows that while doing consecutive reads of the system clocksource, there were 10 delays that showed up in the 15-18 us range.

hwlatdetect used the tracer mechanism to detect unexplained latencies.

NOTE

Previous versions used a kernel module rather than the **ftrace** tracer.

Understanding the results

The output shows the testing method, parameters, and results.

Table 3.1. Testing method, parameters, and results

Parameter	Value	Description
test duration	**10 seconds**	The duration of the test in seconds
detector	**tracer**	The utility that runs the **detector** thread
parameters		
Latency threshold	**10us**	The maximum allowable latency

Parameter	Value	Description
Sample window	**1000000us**	1 second
Sample width	**500000us**	1/2 second
Non-sampling period	**500000us**	1/2 second
Output File	**None**	The file to which the output is saved.
Results		
Max Latency	**18us**	The highest latency during the test that exceeded the **Latency threshold**. If no sample exceeded the **Latency threshold**, the report shows **Below threshold**.
Samples recorded	**10**	The number of samples recorded by the test.
Samples exceeding threshold	**10**	The number of samples recorded by the test where the latency exceeded the **Latency threshold**.
SMIs during run	**0**	The number of System Management Interrupts (SMIs) that occurred during the test run.

The detector thread runs a loop which does the following pseudocode:

```
t1 = timestamp()
 loop:
  t0 = timestamp()
  if (t0 - t1) > threshold
     outer = (t0 - t1)
  t1 = timestamp
  if (t1 - t0) > threshold
     inner = (t1 - t0)
  if inner or outer:
     print
  if t1 > duration:
     goto out
  goto loop
 out:
```

t0 is the timestamp at the start of each loop. **t1** is the timestamp at the end of each loop. The inner loop comparison checks that t0 - t1 does not exceed the specified threshold (10 us default). The outer loop

comparison checks the time between the bottom of the loop and the top t1 - t0. The time between consecutive reads of the timestamp register should be dozens of nanoseconds (essentially a register read, a comparison, and a conditional jump) so any other delay between consecutive reads is introduced by firmware or by the way the system components were connected.

NOTE

The values printed out by the **hwlatdetector** utility for inner and outer are the best case maximum latency. The latency values are the deltas between consecutive reads of the current system clocksource (usually the Time Stamp Counter or TSC register, but potentially the HPET or ACPI power management clock) and any delays between consecutive reads, introduced by the hardware-firmware combination.

After finding the suitable hardware-firmware combination, the next step is to test the real-time performance of the system while under a load.

CHAPTER 4. RUNNING AND INTERPRETING SYSTEM LATENCY TESTS

RHEL for Real Time provides the **rteval** utility to test the system real-time performance under load.

4.1. PREREQUISITES

- The **RHEL for Real Time** package group is installed.
- Root permissions for the system.

4.2. RUNNING SYSTEM LATENCY TESTS

You can run the **rteval** utility to test system real-time performance under load.

Prerequisites

- The **RHEL for Real Time** package group is installed.
- Root permissions for the system.

Procedure

- Run the **rteval** utility.

```
# rteval
```

The **rteval** utility starts a heavy system load of **SCHED_OTHER** tasks. It then measures real-time response on each online CPU. The loads are a parallel **make** of the Linux kernel tree in a loop and the **hackbench** synthetic benchmark.

The goal is to bring the system into a state, where each core always has a job to schedule. The jobs perform various tasks, such as memory allocation/free, disk I/O, computational tasks, memory copies, and other.

Once the loads have started up, **rteval** starts the **cyclictest** measurement program. This program starts the **SCHED_FIFO** real-time thread on each online core. It then measures the real-time scheduling response time.

Each measurement thread takes a timestamp, sleeps for an interval, then takes another timestamp after waking up. The latency measured is **t1 - (t0 + i)**, which is the difference between the actual wakeup time **t1**, and the theoretical wakeup time of the first timestamp **t0** plus the sleep interval **i**.

The details of the **rteval** run are written to an XML file along with the boot log for the system. This report is displayed on the screen and saved to a compressed file.

The file name is in the form **rteval-*<date>*-*N*-tar.bz2**, where ***<date>*** is the date the report was generated, ***N*** is a counter for the Nth run on ***<date>***.

The following is an example of an **rteval** report:

```
System:
Statistics:
```

```
  Samples:          1440463955
  Mean:            4.40624790712us
  Median:           0.0us
  Mode:            4us
  Range:            54us
  Min:             2us
  Max:              56us
  Mean Absolute Dev: 1.0776661507us
  Std.dev:          1.81821060672us

 CPU core 0       Priority: 95
 Statistics:
  Samples:          36011847
  Mean:            5.46434910711us
  Median:           4us
  Mode:            4us
  Range:            38us
  Min:             2us
  Max:              40us
  Mean Absolute Dev: 2.13785341159us
  Std.dev:          3.50155558554us
```

The report includes details about the system hardware, length of the run, options used, and the timing results, both per-cpu and system-wide.

NOTE

To regenerate an **rteval** report from its generated file, run

 # rteval --summarize rteval-*<date>*-*N*.tar.bz2

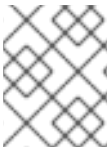
CHAPTER 5. SETTING PERSISTENT KERNEL TUNING PARAMETERS

When you have decided on a tuning configuration that works for your system, you can make the changes persistent across reboots.

By default, edited kernel tuning parameters only remain in effect until the system reboots or the parameters are explicitly changed. This is effective for establishing the initial tuning configuration. It also provides a safety mechanism. If the edited parameters cause the machine to behave erratically, rebooting the machine returns the parameters to the previous configuration.

5.1. MAKING PERSISTENT KERNEL TUNING PARAMETER CHANGES

You can make persistent changes to kernel tuning parameters by adding the parameter to the **/etc/sysctl.conf** file.



NOTE

This procedure does *not* change any of the kernel tuning parameters in the current session. The changes entered into **/etc/sysctl.conf** only affect future sessions.

Prerequisites

- Root permissions

Procedure

1. Open **/etc/sysctl.conf** in a text editor.
2. Insert the new entry into the file with the parameter's value.
   Modify the parameter name by removing the **/proc/sys/** path, changing the remaining slash (/) to a period (**.**), and including the parameter's value.

   For example, to make the command **echo 0 > /proc/sys/kernel/hung_task_panic** persistent, enter the following into **/etc/sysctl.conf**:

   ```
   # Enable gettimeofday(2)
   kernel.hung_task_panic = 0
   ```

3. Save and close the file.
4. Reboot the system for changes to take effect.

Verification

- To verify the configuration:

  ```
  ~]# cat /proc/sys/kernel/hung_task_panic
  0
  ```

CHAPTER 6. IMPROVING PERFORMANCE BY AVOIDING RUNNING UNNECESSARY APPLICATIONS

Every running application uses system resources. Ensuring that there are no unnecessary applications running on your system can significantly improve performance.

Prerequisites

- Root permissions for the system.

Procedure

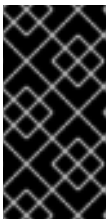
1. Do not run the **graphical interface** where it is not absolutely required, especially on servers. Check if the system is configured to boot into the GUI by default:

```
# systemctl get-default
```

2. If the output of the command is **graphical.target**, configure the system to boot to text mode:

```
# systemctl set-default multi-user.target
```

3. Unless you are actively using a **Mail Transfer Agent (MTA)** on the system you are tuning, disable it. If the MTA is required, ensure it is well-tuned or consider moving it to a dedicated machine.
   For more information, refer to the MTA's documentation.



IMPORTANT

MTAs are 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 the MTAs on your machine are disabled.

4. **Peripheral devices**, such as mice, keyboards, webcams send interrupts that may negatively affect latency. If you are not using a graphical interface, remove all unused peripheral devices and disable them.
   For more information, refer to the devices' documentation.

5. Check for automated **cron** jobs that might impact performance.

```
# crontab -l
```

   Disable the **crond** service or any unneeded **cron** jobs.

   For more information, refer to the **CRON(8)** man page.

6. Check your system for third-party applications and any components added by external hardware vendors, and remove any that are unnecessary.

CHAPTER 7. MINIMIZING OR AVOIDING SYSTEM SLOWDOWNS DUE TO JOURNALING

The order in which journal changes are written to disk may differ from the order in which they arrive. The kernel I/O system can reorder the journal changes to optimize the use of available storage space. Journal activity can result in system latency by re-ordering journal changes and committing data and metadata. As a result, journaling file systems can slow down the system.

XFS is the default filesystem used by RHEL 8. This is a journaling file system. An older file system called **ext2** does not use journaling. Unless your organization specifically requires journaling, consider using **ext2**. In many of Red Hat's best benchmark results, the **ext2** filesystem is used. This is one of the top initial tuning recommendations.

Journaling file systems like **XFS**, record the time a file was last accessed (**atime**). If you need to use a journaling file system, consider disabling **atime**.

7.1. DISABLING ATIME

Disabling **atime** increases performance and decreases power usage by limiting the number of writes to the filesystem journal.

Procedure

To disable **atime**:

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

```
/dev/mapper/rhel-root      /      xfs    defaults&hellip;
```

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

```
/dev/mapper/rhel-root      /      xfs    noatime,nodiratime&hellip;
```

IMPORTANT

Some applications rely on **atime** being updated. Therefore, this option is reasonable only on systems where such applications are not used.

Alternatively, you can use the **relatime** mount option, which ensures that the access time is only updated if the previous access time is older than the current modify time.

7.2. ADDITIONAL RESOURCES

- mkfs.ext2(8) man page
- mkfs.xfs(8) man page
- mount(8) man page

CHAPTER 8. DISABLING GRAPHICS CONSOLE OUTPUT FOR LATENCY SENSITIVE WORKLOADS

The kernel starts passing messages to **printk** as soon as it starts. The kernel sends messages to the log file and also displays on the graphics console even in the absence of a monitor attached to a headless server.

In some systems, the output sent to the graphics console might introduce stalls in the pipeline. This might cause potential delay in task execution while waiting for data transfers. For example, outputs sent to **teletype0 (/dev/tty0)**, might cause potential stalls in some systems.

To prevent unexpected stalls, you can limit or disable the information that is sent to the graphic console by:

- Removing the **tty0** definition.
- Changing the order of console definitions.
- Turning off most **printk** functions and ensuring that you set the **ignore_loglevel** to **not configured**.

This section includes procedures to prevent graphics console from logging on the graphics adapter and control the messages that print on the graphics console.

8.1. DISABLING GRAPHICS CONSOLE LOGGING TO GRAPHICS ADAPTER

teletype (tty), the default kernel console, enables your interaction with the system by passing input data to the system and displaying the output information on the graphics console.

Not configuring the graphics console, prevents it from logging on the graphics adapter. This makes **tty0** unavailable to the system and helps disable printing messages on the graphics console.

NOTE

Disabling graphics console output does not delete information. The information prints in the system log and you can access them using the **journalctl** or **dmesg** utilities.

Procedure

1. Open the **/etc/sysconfig/grub** file.
2. Remove the **console=tty0** value from the **GRUB_CMDLINE_LINUX** key.
3. Run the **grub2-mkconfig** command to re-generate the **/boot/grub2/grub.cfg** file:

```
# grub2-mkconfig -o /boot/sysconfig/grub2/grub.cfg
```

The **grub2-mkconfig** command collects the new configuration changes and re-generates the **/boot/grub2/grub.cfg** file.

8.2. DISABLING MESSAGES FROM PRINTING ON GRAPHICS CONSOLE

You can control the amount of output messages that are sent to the graphics console by configuring the required log levels in the **/proc/sys/kernel/printk** file.

Procedure

1. View the current console log level:

```
$ cat /proc/sys/kernel/printk
  7    4    1    7
```

   The command prints the current settings for system log levels. The numbers correspond to current, default, minimum, and boot-default values for the system logger.

2. Configure the desired log level in the **/proc/sys/kernel/printk** file.

```
$ echo “1” > /proc/sys/kernel/printk
```

   The command changes the current console log level. For example, setting log level 1, will print only alert messages and prevent display of other messages on the graphics console.

CHAPTER 9. MANAGING SYSTEM CLOCKS TO SATISFY APPLICATION NEEDS

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. To improve performance, you can change the clock source used to meet the minimum requirements of a real-time system.

9.1. HARDWARE CLOCKS

Multiple instances of clock sources found in multiprocessor systems, such as non-uniform memory access (NUMA) and Symmetric multiprocessing (SMP), 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 real-time kernel.

The preferred clock source is the Time Stamp Counter (TSC). If the TSC 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 use with the real-time kernel.

9.2. VIEWING THE AVAILABLE CLOCK SOURCES IN YOUR SYSTEM

The list of available clock sources in your system is in the **/sys/devices/system/clocksource/clocksource0/available_clocksource** file.

Procedure

- Display the **available_clocksource** file.

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

  In this example, the available clock sources in the system are TSC, HPET, and ACPI_PM.

9.3. VIEWING THE CLOCK SOURCE CURRENTLY IN USE

The currently used clock source in your system is stored in the **/sys/devices/system/clocksource/clocksource0/current_clocksource** file.

Procedure

- Display the **current_clocksource** file.

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

  In this example, the current clock source in the system is TSC.

9.4. TEMPORARILY CHANGING THE CLOCK SOURCE TO USE

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 real-time 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 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.

IMPORTANT

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

Prerequisites

- Root permissions on the system.

Procedure

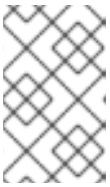
1. View the available clock sources.

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

   In this example, the available clock sources in the system are TSC, HPET, and ACPI_PM.

2. Write the name of the clock source you want to use to the **/sys/devices/system/clocksource/clocksource0/current_clocksource** file.

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



NOTE

This procedure changes the clock source currently in use. When the system reboots, the default clock is used. To make the change persistent, see Making persistent kernel tuning parameter changes.

Verification steps

- Display the **current_clocksource** file to ensure that the current clock source is the specified clock source.

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

   In this example, the current clock source in the system is HPET.

9.5. COMPARING THE COST OF READING HARDWARE CLOCK SOURCES

You can compare the speed of the clocks in your system. Reading from the TSC involves reading a register from the processor. Reading from the HPET clock involves 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.

Prerequisites

- Root permissions on the system.
- The **clock_timing** program must be on the system. For more information, see the clock_timing program.

Procedure

1. Change to the directory in which the **clock_timing** program is saved.

```
# cd clock_test
```

2. View the available clock sources in your system.

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

In this example, the available clock sources in the system are **TSC**, **HPET**, and **ACPI_PM**.

3. View the currently used clock source.

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

In this example, the current clock source in the system is **TSC**.

4. Run the **time** utility in conjunction with the ./ **clock_timing** program. The output displays the duration required to read the clock source 10 million times.

```
# time ./clock_timing

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

The example shows the following parameters:

- **real** - The total time spent beginning from program invocation until the process ends. **real** includes user and kernel 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.

5. Write the name of the next clock source you want to test to the **/sys/devices/system/clocksource/clocksource0/current_clocksource** file.

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

   In this example, the current clock source is changed to **HPET**.

6. Repeat steps 4 and 5 for all of the available clock sources.

7. Compare the results of step 4 for all of the available clock sources.

Additional resources

- **time(1)** man page

9.6. THE CLOCK_TIMING PROGRAM

The **clock_timing** program reads the current clock source 10 million times. In conjunction with the **time** utility it measures the amount of time needed to do this.

Procedure

To create the **clock_timing** program:

1. Create a directory for the program files.

```
$ mkdir clock_test
```

2. Change to the created directory.

```
$ cd clock_test
```

3. Create a source file and open it in a text editor.

```
$ vi clock_timing.c
```

4. Enter the following into the file:

```
#include <time.h>
void main()
{
 int rc;
 long i;
 struct timespec ts;

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

5. Save the file and exit the editor.
6. Compile the file.

```
$ gcc clock_timing.c -o clock_timing -lrt
```

The **clock_timing** program is ready and can be run from the directory in which it is saved.

CHAPTER 10. CONTROLLING POWER MANAGEMENT TRANSITIONS

You can control power management transitions to improve latency.

Prerequisites

- Root permissions for the system.

10.1. POWER SAVING STATES

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 real-time 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 behavior of the **idle=poll** and **processor.max_cstate=1** parameters, but with a more fine-grained control of power saving states. **idle=poll** prevents the processor from entering the **idle** state. **processor.max_cstate=1** prevents the processor from entering deeper C-states (energy-saving modes).

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

10.2. CONFIGURING POWER MANAGEMENT STATES

You can write a value to the **/dev/cpu_dma_latency** file to change the maximum response time for processes, in microseconds. You can also reference this file in an application or script.

Prerequisites

- Root permissions on the system.

Procedure

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 the file. This number represents a maximum response time in microseconds. For the fastest possible response time, use **0**.

Example

The following is an example of a program that uses this method to prevent power transitions and maintain low latency.

```
main()

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);
}
```

10.3. ADDITIONAL RESOURCES

- **Linux System Programming** by Robert Love.

CHAPTER 11. SETTING BIOS PARAMETERS FOR SYSTEM TUNING

This section contains information about various BIOS parameters that you can configure to improve system performance.

NOTE

Every system and BIOS vendor uses different terms and navigation methods. Therefore, this section contains only general information about BIOS settings.

If you need help locating a particular setting, check the BIOS documentation or contact the BIOS vendor.

11.1. DISABLING POWER MANAGEMENT TO IMPROVE RESPONSE TIMES

BIOS power management options help save power by changing the system clock frequency or by putting the CPU into one of various sleep states. These actions are likely to affect how quickly the system responds to external events.

To improve response times, disable all power management options in the BIOS.

11.2. IMPROVING RESPONSE TIMES BY DISABLING ERROR DETECTION AND CORRECTION UNITS

Error Detection and Correction (EDAC) units are devices for detecting and correcting 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 the BIOS uses. This may result in missing crucial event deadlines.

To improve response times, turn off EDAC. If this is not possible, configure EDAC to the lowest functional level.

11.3. IMPROVING RESPONSE TIME BY CONFIGURING SYSTEM MANAGEMENT INTERRUPTS

System Management Interrupts (SMIs) are a hardware vendors facility to ensure that the system is operating correctly. The BIOS code usually services the SMI interrupt. 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 determine the extent to which it is safe to disable them.

WARNING

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

CHAPTER 12. MINIMIZING SYSTEM LATENCY BY ISOLATING INTERRUPTS AND USER PROCESSES

Real-time environments need to minimize or eliminate latency when responding to various events. To do this, you can isolate interrupts (IRQs) from user processes from one another on different dedicated CPUs.

12.1. INTERRUPT AND PROCESS BINDING

Isolating interrupts (IRQs) from user processes on different dedicated CPUs can minimize or eliminate latency in real-time environments.

Interrupts are generally shared evenly between CPUs. This can delay interrupt processing when the CPU has to write new data and instruction caches. These interrupt delays can cause conflicts with other processing being performed on the same CPU.

It is possible to allocate time-critical interrupts and processes to a specific CPU (or a range of CPUs). In this way, the code and data structures for processing this interrupt will most likely be in the processor and instruction caches. As a result, the dedicated process can run as quickly as possible, while all other non-time-critical processes run on the other CPUs. This can be particularly important where the speeds involved are near or at the limits of memory and available peripheral bus bandwidth. 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, tuning applications with similar functions for different companies, required completely different optimal performance tunings.

- One firm saw optimal results when they isolated 2 out of 4 CPUs for operating system functions and interrupt handling. The remaining 2 CPUs were dedicated purely for application handling.
- Another firm found optimal determinism when they bound the network related application processes onto a single CPU which was handling the network device driver interrupt.

IMPORTANT

To bind a process to a CPU, you usually 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, a decimal number, or a hexadecimal number, depending on the command you are using.

Table 12.1. Examples

CPUs	Bitmask	Decimal	Hexadecimal
0	00000000000000000000000000000001	1	0x00000001
0, 1	00000000000000000000000000000011	3	0x00000011

12.2. DISABLING THE IRQBALANCE DAEMON

The **irqbalance** daemon is enabled by default and periodically forces interrupts to be handled by CPUs in an even manner. However in real-time deployments, **irqbalance** is not needed, because applications are typically bound to specific CPUs.

Procedure

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

```
# systemctl status irqbalance
irqbalance.service - irqbalance daemon
   Loaded: loaded (/usr/lib/systemd/system/irqbalance.service; enabled)
   Active: active (running) …
```

2. If **irqbalance** is running, disable it, and stop it.

```
# systemctl disable irqbalance
# systemctl stop irqbalance
```

Verification

- Check that the **irqbalance** status is inactive.

```
~]# systemctl status irqbalance
```

12.3. EXCLUDING CPUS FROM IRQ BALANCING

You can use the IRQ balancing service to specify which CPUs you want to exclude from consideration for interrupt (IRQ) balancing. The **IRQBALANCE_BANNED_CPUS** parameter in the **/etc/sysconfig/irqbalance** configuration file controls these settings. The value of the parameter is a 64-bit hexadecimal bit mask, where each bit of the mask represents a CPU core.

Procedure

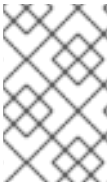
1. Open **/etc/sysconfig/irqbalance** in your preferred text editor and find the section of the file titled **IRQBALANCE_BANNED_CPUS**.

```
# IRQBALANCE_BANNED_CPUS
# 64 bit bitmask which allows you to indicate which cpu's should
# be skipped when reblancing irqs. Cpu numbers which have their
# corresponding bits set to one in this mask will not have any
# irq's assigned to them on rebalance
#
#IRQBALANCE_BANNED_CPUS=
```

2. Uncomment the **IRQBALANCE_BANNED_CPUS** variable.

3. Enter the appropriate bitmask to specify the CPUs to be ignored by the IRQ balance mechanism.

4. Save and close the file.



NOTE

If you are running a system with up to 64 CPU cores, separate each group of eight hexadecimal digits with a comma. For example:
IRQBALANCE_BANNED_CPUS=00000001,0000ff00

Table 12.2. Examples

CPUs	Bitmask
0	00000001
8 - 15	0000ff00
8 - 15, 33	00000001,0000ff00

NOTE

In RHEL 7.2 and higher, the **irqbalance** utility automatically avoids IRQs on CPU cores isolated via the **isolcpus** kernel parameter if **IRQBALANCE_BANNED_CPUS** is not set in **/etc/sysconfig/irqbalance**.

12.4. MANUALLY ASSIGNING CPU AFFINITY TO INDIVIDUAL IRQS

Assigning CPU affinity enables binding and unbinding processes and threads to a specified CPU or range of CPUs. This can reduce caching problems.

Procedure

1. Check the IRQs in use by each device by viewing the **/proc/interrupts** file.

```
~]# cat /proc/interrupts
```

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
```

2. Write the CPU mask to the **smp_affinity** entry of a specific IRQ. The CPU mask must be expressed as a hexadecimal number.
For example, the following command instructs IRQ number 142 to run only on CPU 0.

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

The change only takes effect when an interrupt occurs.

Verification steps

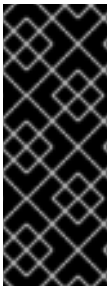
1. Perform an activity that will trigger the specified interrupt.

2. Check **/proc/interrupts** for changes.
   The number of interrupts on the specified CPU for the configured IRQ increased, and the number of interrupts for the configured IRQ on CPUs outside the specified affinity did not increase.

12.5. BINDING PROCESSES TO CPUS WITH THE TASKSET UTILITY

The **taskset** utility uses the process ID (PID) of a task to view or set its CPU affinity. You can use the utility to launch a command with a chosen CPU affinity.

To set the affinity, you need to get the CPU mask to be 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.



IMPORTANT

The taskset utility works on a NUMA (Non-Uniform Memory Access) system, but it does not allow the user to bind threads to CPUs and the closest NUMA memory node. On such systems, taskset is not the preferred tool, and the **numactl** utility should be used instead for its advanced capabilities.

For more information, see the **numactl(8)** man page.

Procedure

- Run **taskset** with the necessary options and arguments.
  - You can specify a CPU list using the -c parameter instead of a CPU mask. In this example, **my_embedded_process** is being instructed to run only on CPUs 0,4,7-11.

    ```
    ~]# taskset -c 0,4,7-11 /usr/local/bin/my_embedded_process
    ```

    This invocation is more convenient in most cases.

  - 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
    ```

  - You can 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
    ```

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

    ```
    ~]# taskset -p 1 7013
    ```

NOTE

You can combine the listed options.

Additional resources

- **taskset(1)** man page
- **numactl(8)** man page

CHAPTER 13. MANAGING OUT OF MEMORY STATES

Out of Memory (OOM) refers to a computing state where all available memory, including swap space, has been allocated. Normally this causes the system to panic and stop functioning as expected.

The following provides instructions for avoiding OOM states on your system.

13.1. PREREQUISITES

- Root permissions on the system.

13.2. CHANGING THE OUT OF MEMORY VALUE

The **/proc/sys/vm/panic_on_oom** file contains a value which is the switch that controls Out of Memory (OOM) behavior. When the file contains **1**, the kernel panics on OOM and stops functioning as expected.

The default value is **0**, which instructs the kernel to call the **oom_killer** function when the system is in an OOM state. Usually, **oom_killer** terminates unnecessary processes, which allows the system to survive.

You can change the value of **/proc/sys/vm/panic_on_oom**.

Procedure

1. Display the current value of **/proc/sys/vm/panic_on_oom**.

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

   To change the value in **/proc/sys/vm/panic_on_oom**:

2. Echo the new value to **/proc/sys/vm/panic_on_oom**.

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

NOTE

It is recommended that you make the Real-Time kernel panic on OOM (**1**). Otherwise, when the system encounters an OOM state, it is no longer deterministic.

Verification steps

1. Display the value of **/proc/sys/vm/panic_on_oom**.

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

2. Verify that the displayed value matches the value specified.

13.3. PRIORITIZING PROCESSES TO KILL WHEN IN AN OUT OF MEMORY STATE

You can prioritize the processes that get terminated by the **oom_killer** function. This can ensure that high-priority processes keep running during an OOM state. Each process has a directory, /**proc**/***PID***. Each directory includes the following files:

- **oom_adj** - Valid scores for **oom_adj** are in the range -16 to +15. This value is used to calculate the performance footprint of the process, using an algorithm that also takes into account how long the process has been running, among other factors.
- **oom_score** - Contains the result of the algorithm calculated using the value in **oom_adj**.

In an Out of Memory state, the **oom_killer** function terminates processes with the highest **oom_score**.

You can prioritize the processes to terminate by editing the **oom_adj** file for the process.

Prerequisites

- Know the process ID (PID) of the process you want to prioritize.

Procedure

1. Display the current **oom_score** for a process.

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

2. Display the contents of **oom_adj** for the process.

```
# cat /proc/12465/oom_adj
13
```

3. Edit the value in **oom_adj**.

```
# echo -5 > /proc/12465/oom_adj
```

Verification steps

1. Display the current **oom_score** for the process.

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

2. Verify that the displayed value is lower than the previous value.

13.4. DISABLING THE OUT OF MEMORY KILLER FOR A PROCESS

You can disable the **oom_killer** function for a process by setting **oom_adj** to the reserved value of **-17** This will keep the process alive, even in an OOM state.

Procedure

1. Set the value in **oom_adj** to **-17**.

```
# echo -17 > /proc/12465/oom_adj
```

Verification steps

1. Display the current **oom_score** for the process.

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

2. Verify that the displayed value is **0**.

CHAPTER 14. LOWERING CPU USAGE BY DISABLING THE PC CARD DAEMON

The **pcscd** daemon manages connections to parallel communication (PC or PCMCIA) and smart card (SC) 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 real-time tasks and other undesirable impacts on determinism.

Prerequisites

- Root permissions on the system.

Procedure

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

```
# systemctl status pcscd
● pcscd.service - PC/SC Smart Card Daemon
     Loaded: loaded (/usr/lib/systemd/system/pcscd.service; indirect; vendor preset: disabled)
     Active: active (running) since Mon 2021-03-01 17:15:06 IST; 4s ago
TriggeredBy: ● pcscd.socket
       Docs: man:pcscd(8)
   Main PID: 2504609 (pcscd)
      Tasks: 3 (limit: 18732)
     Memory: 1.1M
        CPU: 24ms
     CGroup: /system.slice/pcscd.service
             └─2504609 /usr/sbin/pcscd --foreground --auto-exit
```

   The **Active** parameter shows the status of the **pcsd** daemon.

2. If the **pcsd** daemon is running, stop it.

```
# systemctl stop pcscd
Warning: Stopping pcscd.service, but it can still be activated by:
  pcscd.socket
```

3. Configure the system to ensure that the **pcsd** daemon does not restart when the system boots

```
# systemctl disable pcscd
Removed /etc/systemd/system/sockets.target.wants/pcscd.socket.
```

Verification steps

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

```
# systemctl status pcscd
● pcscd.service - PC/SC Smart Card Daemon
     Loaded: loaded (/usr/lib/systemd/system/pcscd.service; indirect; vendor preset: disabled)
     Active: inactive (dead) since Mon 2021-03-01 17:10:56 IST; 1min 22s ago
TriggeredBy: ● pcscd.socket
```

```
     Docs: man:pcscd(8)
Main PID: 4494 (code=exited, status=0/SUCCESS)
     CPU: 37ms
```

2. Ensure that the value for the **Active** parameter is **inactive (dead)**.

CHAPTER 15. BALANCING LOGGING PARAMETERS

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

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

Procedure

To enable remote logging:

1. Configure the machine to which the logs will be sent. For more information, see Remote Syslogging with rsyslog on Red Hat Enterprise Linux.

2. Configure each system that will send logs to the remote log server, so that its **syslog** output is written to the server, rather than to the local file system. To do so, edit the **/etc/rsyslog.conf** file on each client system. For each of the logging rules defined in that file, 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 configures the client system to log all kernel messages to the remote machine at **@my.remote.logging.server**.

   Alternatively, you can configure **syslogd** to log all locally generated system messages, by adding the following line to the **/etc/rsyslog.conf** file:

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

IMPORTANT

The **syslogd** daemon does not include built-in rate limiting on its generated network traffic. Therefore, Red Hat recommends that when using RHEL for Real Time systems, only log messages that are required to be remotely logged by your organization. For example, kernel warnings, authentication requests, and the like. Other messages should be logged locally.

Additional resources

- syslog(3) man page
- rsyslog.conf(5) man page
- rsyslogd(8) man page

CHAPTER 16. IMPROVING LATENCY USING THE TUNA CLI

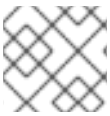
You can use the **tuna** CLI to improve latency on your system. The options used with the **tuna** command determine the method invoked to improve latency.

16.1. PREREQUISITES

- The **RHEL for Real Time** package group and the **tuna** package are installed.
- Root permissions for the system.

16.2. THE TUNA CLI

The **tuna** command-line interface (CLI) is a tool to help you make tuning changes to your system.



NOTE

A new graphical interface is being developed for **tuna**, but it has not yet been released.

The **tuna** CLI can be used to adjust scheduler tunables, tune thread priority, IRQ handlers, and isolate CPU cores and sockets. **tuna** aims to reduce the complexity of performing tuning tasks. 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 changes have been made.

The **tuna** CLI has both action options and modifier options. Modifier options must be specified on the command-line before the actions they are intended to modify. All modifier options apply to the actions that follow until the modifier options are overridden.

16.3. ISOLATING CPUS USING THE TUNA CLI

You can use the **tuna** CLI to isolate interrupts (IRQs) from user processes on different dedicated CPUs to minimize latency in real-time environments. For more information about isolating CPUs, see Interrupt and process binding.

Prerequisites

- The **RHEL for Real Time** package group and the **tuna** package are installed.
- Root permissions for the system.

Procedure

- Isolate one or more CPUs.

```
# tuna --cpus=cpu_list --isolate
```

where ***cpu_list*** is a comma-separated list of the CPUs to isolate.

For example:

```
# tuna --cpus=0,1 --isolate
```

16.4. MOVING INTERRUPTS TO SPECIFIED CPUS USING THE TUNA CLI

You can use the **tuna** CLI to move interrupts (IRQs) to dedicated CPUs to minimize or eliminate latency in real-time environments. For more information about moving IRQs, see Interrupt and process binding.

Prerequisites

- The **RHEL for Real Time** package group and the **tuna** package are installed.
- Root permissions for the system.

Procedure

1. List the CPUs to which a list of IRQs is attached.

   ```
   # tuna --irqs=irq_list --show_irqs
   ```

   where ***irq_list*** is a comma-separated list of the IRQs for which you want to list attached CPUs.

   For example:

   ```
   # tuna --irqs=128 --show_irqs
     # users         affinity
   128 iwlwifi         0,1,2,3
   ```

2. Attach a list of IRQs to a list of CPUs.

   ```
   # tuna --irqs=irq_list --cpus=cpu_list --move
   ```

   where ***irq_list*** is a comma-separated list of the IRQs you want to attach and ***cpu_list*** is a comma-separated list of the CPUs to which they will be attached.

   For example:

   ```
   # tuna --irqs=128 --cpus=3 --move
   ```

Verification steps

- Compare the state of the selected IRQs before and after moving any IRQ to a specified CPU.

  ```
  # tuna --irqs=128 --show_irqs
    # users         affinity
  128 iwlwifi         3
  ```

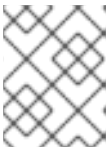
16.5. CHANGING PROCESS SCHEDULING POLICIES AND PRIORITIES USING THE TUNA CLI

You can use the **tuna** CLI to change process scheduling policy and priority.

Prerequisites

- The **RHEL for Real Time** package group and the **tuna** package are installed.

- Root permissions for the system.



NOTE

Assigning the OTHER and NATCH scheduling policies does not require root permissions.

Procedure

1. View the information for a thread.

```
# tuna --threads=thread_list --show_threads
```

where ***thread_list*** is a comma-separated list of the processes you want to display.

For example:

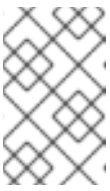
```
# tuna --threads=rngd --show_threads
                    thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
 3571   OTHER     0  0,1,2,3    167697          134           rngd
```

2. Modify the process scheduling policy and the priority of the thread.

```
# tuna --threads=thread_list --priority scheduling_policy:priority_number
```

where:

- ***thread_list*** is a comma-separated list of the processes whose scheduling policy and priority you want to display.
- ***scheduling_policy*** is one of the following:
  - OTHER
  - BATCH
  - **FIFO** - First In First Out
  - **RR** - Round Robin
- ***priority_number*** is a priority number from 0 to 99, where **0** is no priority and **99** is the highest priority.



NOTE

The **OTHER** and **BATCH** scheduling policies do not require specifying a priority. In addition, the only valid priority (if specified) is **0**. The **FIFO** and **RR** scheduling policies require a priority of **1** or more.

For example:

```
# tuna --threads=rngd --priority FIFO:1
```

Verification steps

- View the information for the thread to ensure that the information changes.

```
# tuna --threads=rngd --show_threads
                    thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
 3571   FIFO     1  0,1,2,3    167697          134            rngd
```

CHAPTER 17. INSTALLING KDUMP AND KEXEC

In many cases, the **kdump** service is installed and activated by default on RHEL installations.

The following provides information about how to install and activate **kdump** when it is not enabled by default.

17.1. PREREQUISITES

- Administrator privileges

17.2. KDUMP AND KEXEC

The **kdump** service provides a crash dumping mechanism. The service enables you to save the contents of the system's memory for later analysis. **kdump** uses the **kexec** system call to boot into a second kernel (a capture kernel) without rebooting. Then **kdump** captures the contents of the crashed kernel's memory (a crash dump or a vmcore) and saves it. The second kernel resides in a reserved part of the system memory.

You can enable **kdump** for all installed kernels on a machine or only for specified kernels. This is useful when there are multiple kernels used on a machine, some of which are stable enough that there is no concern that they might crash.

A kernel crash dump might be the only information available in the event of a system failure (a critical bug). Therefore, ensuring that **kdump** is operational is important in mission-critical environments. Red Hat advises that system administrators regularly update and test kexec-tools during your normal kernel update cycle. This is especially important when new kernel features are implemented.

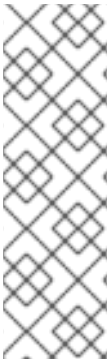
When **kdump** is installed, a default **/etc/kdump.conf** file is created. The file includes the default minimum **kdump** configuration. You *can* edit this file to customize the **kdump** configuration, but it is not required.

The following is the high level procedure for installing, configuring, enabling, and starting **kdump**:

1. (Optional) Edit **kdump.conf** to configure **kdump**.
2. Add the **kdump** kernel to the **grub** configuration.
3. Enable **kdump**.
4. Reboot the machine to start **kdump**.

17.3. INSTALLING KDUMP AND KEXEC

You can install **kdump** and **kexec** from the command line.



NOTE

The **Anaconda** installer includes a screen for **kdump** configuration when performing an interactive installation using the graphical or text interface. The installer screen is titled Kdump and is available from the main Installation Summary screen and only allows limited configuration. You can only select whether **kdump** is enabled and how much memory is reserved.

Some installation options, such as custom **Kickstart** installations, in some cases do not install or enable **kdump** by default.

Prerequisites

- An active RHEL subscription
- A repository containing the **kexec-tools** package for your system CPU architecture
- Fulfilled requirements for **kdump** configurations and targets

Procedure

1. Check if **kdump** is installed on your system.

```
$ rpm -q kexec-tools
```

   If the package is already installed, the output shows the version number of the **kexec-tools** package.

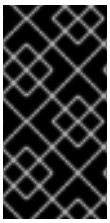
```
kexec-tools-2.0.17-11.el8.x86_64
```

   If the package is not installed, the output shows the following:

```
package kexec-tools is not installed
```

2. Install **kdump** and the necessary tools.

```
# yum install kexec-tools
```



IMPORTANT

Starting with RHEL 7.4 (kernel-3.10.0-693.el7) the Intel IOMMU driver is supported with kdump. For prior versions, RHEL 7.3 (kernel-3.10.0-514[.XYZ].el7) and earlier, it is advised that Intel IOMMU support is disabled. Otherwise, the **kdump** kernel is likely to become unresponsive.

Verification

1. Check if **kdump** is installed on your system.

```
$ rpm -q kexec-tools
kexec-tools-2.0.17-11.el8.x86_64
```

CHAPTER 18. ENSURING THAT DEBUGFS IS MOUNTED

The **debugfs** file system is specially designed for debugging and making information available to users. It is mounted automatically in RHEL 8 in the /**sys**/**kernel**/**debug**/ directory.

NOTE

The **debugfs** file system is mounted using the **ftrace** and **trace-cmd** commands.

Procedure

To verify that **debugfs** is mounted:

- Run the following command:

```
# mount | grep ^debugfs
debugfs on /sys/kernel/debug type debugfs (rw,nosuid,nodev,noexec,relatime,seclabel)
```

If **debugfs** is mounted, the command displays the mount point and properties for **debugfs**.

If **debugfs** is not mounted, the command returns nothing.

CHAPTER 19. CREATING A BASIC DUMP KERNEL

The memory for the [service] **kdump** service is reserved when the system boots. The memory size is configured in the system's Grand Unified Bootloader (GRUB) 2 configuration file. The memory size depends on the **crashkernel=** ***value*** specified in the configuration file and the size of the system's physical memory.

Prerequisites

- Administrator privileges.

19.1. KDUMP AND KEXEC

The **kdump** service provides a crash dumping mechanism. The service enables you to save the contents of the system's memory for later analysis. **kdump** uses the **kexec** system call to boot into a second kernel (a capture kernel) without rebooting. Then **kdump** captures the contents of the crashed kernel's memory (a crash dump or a vmcore) and saves it. The second kernel resides in a reserved part of the system memory.

You can enable **kdump** for all installed kernels on a machine or only for specified kernels. This is useful when there are multiple kernels used on a machine, some of which are stable enough that there is no concern that they might crash.

A kernel crash dump might be the only information available in the event of a system failure (a critical bug). Therefore, ensuring that **kdump** is operational is important in mission-critical environments. Red Hat advises that system administrators regularly update and test kexec-tools during your normal kernel update cycle. This is especially important when new kernel features are implemented.

When **kdump** is installed, a default **/etc/kdump.conf** file is created. The file includes the default minimum **kdump** configuration. You *can* edit this file to customize the **kdump** configuration, but it is not required.

The following is the high level procedure for installing, configuring, enabling, and starting **kdump**:

1. (Optional) Edit **kdump.conf** to configure **kdump**.
2. Add the **kdump** kernel to the **grub** configuration.
3. Enable **kdump**.
4. Reboot the machine to start **kdump**.

19.2. CONFIGURING KDUMP MEMORY USAGE

The **crashkernel** option can be defined in a number of ways. You can specify the **crashkernel=*value*** or configure the **auto** option.

The **crashkernel=auto** boot option reserves memory automatically, depending on the total amount of the system's physical memory. When configured, the kernel automatically reserves an appropriate amount of required memory for the kdump kernel. This helps to prevent Out-of-Memory (OOM) errors.

NOTE

The automatic memory allocation for **kdump** varies based on system hardware architecture and available memory size.

For example, on systems with AMD64 and Intel 64 processors configured with the **crashkernel=auto** parameter, **kdump** memory is only allocated when the available memory is more than 1GB. On systems with 64-bit ARM architecture and IBM Power Systems configured with the **crashkernel=auto** parameter, **kdump** memory is only allocated when the available memory is more than 2GB.

If the system has less than the minimum memory threshold for automatic allocation, you can configure the amount of reserved memory manually.

Use the following procedure to manually configure the amount of memory to reserve for **kdump**.

Prerequisites

- The necessary **kdump** memory and targets.
  For more information on the necessary memory and targets, see the sections below.

Procedure

1. Edit the **/etc/default/grub** file using root permissions.

2. Set the **crashkernel** option to the required value.
   For example, to reserve 128 MB of memory:

   ```
   crashkernel=128M
   ```

   NOTE

   You can set the amount of reserved memory to a variable depending on the total amount of installed memory. The syntax for memory reservation into a variable is crashkernel=*<range1>:<size1>,<range2>:<size2>*.

   For example: **crashkernel=512M-2G:64M,2G-:128M** reserves 64 MB of memory if the total amount of system memory is 512 MB or more and less than 2 GB. If the total amount of system memory is more than 2 GB, 128 MB is reserved for **kdump** instead.

3. (Optional) Offset the reserved memory.
   Some systems require that **kdump** memory is reserved with a fixed offset. This is because the **crashkernel** reservation occurs very early in the boot, and the system needs to reserve some memory for special usage. If an offset is configured, the reserved memory begins there.

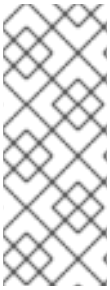
   To offset the reserved memory:

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

   This example reserves 128 MB of **kdump** memory offset by 16 MB (at physical address 0x01000000). If the offset parameter is set to **0** or omitted entirely, **kdump** offsets the

reserved memory automatically. You can also use this syntax when setting a variable memory reservation as described above. In this case, the offset is always specified last. For example, **crashkernel=512M-2G:64M,2G-:128M@16M)**.

4. Update the GRUB2 configuration file.



NOTE

An alternative way to configure memory for **kdump** is to append the **crashkernel=*<SOME_VALUE>*** parameter to the **kernelopts** variable with the **grub2-editenv** command. This updates all of your boot entries.

Alternatively, you can use the **grubby** utility to update the kernel command line parameters of just one entry.

Additional resources

- Configuring kernel command-line parameters
- The grub2-mkconfig script silently ignores options in GRUB_CMDLINE_LINUX
- How to manually modify the boot parameter in grub before the system boots
- How to install and boot custom kernels in Red Hat Enterprise Linux 8
- **grubby(8)** man page

19.3. CONFIGURING THE KDUMP TARGET

When a kernel crash is captured, the core dump can be stored as a file in one of the following locations:

- To the local file system
- Directly to a device
- Sent over a network using the NFS (Network File System) or the SSH (Secure Shell) protocol

NOTE

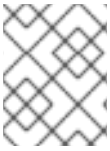
Only one of these options can be set at a time.

By default, the **vmcore** file is stored in the **/var/crash/** directory of the local file system.

If you do not specify a dump target in the **/etc/kdump.conf** file, then the path represents the absolute path from the root directory. Depending on what is mounted in the current system, the dump target and the adjusted dump path are taken automatically.

Prerequisites

- The necessary **kdump** memory and targets



NOTE

The specified target directory must exist when the **kdump systemd** service starts. Otherwise, the system fails.

Procedure

This section provides information on configuring the **kdump** target to each of the available targets.

To a directory on the local file system

To specify a directory to which the crash dump will be saved:

1. Open the **/etc/kdump.conf** file in a text editor as administrator.

```
$ vi /etc/kdump.conf
```

2. Delete the hash sign (**#**) at the beginning of the line that specifies the mounted system device on which the target directory resides.

3. Change **ext4** to the file system type, if necessary.

4. Specify the device using one of the following:

   - The device name (for example, **/dev/vg/lv_kdump**)
   - The file system label (for example: **LABEL=/boot**)
   - The UUID (for example, **UUID=03138356-5e61-4ab3-b58e-27507ac41937**)

IMPORTANT

It is recommended that you specify storage devices using a **LABEL** or **UUID**. Disk device names such as **/dev/sda3** may not be consistent across reboots.

IMPORTANT

When dumping to Direct Access Storage Device (DASD) on IBM Z hardware, you must ensure that the dump devices are correctly specified in **/etc/dasd.conf** before proceeding.

5. Delete the hash sign from the beginning of the **#path /var/crash** line.

6. Change **/var/crash** to the sub-directory where you want to save the crash dump.

NOTE

If using the default system device, the path specified is relative to the **/var/crash** directory.

Therefore, if the **ext4** file system is already mounted at **/var/crash** and the path is set as **/var/crash**, the crash dump will be saved in the **/var/crash/var/crash** directory,

To save the crash dump in the **/var/crash** directory, specify the path as **/**.

7. Save and close the file.

Directly to a device

To specify a device to which the crash dump will be saved directly:

1. Open the **/etc/kdump.conf** file in a text editor as administrator.

```
$ vi /etc/kdump.conf
```

2. Delete the hash sign (**#**) at the beginning of the line **#raw /dev/vg/lv_kdump**.
3. Replace the value with the intended device name.
   For example: **raw /dev/sdb1**
4. Save and close the file.

To a remote machine using the NFS protocol

To store the dump to a remote machine using the NFS protocol:

1. Open the **/etc/kdump.conf** file in a text editor as administrator.

```
$ vi /etc/kdump.conf
```

2. Delete the hash sign (**#**) at the beginning of the line **#nfs my.server.com:/export/tmp**.
3. Replace the value with a valid hostname and directory path.
   For example: **nfs penguin.example.com:/export/cores**
4. Save and close the file.

To a remote machine using the SSH protocol

To store the dump to a remote machine using the SSH protocol:

1. Open the **/etc/kdump.conf** file in a text editor as administrator.

```
$ vi /etc/kdump.conf
```

2. Delete the hash sign (**#**) at the beginning of the line **#ssh user@my.server.com**.
3. Replace the value with a valid username and hostname.
4. Delete the hash sign (**#**) at the beginning of the line **#sshkey /root/.ssh/kdump_id_rsa**.
5. Change **kdump_id_rsa** to the location of a key valid on the server you are trying to dump to.
   For example:

```
ssh john@penguin.example.com
sshkey /root/.ssh/mykey
```

6. Save and close the file.

Additional resources

- Supported kdump targets
- Configuring basic system settings

19.4. CONFIGURING THE KDUMP CORE COLLECTOR

The **kdump** service uses a **core_collector** program to capture the vmcore image. In RHEL, the **makedumpfile** utility is the default core collector. It helps shrink the dump file by:

- Compressing the size of a dump file using the **zlib**, **lzo**, or **snappy** utility.
- Excluding unnecessary pages by specifying the pages to exclude from the dump by using kernel debug information to determine the kernel use of memory.
- Filtering the page types to be included in the dump.

Prerequisites

- The necessary **kdump** memory and targets

Procedure

1. Open the **/etc/kdump.conf** file in a text editor as administrator.

```
$ vi /etc/kdump.conf
```

2. Delete the hash sign (**#**) at the beginning of the line that starts **core_collector makedumpfile**.
3. Edit the options in the line to specify the contents and compression method to use for the dump.
   For more information about the available options, see *The kdump configuration file* .
4. Save and close the file.

Verification

1. View the **/etc/kdump.conf** file.
   cat /etc/kdump.conf
2. Ensure that the line that starts **core_collector makedumpfile** contains the specified options.

Additional resources

- **makedumpfile**(8) man page

19.5. CONFIGURING THE KDUMP DEFAULT FAILURE RESPONSES

By default, when **kdump** fails to create a **vmcore** file at the configured target location, the system reboots and the dump is lost in the process. You can change this behavior.

Prerequisites

- The necessary **kdump** memory and targets

Procedure

1. Open the **/etc/kdump.conf** file in a text editor as administrator.

```
$ vi /etc/kdump.conf
```

2. Delete the hash sign (**#**) at the beginning of the line that starts **failure_action**.

3. Edit the options in the line to specify the contents and compression method to use for the dump.
   For more information about the available options, see *The kdump configuration file* .

4. Save and close the file.

Verification

1. View the **/etc/kdump.conf** file.
   cat /etc/kdump.conf

2. Ensure that the line that starts **core_collector makedumpfile** contains the specified options.

19.6. THE KDUMP CONFIGURATION FILE

The **kdump** configuration file, **/etc/kdump.conf**, contains options and commands for the kernel crash dump.

The first part of the file provides comments explaining the available options and commands. The second part of the file includes a default configuration. Options that are not in the default configuration are commented out using a hash mark at the start of each option. This makes it easy to modify the file correctly.

NOTE

The information here includes only some of the options that can be configured in this file.

kdump.conf configuration options

Memory to reserve

The **crashkernel** parameter defines the amount of memory reserved for the kernel crash dump. The following options are available:

- An absolute value in megabytes
  For example: **crashkernel=128M** for 128 megabytes of reserved memory.

- Some systems require that **kdump** memory is reserved with a fixed offset. This is because the **crashkernel** reservation is very early in the boot, and the system needs to reserve some memory for special usage. If an offset is configured, the reserved memory begins there.
  For example, **crashkernel=128M@16M** for 128 megabytes of reserved memory offset by 16 megabytes

- Variable amounts. The amount of memory reserved is based on the amount of memory in the system.

For example, **crashkernel=512M-2G:64M,2G-:128M@16M** for reserving 64 megabytes in a system with between 1/2 a megabyte and two gigabybtes of memory and 128 megabytes for systems with more than two gigabybtes of memory.

NOTE

You can combine variable amounts with offsets.

For example, **crashkernel=512M-2G:64M,2G-:128M@16M**.

- **auto** - Automatically allocates memory for the crash kernel dump based on the system hardware architecture and available memory size.
  For example, on systems with AMD64 and Intel 64 processors configured with the **crashkernel=auto** parameter, **kdump** memory is only allocated when the available memory is more than 1GB. On systems with 64-bit ARM architecture and IBM Power Systems configured with the **crashkernel=auto** parameter, **kdump** memory is only allocated when the available memory is more than 2GB.

  If the system has less than the minimum memory threshold for automatic allocation, you can configure the amount of reserved memory manually.

Target

The location where the kernel crash dump will be saved. The following options are available:

- **raw** - Defines a device to which the kernel crash dump will be sent. Use persistent device names for partition devices, such as **/dev/vg/*<devname>***.
- **path** - Defines the device, file system type, and the path to a directory on the local file system. You can specify the device using the device name (for example, **/dev/vg/lv_kdump**), file system label (for example, **LABEL=/boot**), or the UUID (for example, **UUID=03138356-5e61-4ab3-b58e-27507ac41937**).
- **nfs** - Defines an NFS target with a hostname and directory path. For example, **nfs penguin.example.com:/export/cores**.
- **ssh** - Defines an SSH target (for example, **ssh *john@penguin.example.com***. The **sshkey** variable defines the location of the SSH key on the server.

Shrinking the dump file

The **makedumpfile** utility is a dump program that helps shrink the dump file using the following methods:

- Compressing the size of a dump file using one of the following options:
  - **-c** - Compresses the file using the **zlib** utility
  - **-l** - Compresses the file using **lzo** utility
  - **-p** - Compresses the file using the **snappy** utility
- Excluding unnecessary pages by using the **-d** option and specifying the pages to exclude. **makedumpfile** needs the first kernel debug information to understand how first kernel uses the memory. This helps it detect the pages that are needed for the dump.

- Filtering the pages to be included in the dump using the **--message-level** option and specifying the page types to include by adding the following filtering options:
  - **1** - zero pages
  - **2** - cache pages
  - **4** - cache private pages
  - **8** - user pages
  - **16** - free pages
    For example, to specify that only cache pages, cache private pages, and user pages are included in the dump, specify **--message-level 14** (2 + 4 + 8).

NOTE

The **makedumpfile** command supports removal of transparent **huge pages** and **hugetlbfs** pages from RHEL 7.3 and later. Consider both these types of pages user pages and remove them using the **-8** option.

The kdump default failure response

When **kdump** fails to create a core dump, the default failure response of the operating system is to reboot. However, you can configure the **kdump** utility to perform a different operation in case it fails to save the core dump to the primary target.
Use the **failure_action** parameter to specify one of the following available default failure actions:

Option	Description
dump_to_rootfs	**kdump** tries to save the core dump to the root file system. This option is especially useful in combination with a network target. If the network target is unreachable, this option configures **kdump** to save the core dump locally. The system reboots afterwards.
reboot	**kdump** reboots the system. The core dump is lost.
halt	**kdump** halts the system. The core dump is lost.
poweroff	**kdump** powers down the system. The core dump is lost.
shell	**kdump** opens a shell session from within the **initramfs** utility. This allows the user to record the core dump manually.

Additional resources

- **etc/kdump.conf**

CHAPTER 20. ENABLING KDUMP

This section provides the information and procedures necessary to enable and start the **kdump** service for all installed kernels or for a specific kernel.

20.1. KDUMP AND KEXEC

The **kdump** service provides a crash dumping mechanism. The service enables you to save the contents of the system's memory for later analysis. **kdump** uses the **kexec** system call to boot into a second kernel (a capture kernel) without rebooting. Then **kdump** captures the contents of the crashed kernel's memory (a crash dump or a vmcore) and saves it. The second kernel resides in a reserved part of the system memory.

You can enable **kdump** for all installed kernels on a machine or only for specified kernels. This is useful when there are multiple kernels used on a machine, some of which are stable enough that there is no concern that they might crash.

A kernel crash dump might be the only information available in the event of a system failure (a critical bug). Therefore, ensuring that **kdump** is operational is important in mission-critical environments. Red Hat advises that system administrators regularly update and test kexec-tools during your normal kernel update cycle. This is especially important when new kernel features are implemented.

When **kdump** is installed, a default **/etc/kdump.conf** file is created. The file includes the default minimum **kdump** configuration. You *can* edit this file to customize the **kdump** configuration, but it is not required.

The following is the high level procedure for installing, configuring, enabling, and starting **kdump**:

1. (Optional) Edit **kdump.conf** to configure **kdump**.
2. Add the **kdump** kernel to the **grub** configuration.
3. Enable **kdump**.
4. Reboot the machine to start **kdump**.

20.2. ENABLING KDUMP FOR ALL INSTALLED KERNELS

You can enable and start the **kdump** service for all kernels installed on the machine.

Prerequisites

- Administrator privileges

Procedure

1. Add the **kdump** kernel to the system's Grand Unified Bootloader (GRUB) 2 configuration file:

```
# /sbin/grubby --update-kernel=ALL --args="crashkernel=auto"
```

2. Enable the **kdump** service.

```
# systemctl enable kdump.service
```

3. Reboot the machine to start the service.

Verification

- Check that the **kdump** service is running:

```
# /bin/systemctl status kdump.service

○ kdump.service - Crash recovery kernel arming
     Loaded: loaded (/usr/lib/systemd/system/kdump.service; enabled; vendor preset:
disabled)
     Active: active (live)
```

20.3. ENABLING KDUMP FOR A SPECIFIC INSTALLED KERNEL

You can enable the **kdump** service for a specific kernel on the machine.

Prerequisites

- Administrator privileges

Procedure

1. List the kernels installed on the machine.

```
# ls -a /boot/vmlinuz-*
/boot/vmlinuz-0-rescue-2930657cd0dc43c2b75db480e5e5b4a9 /boot/vmlinuz-4.18.0-
330.el8.x86_64 /boot/vmlinuz-4.18.0-330.rt7.111.el8.x86_64
```

2. Add a specific **kdump** kernel to the system's Grand Unified Bootloader (GRUB) 2 configuration file.
   For example:

```
# /sbin/grubby --update-kernel=vmlinuz-4.18.0-330.el8.x86_64 --args="crashkernel=auto"
```

3. Enable the **kdump** service.

```
# systemctl enable kdump.service
```

4. Reboot the machine to start the service.

Verification

- Check that the **kdump** service is running:

```
# /bin/systemctl status kdump.service

○ kdump.service - Crash recovery kernel arming
     Loaded: loaded (/usr/lib/systemd/system/kdump.service; enabled; vendor preset:
disabled)
     Active: active (live)
```

CHAPTER 21. NON-UNIFORM MEMORY ACCESS

The **taskset** utility only works on CPU affinity and has no knowledge of other NUMA resources such as memory nodes. 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.

Additional resources

- Andi Kleen's whitepaper, An NUMA API for Linux
- numactl(8) man page

CHAPTER 22. SETTING SCHEDULER PRIORITIES

Red Hat Enterprise Linux for Real Time kernel allows fine-grained control of scheduler priorities. It also allows application-level programs to be scheduled at a higher priority than kernel threads.

WARNING

Setting scheduler priorities can carry consequences and may cause the system to become unresponsive or behave unpredictably if crucial kernel processes are prevented from running as needed. Ultimately, the correct settings are workload-dependent.

22.1. VIEWING THREAD SCHEDULING PRIORITIES

Thread priorities are set using a series of levels, ranging from **0** (lowest priority) to **99** (highest priority). The **systemd** service manager can be used to change the default priorities of threads after the kernel boots.

Procedure

- To view scheduling priorities of running threads, use the tuna utility:

```
# tuna --show_threads
                    thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
 2      OTHER     0    0xfff       451            3        kthreadd
 3       FIFO     1        0     46395            2     ksoftirqd/0
 5      OTHER     0        0        11            1    kworker/0:0H
 7       FIFO    99        0         9            1   posixcputmr/0
...[output truncated]...
```

22.2. CHANGING THE PRIORITY OF SERVICES DURING BOOTING

Using **systemd**, you can set up real-time priority for services launched during the boot process.

Unit configuration directives are used to change the priority of a service during boot process. The boot process priority change is done by using the following directives in the service section of **/etc/systemd/system/*service*.system.d/priority.conf**:

CPUSchedulingPolicy=

Sets the CPU scheduling policy for executed processes. Takes one of the scheduling classes available on Linux:

- **other**
- **batch**
- **idle**

- **fifo**
- **rr**

CPUSchedulingPriority=

Sets the CPU scheduling priority for an executed processes. The available priority range depends on the selected CPU scheduling policy. For real-time scheduling policies, an integer between **1** (lowest priority) and **99** (highest priority) can be used.

Prerequisites

- Administrator privileges
- A service that runs on boot

Procedure

For an existing service:

1. Create a supplementary service configuration directory file for the service.

2. Add the scheduling policy and priority to the file in the ***[SERVICE]*** section.
   For example:

```
[SERVICE]
CPUSchedulingPolicy=fifo
CPUSchedulingPriority=20
EOF
```

3. Reload the **systemd** scripts configuration.

4. Restart the service.

Verification

- Display the service's priority.

The output shows the configured priority of the service.

For example:

```
                  thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
826     FIFO    20  0,1,2,3        13            0          mcelog
```

Additional resources

- Working with systemd unit files.

22.3. CONFIGURING THE CPU USAGE OF A SERVICE

Using **systemd**, you can specify the CPUs on which services can run.

Prerequisites

- Administrator privileges.

Procedure

1. Create a supplementary service configuration directory file for the service.

```
```

2. Add the CPUs to use for the service to the file using the **CPUAffinity** attribute in the ***[SERVICE]*** section.
   For example:

```
[SERVICE]
CPUAffinity=0,1
EOF
```

3. Reload the systemd scripts configuration.

```
```

4. Restart the service.

```
```

Verification

- Display the CPUs to which the specified service is limited.

```
```

where ***service*** is the specified service.

The following output shows that the **mcelog** service is limited to CPUs 0 and 1.

```
                  thread       ctxt_switches
 pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
12954   FIFO    20      0,1         2             1          mcelog
```

22.4. PRIORITY MAP

Scheduler priorities are defined in groups, with some groups dedicated to particular kernel functions.

Table 22.1. Thread priority table

Priority	Threads	Description
1	Low priority kernel threads	This priority is usually reserved for the tasks that need to be just above **SCHED_OTHER**.
2 - 49	Available for use	The range used for typical application priorities.
50	Default hard-IRQ value	This priority is the default value for hardware-based interrupts.
51 - 98	High priority threads	Use this range for threads that execute periodically and must have quick response times. Do **not** use this range for CPU-bound threads, because it will prevent responses to lower level interrupts.
99	Watchdogs and migration	System threads that must run at the highest priority.

22.5. ADDITIONAL RESOURCES

- Working with systemd unit files

CHAPTER 23. INFINIBAND IN RHEL FOR RT

Infiniband is a type of communications architecture often used to increase bandwidth, improve quality of service (QOS), and provide for failover. It can also be used to improve latency using the Remote Direct Memory Access (RDMA) mechanism.

Support for Infiniband under RHEL for Real Time is the same as that offered under RHEL 8.

Additional resources

- Getting Started with Infiniband

CHAPTER 24. USING ROCE AND HIGH-PERFORMANCE NETWORKING

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

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

Support for **RoCE** and **HPN** under RHEL for Real Time does not differ from the support offered under RHEL 8.

NOTE

For more information on how to set up ethernet networks, see Configuring RoCE.

CHAPTER 25. TRACING LATENCIES WITH TRACE-CMD

The **trace-cmd** utility is a front end to the **ftrace** utility. It can enable **ftrace** actions, without the need to write to the **/sys/kernel/debug/tracing/** directory. **trace-cmd** does not add any overhead when it is installed.

Prerequisites

- Administrator privileges

25.1. INSTALLING TRACE-CMD

The **trace-cmd** utility provides a front-end to the **ftrace** utility.

Prerequisites

- Administrator privileges

Procedure

- Install **trace-cmd**.

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

25.2. RUNNING TRACE-CMD

You can use the **trace-cmd** utility to access all **ftrace** functionality.

Prerequisites

- Administrator privileges

Procedure

- Enter **trace-cmd *command***
  where ***command*** is an **ftrace** option.

NOTE

See the **trace-cmd(1)** man page for a complete list of commands and options. Most of the individual commands also have their own man pages, **trace-cmd-*command***.

25.3. TRACE-CMD EXAMPLES

This provides a number of **trace-cmd** examples.

Examples

- Enable and start recording functions executing within the kernel while *myapp* runs.

```
# trace-cmd record -p function myapp
```

This records functions from all CPUs and all tasks, even those not related to *myapp*.

- Display the result.

```
# trace-cmd report
```

- Record only functions that start with **sched** while *myapp* runs.

```
# trace-cmd record -p function -l 'sched*' myapp
```

- Enable all the IRQ events.

```
# trace-cmd start -e irq
```

- Start the **wakeup_rt** tracer.

```
# trace-cmd start -p wakeup_rt
```

- Start the **preemptirqsoff** tracer, while disabling function tracing.

```
# trace-cmd start -p preemptirqsoff -d
```

NOTE

The version of **trace-cmd** in RHEL 8 turns off **ftrace_enabled** instead of using the **function-trace** option. You can enable **ftrace** again with **trace-cmd start -p** function.

- Restore the state in which the system was before **trace-cmd** started modifying it.

```
# trace-cmd start -p nop
```

This is important if you want to use the **debugfs** file system after using **trace-cmd**, whether or not the system was restarted in the meantime.

- Trace a single trace point.

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

- Stop tracing.

```
# trace-cmd record stop
```

Additional resources

- **trace-cmd(1)** man page

CHAPTER 26. ISOLATING CPUS USING TUNED-PROFILES-REALTIME

To give application threads the most execution time possible, you can isolate CPUs. Therefore, remove as many extraneous tasks from a CPU as possible. Isolating CPUs generally involves:

- Removing all user-space threads
- Removing any unbound kernel threads (bound kernel threads are tied to a specific CPU and may not be moved)
- Removing interrupts by modifying the **/proc/irq/N/smp_affinity** property of each Interrupt Request (IRQ) number **N** in the system

This section shows how to automate these operations using the **isolated_cores=cpulist** configuration option of the **tuned-profiles-realtime** package.

Prerequisites

- Administrator privileges

26.1. CHOOSING CPUS TO ISOLATE

Choosing the CPUs to isolate requires careful consideration of the CPU topology of the system. Different use cases may require different configuration:

- If you have a multi-threaded application where threads need to communicate with one another by sharing cache, they may need to be kept on the same NUMA node or physical socket.
- If you run multiple unrelated real-time applications, separating the CPUs by NUMA node or socket may be suitable.

The **hwloc** package provides utilities that are useful for getting information about CPUs, including **lstopo-no-graphics** and **numactl**.

Prerequisites

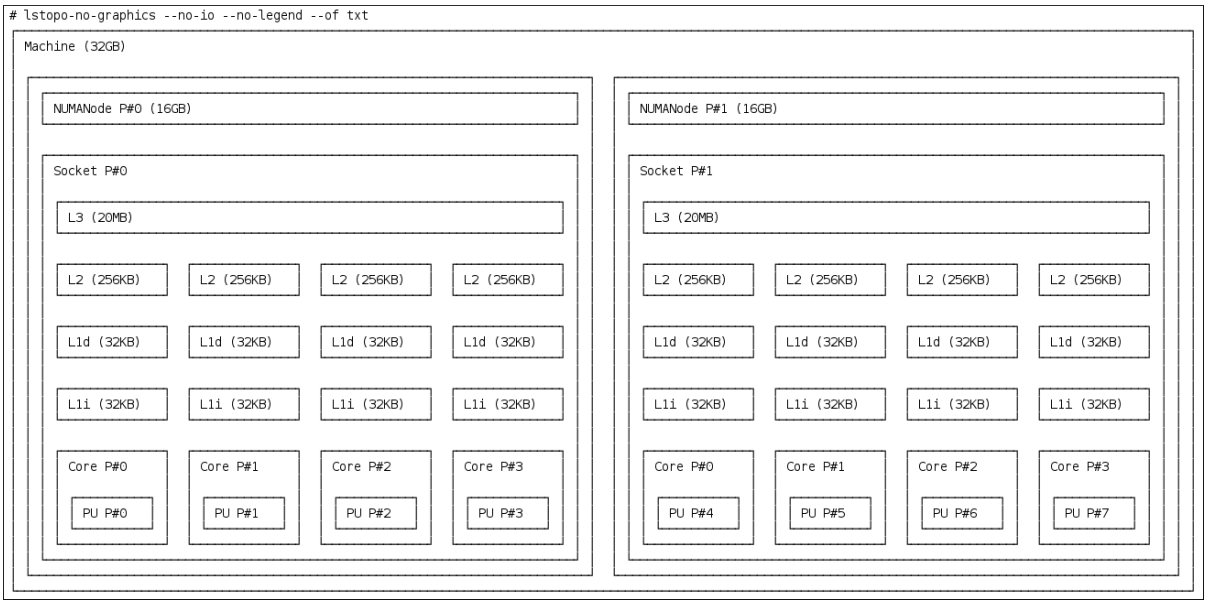
- The **hwloc** package must be installed.

Procedure

1. View the layout of available CPUs in physical packages:

```
# lstopo-no-graphics --no-io --no-legend --of txt
```

Figure 26.1. Showing the layout of CPUs using lstopo-no-graphics



This command is useful for multi-threaded applications, because it shows how many cores and sockets are available and the logical distance of the NUMA nodes.

Additionally, the **hwloc-gui** package provides the **lstopo** utility, which produces graphical output.

2. View more information about the CPUs, such as the distance between nodes:

```
# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3
node 0 size: 16159 MB
node 0 free: 6323 MB
node 1 cpus: 4 5 6 7
node 1 size: 16384 MB
node 1 free: 10289 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

Additional resources

- hwloc(7) man page

26.2. ISOLATING CPUS USING TUNED'S ISOLATED_CORES OPTION

The initial mechanism for isolating CPUs is specifying the boot parameter **isolcpus=cpulist** on the kernel boot command line. The recommended way to do this for RHEL for Real Time is to use the **tuned** daemon and its **tuned-profiles-realtime** package.

Prerequisites

- The **tuned** and **tuned-profiles-realtime** packages are installed.

Procedure

1. As a root user, open **/etc/tuned/realtime-variables.conf** in a text editor.
2. Set **isolated_cores=*cpulist*** to specify the CPUs that you want to isolate. You can use CPU numbers and ranges.

   #### Examples:

   ```
   isolated_cores=0-3,5,7
   ```

   This isolates cores 0, 1, 2, 3, 5, and 7.

   In a two socket system with 8 cores, where NUMA node 0 has cores 0-3 and NUMA node 1 has cores 4-8, to allocate two cores for a multi-threaded application, specify:

   ```
   isolated_cores=4,5
   ```

   This prevents any user-space threads from being assigned to CPUs 4 and 5.

   To pick CPUs from different NUMA nodes for unrelated applications, specify:

   ```
   isolated_cores=0,4
   ```

   This prevents any user-space threads from being assigned to CPUs 0 and 4.
3. Activate the realtime **tuned** profile using the **tuned-adm** utility.

   ```
   # tuned-adm profile realtime
   ```
4. Reboot the machine.

Verification

- Search for the **isolcpus** parameter in the kernel command line:

  ```
  $ cat /proc/cmdline | grep isolcpus
  BOOT_IMAGE=/vmlinuz-4.18.0-305.rt7.72.el8.x86_64 root=/dev/mapper/rhel_foo-root ro
  crashkernel=auto rd.lvm.lv=rhel_foo/root rd.lvm.lv=rhel_foo/swap console=ttyS0,115200n81
  isolcpus=0,4
  ```

CHAPTER 27. ISOLATING CPUS USING THE NOHZ AND NOHZ_FULL PARAMETERS

The **nohz** and **nohz_full** parameters modify activity on specified CPUs. To enable these kernel boot parameters, you need to use one of the following tuned profiles: **realtime-virtual-host**, **realtime-virtual-guest**, or **cpu-partitioning**.

nohz=on

Reduces timer activity on a particular set of CPUs.
The **nohz** parameter is mainly used to reduce timer interrupts on idle CPUs. This helps battery life by allowing idle CPUs to run in reduced power mode. While not being directly useful for real-time response time, the **nohz** parameter does not directly impact real-time response time negatively. But the **nohz** parameter is required to activate the **nohz_full** parameter that does have positive implications for real-time performance.

nohz_full=*cpulist*

The **nohz_full** parameter treats the timer ticks of a list of specified CPUs differently. If a CPU is specified as a **nohz_full** CPU and there is only one runnable task on the CPU, then the kernel stops sending timer ticks to that CPU. As a result, more time may be spent running the application and less time spent servicing interrupts and context switching.

Additional resources

- Configuring Kernel Tick Time

CHAPTER 28. LIMITING SCHED_OTHER TASK MIGRATION

You can limit the tasks that **SCHED_OTHER** migrates to other CPUs using the **sched_nr_migrate** variable.

Prerequisites

- Administrator privileges.

28.1. 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 adjusted to specify the number of tasks that will move at a time. Because real-time 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, thus disabling interrupts.

If there are a large number of tasks that need to be moved, it occurs while interrupts are disabled, so no timer events or wakeups will be allowed to happen simultaneously. This can cause severe latencies for real-time tasks when **sched_nr_migrate** is set to a large value.

28.2. LIMITING SCHED_OTHER TASK MIGRATION USING THE SCHED_NR_MIGRATE VARIABLE

Increasing the **sched_nr_migrate** variable provides high performance from **SCHED_OTHER** threads that spawn many tasks at the expense of real-time latency.

For low real-time task latency at the expense of **SCHED_OTHER** task performance, the value must be lowered. The default value is **8**.

Procedure

- To adjust the value of the **sched_nr_migrate** variable, echo the value directly to **/proc/sys/kernel/sched_nr_migrate**:

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

Verification

- View the contents of **/proc/sys/kernel/sched_nr_migrate**:

```
~]# cat > /proc/sys/kernel/sched_nr_migrate
2
```

CHAPTER 29. IMPROVING CPU PERFORMANCE BY USING RCU CALLBACKS

The **Read-Copy-Update** (**RCU**) system is a lockless mechanism for mutual exclusion of threads inside the kernel. As a consequence of performing RCU operations, call-backs are sometimes queued on CPUs to be performed at a future moment when removing memory is safe.

To improve CPU performance using RCU callbacks:

- You can remove CPUs from being candidates for running CPU callbacks.
- You can assign a CPU to handle all RCU callbacks. This CPU is called the housekeeping CPU.
- You can relieve CPUs from the responsibility of awakening RCU offload threads.

This combination reduces the interference on CPUs that are dedicated for the user's workload.

Prerequisites

- Administrator privileges
- The **tuna** package is installed

29.1. OFFLOADING RCU CALLBACKS

You can offload **RCU** callbacks using the **rcu_nocbs** and **rcu_nocb_poll** kernel parameters.

Procedure

- To remove one or more CPUs from the candidates for running RCU callbacks, specify the list of CPUs in the **rcu_nocbs** kernel parameter, for example:

```
rcu_nocbs=1,4-6
```

or

```
rcu_nocbs=3
```

The second example instructs the kernel that CPU 3 is a no-callback CPU. This means that RCU callbacks will not be done in the **rcuc/$CPU** thread pinned to CPU 3, but in the **rcuo/$CPU** thread. You can move this trhead to a housekeeping CPU to relieve CPU 3 from being assigned RCU callback jobs.

29.2. MOVING RCU CALLBACKS

You can assign a housekeeping CPU to handle all RCU callback threads. To do this, use the **tuna** command and move all RCU callbacks to the housekeeping CPU.

Procedure

- Move RCU callback threads to the housekeeping CPU:

```
# tuna --threads=rcu --cpus=x --move
```

where ***x*** is the CPU number of the housekeeping CPU.

This action relieves all CPUs other than CPU *X* from handling RCU callback threads.

29.3. RELIEVING CPUS FROM AWAKENING RCU OFFLOAD THREADS

Although the RCU offload threads can perform the RCU callbacks on another CPU, each CPU is responsible for awakening the corresponding RCU offload thread. You can relieve a CPU from this responsibility,

Procedure

- Set the **rcu_nocb_poll** kernel parameter.
  This command causes a timer to periodically raise the RCU offload threads to check if there are callbacks to run.

29.4. ADDITIONAL RESOURCES

- Avoiding RCU Stalls in the real-time kernel

CHAPTER 30. REAL TIME SCHEDULING ISSUES AND SOLUTIONS

This section provides information about real time scheduling issues and the available solutions.

Real time scheduling policies

The two real time scheduling policies in RHEL for Real Time share one main characteristic: they run until they are preempted by a higher priority thread or until they "wait", either by sleeping or performing I/O. In the case of **SCHED_RR**, a thread may be preempted by the operating system so that another thread of equal **SCHED_RR** priority may run. In either of these cases, no provision is made by the POSIX specifications that define the policies for allowing lower priority threads to get any CPU time.

This characteristic of real-time threads means that it is easy to write an application which monopolizes 100% of a given CPU. However, this causes problems for the operating system. For examplem, the operating system is responsible for managing both system-wide and per-CPU resources and must periodically examine data structures describing these resources and perform housekeeping activities with them. But if a core is monopolized by a **SCHED_FIFO** thread, it cannot perform its housekeeping tasks. Eventually the entire system becomes unstable, potentially crashing.

On the RHEL for Real Time kernel, interrupt handlers run as threads with a **SCHED_FIFO** priority. (he default priority is **50**. A cpu-hog thread with a **SCHED_FIFO** or **SCHED_RR** policy higher than the interrupt handler threads can prevent interrupt handlers from running. This causes programs waiting for data signaled by those interrupts to be starved and fail.

Real time scheduler throttling

Red Hat Enterprise Linux for Real Time comes with a safeguard mechanism that allows the system administrator to allocate bandwith for use by real time tasks. This safeguard mechanism is known as real time scheduler throttling. Real time scheduler throttling is controlled by two parameters in the **/proc** file system:

- **/proc/sys/kernel/sched_rt_period_us**
  Defines the period in µs (microseconds) to be considered 100% of CPU bandwidth. The default value is **1,000,000 µs** (1 second). Changes to the value of the period must be very well thought out, as a period too long or too small are equally dangerous.

- **/proc/sys/kernel/sched_rt_runtime_us**
  The total bandwidth available for all real time tasks. The default value is **950,000 µs** (0.95 s) or, in other words, 95% of the CPU bandwidth. Setting the value to **-1** means that real time tasks may use up to 100% of CPU time. This is only adequate when the real time tasks are well engineered and have no obvious caveats, such as unbounded polling loops.

  The default values for the real time throttling mechanism define that the real time tasks can use 95% of the CPU time. The remaining 5% will be devoted to non-real time tasks, such as tasks running under **SCHED_OTHER** and similar scheduling policies. It is important to note that if a single real time task occupies that 95% CPU time slot, the remaining real time tasks on that CPU will not run. Only non-real time tasks use the remaining 5% of CPU time.

  The impact of the default values include the following:

  - Rogue real time tasks do not lock up the system by not allowing non-real time tasks to run.
  - Real time tasks have at most 95% of CPU time available for them, which can affect their performance.

Additional resources

Additional resources

- Real-Time group scheduling

CHAPTER 31. TRACING LATENCIES USING FTRACE

The **ftrace** utility is one of the diagnostic facilities provided with the RHEL for Real Time kernel. **ftrace** can be used by developers to analyze and debug latency and performance issues that occur outside of the 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 of the **ftrace** tracers, such as the function tracer, can produce exceedingly large amounts of data, which can turn trace log analysis into a time-consuming task. However, you can instruct the tracer to begin and end only when the application reaches critical code paths.

Prerequisites

- Root privileges

31.1. USING THE FTRACE UTILITY TO TRACE LATENCIES

You can trace latencies using the **ftrace** utility.

Prerequisites

- You must be a root user

Procedure

1. View the available tracers on the system.

```
# cat /sys/kernel/debug/tracing/available_tracers
function_graph wakeup_rt wakeup preemptirqsoff preemptoff irqsoff function nop
```

   The user interface for **ftrace** is a series of files within **debugfs**.

   The **ftrace** files are also located in the **/sys/kernel/debug/tracing/** directory.

2. Move to the **/sys/kernel/debug/tracing/** directory.

```
# cd /sys/kernel/debug/tracing
```

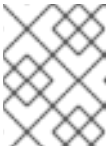
   The files in this directory can only be modified by the root user, because enabling tracing can have an impact on the performance of the system.

3. To start a tracing session:

   a. Select a tracer you want to use from the list of available tracers in **/sys/kernel/debug/tracing/available_tracers**.

   b. Insert the name of the selector into the **/sys/kernel/debug/tracing/current_tracer**.

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



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.

4. The function-trace option is useful because tracing latencies with **wakeup_rt**, **preemptirqsoff**, and so on automatically enables **function tracing**, which may exaggerate the overhead.
Check if **function** and **function_graph** tracing are enabled:

```
# cat /sys/kernel/debug/tracing/options/function-trace
1
```

- A value of **1** indicates that **function** and **function_graph** tracing are enabled.
- A value of **0** indicates that **function** and **function_graph** tracing are disabled.

5. By default, **function** and **function_graph** tracing are enabled. To turn **function** and **function_graph** tracing on or off, echo the appropriate value to the **/sys/kernel/debug/tracing/options/function-trace** file.

```
# echo 0 > /sys/kernel/debug/tracing/options/function-trace
# echo 1 > /sys/kernel/debug/tracing/options/function-trace
```

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 an unexpected trace output.

6. Adjust the details and parameters of the tracers by changing the values for the various files in the **/debugfs/tracing/** directory.
For example:

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** always shows the highest recorded latency since it was last reset.

- To reset the maximum latency, echo **0** into the **tracing_max_latency** file.

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

- To see only latencies greater than a set amount, echo the amount in microseconds:

```
# echo 200 > /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
```

7. View the trace logs:

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

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

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

9. View the functions being traced:

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

10. Filter the functions being traced by editing the settings in **/sys/kernel/debug/tracing/set_ftrace_filter**. If no filters are specified in the file, all functions are traced.

11. To change filter settings, 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.
For examples, see ftrace examples.

31.2. FTRACE FILES

The following are the main files in the **/sys/kernel/debug/tracing/** directory.

ftrace files

trace

The file that shows the output of an **ftrace** trace. This is really a snapshot of the trace in time, because the trace stops when this file is read, and it does not consume the events read. That is, if the user disabled tracing and reads this file, it will report the same thing every time it is read.

trace_pipe

The file that shows the output of an **ftrace** trace as it reads the trace live. It is a producer/consumer trace. That is, each read will consume the event that is read. This can be used to read an active trace without stopping the trace as it is read.

available_tracers

A list of ftrace tracers that have been compiled into the kernel.

current_tracer

Enables or disables an **ftrace** tracer.

events

A directory that contains events to trace and can be used to enable or disable events, as well as set filters for the events.

tracing_on

Disable and enable recording to the **ftrace** buffer. Disabling tracing via the **tracing_on** file does not disable the actual tracing that is happening inside the kernel. It only disables writing to the buffer. The work to do the trace still happens, but the data does not go anywhere.

31.3. FTRACE TRACERS

Depending on how the kernel is configured, not all tracers may be available for a given kernel. For the RHEL for Real Time kernels, the trace and debug kernels have different tracers than the production kernel does. This is because some of the tracers have a noticeable overhead when the tracer is configured into the kernel, but not active. Those tracers are only enabled for the **trace** and **debug** kernels.

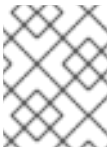
Tracers

function

One of the most widely applicable tracers. Traces the function calls within the kernel. This can cause noticeable overhead depending on the number of functions traced. When not active, it creates little overhead.

function_graph

The **function_graph** tracer is designed to present results in a more visually appealing format. This tracer also traces the exit of the function, displaying a flow of function calls in the kernel.



NOTE

This tracer has more overhead than the **function** tracer when enabled, but the same low overhead when disabled.

wakeup

A full CPU tracer that reports the activity happening across all CPUs. It records the time that it takes to wake up the highest priority task in the system, whether that task is a real time task or not. Recording the max time it takes to wake up a non-real time task hides the times it takes to wake up a real time task.

wakeup_rt

A full CPU tracer that reports the activity happening across all CPUs. It records the time that it takes from the current highest priority task to wake up to until the time it is scheduled. This tracer only records the time for real time tasks.

preemptirqsoff

Traces the areas that disable pre-emption or 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.

nop

The default tracer. It does not provide any tracing facility itself, but as events may interleave into any tracer, the **nop** tracer is used for specific interest in tracing events.

31.4. FTRACE EXAMPLES

The following provides a number of examples for changing the filtering of functions being traced. You can use the * wildcard at both the beginning and end of a word. For example: ***irq**** will select all functions that contain **irq** in the name. The wildcard cannot, however, be used inside a word.

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

Examples of filters

- Trace only the **schedule** function:

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

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

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

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

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

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

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

CHAPTER 32. 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 Red Hat Enterprise Linux for Real Time 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 documentation:

32.1. 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 the amount of interrupts, packets can be collected and a single interrupt generated for a collection of packets.
In systems that transfer large amounts of data where throughput is a priority, using the default value or increasing coalesce can increase throughput and lower the number of interrupts hitting CPUs. 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.

See also Section 32.4, “Reduce TCP performance spikes” and Section 33.1, “Reducing the TCP delayed ACK timeout”.

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)

32.2. 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.

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. See Remote Syslogging with rsyslog on Red Hat Enterprise Linux for details.

2. 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/rsyslog.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**.

3. It is also possible to configure **syslogd** to log all locally generated system messages, by adding a wildcard line to the **/etc/rsyslog.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 Red Hat Enterprise Linux for Real Time 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)
- rsyslog.conf(5)
- rsyslogd(8)

32.3. 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 real-time tasks and other undesirable impacts on determinism.

Disabling the pcscd daemon

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

```
~]# systemctl status pcscd
pcscd.service - PC/SC Smart Card Daemon
   Loaded: loaded (/usr/lib/systemd/system/pcscd.service; static)
   Active: active (running) &hellip;
```

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

```
~]# systemctl stop pcscd
```

3. Ensure that **pcscd** does not restart on boot.

```
~]# systemctl disable pcscd
```

32.4. REDUCE TCP PERFORMANCE SPIKES

Turn timestamps off to reduce performance spikes related to timestamp generation. The **sysctl** command controls the values of TCP related entries, setting the timestamps kernel parameter found at **/proc/sys/net/ipv4/tcp_timestamps**.

- Turn timestamps off with the following command:

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

- Turn timestamps on with the following command:

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

- 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.

32.5. REDUCE CPU PERFORMANCE SPIKES

The kernel command line parameter **skew_tick** helps to smooth jitter on moderate to large systems with latency-sensitive applications running. A common source of latency spikes on a realtime Linux system is when multiple CPUs contend on common locks in the Linux kernel timer tick handler. The usual locks responsible for the contention are the **xtime_lock**, which is used by the timekeeping system, and the RCU (Read-Copy-Update) structure locks.

Using the **skew_tick=1** boot parameter reduces contention on these kernel locks. The parameter ensures that the ticks per CPU do not occur simultaneously by making their start times 'skewed'. Skewing the start times of the per-CPU timer ticks decreases the potential for lock conflicts, reducing system jitter for interrupt response times.

CHAPTER 33. REALTIME-SPECIFIC TUNING

Once you have completed the optimization in Chapter 32, *General system tuning* you are ready to start Red Hat Enterprise Linux for Real Time specific tuning. You must have the Red Hat Enterprise Linux for Real Time kernel installed for these procedures.

IMPORTANT

Do not attempt to use the tools in this section without first having completed Chapter 32, *General system tuning*. You will not see a performance improvement.

33.1. REDUCING THE TCP DELAYED ACK TIMEOUT

On Red Hat Enterprise Linux, there are two modes used by TCP to acknowledge data reception:

Quick ACK

- This mode is used at the start of a TCP connection so that the congestion window can grow quickly.
- The acknowledgment (ACK) timeout interval (ATO) is set to **tcp_ato_min**, the minimum timeout value.
- To change the default TCP ACK timeout value, write the required value in milliseconds to the **/proc/sys/net/ipv4/tcp_ato_min** file:

```
~]# echo 4 > /proc/sys/net/ipv4/tcp_ato_min
```

Delayed ACK

- After the connection is established, TCP assumes this mode, in which ACKs for multiple received packets can be sent in a single packet.
- ATO is set to **tcp_delack_min** to restart or reset the timer.
- To change the default TCP Delayed ACK value, write the required value in milliseconds to the **/proc/sys/net/ipv4/tcp_delack_min** file:

```
~]# echo 4 > /proc/sys/net/ipv4/tcp_delack_min
```

TCP switches between the two modes depending on the current congestion.

Some applications that send small network packets could experience latencies due to the TCP quick and delayed acknowledgment timeouts, which previously were 40 ms by default. That means small packets from an application that seldom sends information through the network could experience a delay up to 40 ms to receive the acknowledgment that a packet has been received by the other side. To minimize this issue, both **tcp_ato_min** and **tcp_delack_min** timeouts are now 4 ms by default.

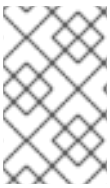
These default values are tunable and can be adjusted according to the needs of the user's environment, as described above.

NOTE

Using timeout values that are too low or too high might have a negative impact on the network throughput and latencies experienced by applications. Different environments might require different settings of these timeouts.

CHAPTER 34. APPLICATION TUNING AND DEPLOYMENT

This chapter contains tips related to enhancing and developing Red Hat Enterprise Linux for Real Time applications.



NOTE

In general, try to use *POSIX* (Portable Operating System Interface) defined APIs. Red Hat Enterprise Linux for Real Time is compliant with POSIX standards, and latency reduction in the Red Hat Enterprise Linux for Real Time kernel is also based on POSIX.

Further reading

For further reading on developing your own Red Hat Enterprise Linux for Real Time applications, start by reading the RTWiki Article.

34.1. SIGNAL PROCESSING IN REAL-TIME APPLICATIONS

Traditional UNIX and POSIX signals have their uses, especially for error handling, but they are not suitable for use in real-time 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 real-time 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

34.2. 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.

For more information, see Arnaldo Carvalho de Melo's paper on Earthquaky kernel interfaces.

Related manual pages

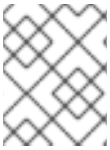
For more information, or for further reading, the following man pages are related to the information given in this section.

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

34.3. MUTEX OPTIONS

Standard mutex creation

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



NOTE

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

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

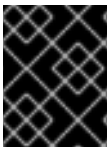
```
pthread_mutex_t my_mutex;

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 Red Hat Enterprise Linux for Real Time kernel. There are a number of mutex options that must be considered when writing or porting an application.

Advanced mutex options

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

34.4. TCP_NODELAY AND SMALL BUFFER WRITES

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

Using TCP_NODELAY and TCP_CORK to improve network latency

1. Applications that require lower latency on every packet sent 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)

34.5. SETTING REAL-TIME SCHEDULER PRIORITIES

Using **systemd** to set scheduler priorities is described in "Changing the priority of services during booting". In the example given in that procedure, some kernel threads could 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 to 89.

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 50 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 49. 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 Red Hat Enterprise Linux for Real Time 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.

34.6. LOADING DYNAMIC LIBRARIES

When developing your real-time application, 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 application 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 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)

34.7. 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 Red Hat Enterprise Linux for Real Time, 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 clocks is a standard for implementing and representing time sources. The POSIX clocks can be selected by each application, without affecting other applications in the system. This is in contrast to the hardware clocks, 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 Red Hat Enterprise Linux for Real Time Reference guide for Red Hat Enterprise Linux for Real Time.

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.

```
~]$ gcc 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

34.8. ABOUT PERF

Perf is a performance analysis tool. It provides a simple command line interface and separates the CPU hardware difference in Linux performance measurements. Perf is based on the **perf_events** interface exported by the kernel.

One advantage of perf is that it is both kernel and architecture neutral. The analysis data can be reviewed without requiring specific system configuration.

To be able to use **perf**, install the **perf** package by running the following command as **root**:

```
~]# yum install perf
```

Perf has the following options. Examples of the most common options and features follow, but further information on all options are available with the **perf help COMMAND**.

Example of perf options

```
]# perf

 usage: perf [--version] [--help] COMMAND [ARGS]

 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data file
```

```
  bench           General framework for benchmark suites
  buildid-cache   Manage build-id cache.
  buildid-list    List the buildids in a perf.data file
  diff            Read two perf.data files and display the differential profile
  evlist          List the event names in a perf.data file
  inject          Filter to augment the events stream with additional information
  kmem            Tool to trace/measure kernel memory(slab) properties
  kvm             Tool to trace/measure kvm guest os
  list            List all symbolic event types
  lock            Analyze lock events
  record          Run a command and record its profile into perf.data
  report          Read perf.data (created by perf record) and display the profile
  sched           Tool to trace/measure scheduler properties (latencies)
  script          Read perf.data (created by perf record) and display trace output
  stat            Run a command and gather performance counter statistics
  test            Runs sanity tests.
  timechart       Tool to visualize total system behavior during a workload
  top             System profiling tool.
  trace           strace inspired tool
  probe           Define new dynamic tracepoints

See 'pert help COMMAND' for more information on a specific command.
```

These following examples show a selection of the most used features, including record, archive, report, stat and list.

perf record

The perf record feature is used for collecting system-wide statistics. It can be used in all processors.

```
~]# perf record -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.725 MB perf.data (~31655 samples) ]
```

In this example, all CPUs are denoted with the option **-a**, and the process was terminated after a few seconds. The results show that it collected 0.725 MB of data, and created the following file of results.

```
~]# ls
perf.data
```

Example of the perf report and archive features

The data from the perf **record** feature can now be directly investigated using the perf **report** commands. If the samples are to be analyzed on a different system, use the perf **archive** command. This will not always be necessary as the DSOs (such as binaries and libraries) may already be present in the analysis system, such as the ~/**.debug**/ cache or if both systems have the same set of binaries.

Run the archive command to create an archive of results.

```
~]# perf archive
```

Collect the results as a tar archive to prepare the data for the pref **report**.

```
~]# tar xvf perf.data.tar.bz2 -C ~/.debug
```

Run the perf **report** to analyze the tarball.

```
~]# perf report
```

The output of the report is sorted according to the maximum CPU usage in percentage by the application. It shows if the sample has occurred in kernel or user space of the process.

A kernel sample, if not taking place in a kernel module will be marked by the notation **[kernel.kallsyms]**. If a kernel sample is taking place in the kernel module, it will be marked as **[module]**, **[ext4]**. For a process in user space, the results might show the shared library linked with the process.

The report denotes whether the process also occurs in kernel or user space. The result **[.]** indicates user space and **[k]** indicates kernel space. Finer grained details are available for review, including data appropriate for experienced perf developers.

Example of the perf list and stat features

The perf list and stat features show all the hardware or software trace points that can be probed.

The following example shows how to view the number of context switches with the perf **stat** feature.

```
~]# perf stat -e context-switches -a sleep 5
Performance counter stats for 'sleep 5':

         15,619 context-switches

      5.002060064 seconds time elapsed
```

The results show that in 5 seconds, 15619 context switches took place. Filesystem activity is also viewable, as shown in the following example script.

```
~]# for i in {1..100}; do touch /tmp/$i; sleep 1; done
```

In another terminal, run the following perf **stat** feature.

```
~]# perf stat -e ext4:ext4_request_inode -a sleep 5
 Performance counter stats for 'sleep 5':

             5 ext4:ext4_request_inode

      5.002253620 seconds time elapsed
```

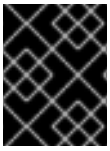
The results show that in 5 seconds the script asked to create 5 files, indicating that there are 5 inode requests.

There are a range of available options to get the hardware tracepoint activity. The following example shows a selection of the options in the perf **list** feature.

```
List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                               [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
  instructions                                       [Hardware event]
  cache-references                                   [Hardware event]
  cache-misses                                       [Hardware event]
```

```
branch-instructions OR branches                  [Hardware event]
branch-misses                                    [Hardware event]
bus-cycles                                       [Hardware event]

cpu-clock                                        [Software event]
task-clock                                       [Software event]
page-faults OR faults                            [Software event]
minor-faults                                     [Software event]
major-faults                                     [Software event]
context-switches OR cs                           [Software event]
cpu-migrations OR migrations                     [Software event]
alignment-faults                                 [Software event]
emulation-faults                                 [Software event]
...[output truncated]...
```



IMPORTANT

Sampling at too high a frequency can negatively impact the performance of your real-time system.

CHAPTER 35. CONTAINER SETUP AND TUNING

This chapter describes how to configure the real time kernel in order to utilize containers support.

35.1. PREREQUISITES

- Install podman and other container-related utilities.
- Get familiar with administration and management of Linux containers on RHEL 8.
- Install **kernel-rt** and other real time related packages mentioned in the Installation Guide of RHEL for Real Time 8.
- For information about tuning the Red Hat Enterprise Linux for Real Time kernel, refer to the Tuning Guide of RHEL for Real Time 8.

35.2. CREATING AND RUNNING A CONTAINER

All the following options can be used with both the real time kernel and the main RHEL kernel. The **kernel-rt** package brings potential determinism improvements and allows the usual troubleshooting.

The following procedure describes how to configure the Linux containers in relation with the real time kernel.

1. Create and change into a directory you want to use for the container:

```
# mkdir cyclictest
# cd cyclictest/
```

2. Log into a host that provides a container registry service:

```
# podman login registry.redhat.io
Username: my_customer_portal_login
Password: **
Login Succeeded!
```

For more information about logging into the registry host, refer to Building, running, and managing containers.

3. **Create the following Dockerfile:**

```
# vim Dockerfile
FROM rhel8
RUN subscription-manager repos --enable=rhel-8-for-x86_64-rt-rpm
RUN dnf -y install rt-tests
ENTRYPOINT cyclictest --smp -p95
```

4. **Build the container image from the directory containing the Dockerfile:**

```
# podman build -t cyclictest .
```

5. **Run the container using the image you built in the previous step:**

```
# podman run --device=/dev/cpu_dma_latency --cap-add ipc_lock --cap-add \
 sys_nice --cap-add sys_rawio --rm -ti cyclictest
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.08 0.10 0.09 2/947 15

T: 0 ( 8) P:95 I:1000 C: 3209 Min: 1 Act: 1 Avg: 1 Max:  14
T: 1 ( 9) P:95 I:1500 C: 2137 Min: 1 Act: 2 Avg: 1 Max:  23
T: 2 (10) P:95 I:2000 C: 1601 Min: 1 Act: 2 Avg: 2 Max:   7
T: 3 (11) P:95 I:2500 C: 1280 Min: 1 Act: 2 Avg: 2 Max:  72
T: 4 (12) P:95 I:3000 C: 1066 Min: 1 Act: 1 Avg: 1 Max:   7
T: 5 (13) P:95 I:3500 C:  913 Min: 1 Act: 2 Avg: 2 Max:  87
T: 6 (14) P:95 I:4000 C:  798 Min: 1 Act: 1 Avg: 2 Max:   7
T: 7 (15) P:95 I:4500 C:  709 Min: 1 Act: 2 Avg: 2 Max:  29
```

The example above shows the **podman run** command with the required, real time specific, options. For example, the first in first out (FIFO) scheduler policy is made available for workloads running inside the container through the **--cap-add=sys_nice** option. This option also allows setting the CPU affinity of threads, which is another important configuration dimension when tuning a real time workload.

The **--device=/dev/cpu_dma_latency** option makes the host device available inside the container (subsequently used by **cyclictest** workload to configure the CPU idle time management). If such device is not made available, an error similar to the message below appears:

```
WARN: stat /dev/cpu_dma_latency failed: No such file or directory
```

When confronted with error messages like in the example above, refer to the **podman-run** manual page. There are other options that are helpful in getting a specific workload running inside a container.

In some cases, you need to additionally add the **--device=/dev/cpu** option to add that directory hierarchy, mapping per-CPU device files such as **/dev/cpu/*/msr**.

35.3. FURTHER CONSIDERATIONS

The main RHEL kernels enable the real time group scheduling feature **CONFIG_RT_GROUP_SCHED** by default, however for real time kernels this feature is disabled.

The **CONFIG_RT_GROUP_SCHED** feature was developed independently of the **PREEMPT_RT** patchset used in the **kernel-rt** package and is intended to operate on real time processes on the main RHEL kernel. The **CONFIG_RT_GROUP_SCHED** feature is known to cause latency spikes and is therefore disabled on **PREEMPT_RT** enabled kernels. Therefore when testing your workload in a container running on the main RHEL kernel, some realtime bandwidth must be allocated to the container to be able to run the **SCHED_FIFO** or **SCHED_RR** tasks inside it. Refer to the **podman-run(1)** manual page for more information about allocating the real time bandwidth.

- Configure the following global setting before using podman's **--cpu-rt-runtime** command line option:

  ```
  # echo 950000 > /sys/fs/cgroup/cpu,cpuacct/machine.slice/cpu.rt_runtime_us
  ```

  For CPU isolation, use the existing recommendations for setting aside a set of cores for the RT workload. Then use **podman run --cpuset-cpus** command with the list of isolated CPU cores to be used.

To avoid cross Non-Uniform Memory Access (NUMA) node memory access, use the **podman run --cpuset-mems=[number_of_memory_nodes]** command, to specify the NUMA memory nodes to use. For more details, refer to the **podman-run** manual page.

Use **podman run --memory-reservation=[limit]** command to make sure that the minimal amount of memory required by the real time workload running on the container is set aside at container start time. For more details about memory reservation, see the **podman-run** manual page.