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: 2023-02-06

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

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. Let us know how we can improve it.

Submitting comments on specific passages

1. View the documentation in the **Multi-page HTML** format and ensure that you see the **Feedback** button in the upper right corner after the page fully loads.
2. Use your cursor to highlight the part of the text that you want to comment on.
3. Click the **Add Feedback** button that appears near the highlighted text.
4. Add your feedback and click **Submit**.

Submitting feedback through Bugzilla (account required)

1. Log in to the Bugzilla website.
2. Select the correct version from the **Version** menu.
3. Enter a descriptive title in the **Summary** field.
4. Enter your suggestion for improvement in the **Description** field. Include links to the relevant parts of the documentation.
5. Click **Submit Bug**.

CHAPTER 1. REAL-TIME KERNEL TUNING IN RHEL 8

Latency, or response time, refers to the time from an event and to the system response. It 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 replacement kernel that can be set for latency to meet those requirements. **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 for well-tuned systems and 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**.

WARNING

Failure to perform these tasks might 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.

1.3. 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

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

1.4. 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

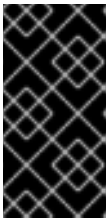
1. Do not run the **graphical interface** where it is not absolutely required, especially on servers. Check if the system is configured to boot into the GUI by default:

```
# 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.

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

Additional resources

- the **cron(8)** man page

1.5. 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
- the **numactl(8)** man page

1.6. 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.

1.7. INFINIBAND IN RHEL FOR REAL TIME

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 by using the Remote Direct Memory Access (RDMA) mechanism.

The support for InfiniBand on RHEL for Real Time is the same as the support available on Red Hat Enterprise Linux 8. For more information, see Configuring InfiniBand and RDMA networks.

1.8. USING ROCEE AND HIGH-PERFORMANCE NETWORKING

RoCEE (RDMA over Converged Enhanced 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 **RoCEE** 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 **RoCEE** and **HPN** under RHEL for Real Time does not differ from the support offered under RHEL 8.

Additional resources

- Configuring RoCE.

1.9. REDUCING CPU PERFORMANCE SPIKES

The kernel command line **skew_tick** parameter smooths jitter on moderate to large systems with latency-sensitive applications running. A common source of latency spikes on a real time Linux system is when multiple CPUs contend on common locks in the Linux kernel timer tick handler.

Prerequisites

- You have administrator permissions.

Procedure

- Set the **skew_tick** boot parameter to **1**.

1.10. 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

- Real-Time group scheduling

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 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

- the **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.

NOTE

The values printed by the **hwlatdetect** utility for inner and outer are the maximum latency values. They are deltas between consecutive reads of the current system clocksource (usually the TSC 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.

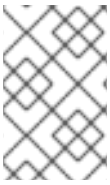
The file name is in the form **rteval-*<date>*-*N*-tar.bz2**, where ***<date>*** is the date the report was generated, ***N*** is a counter for the Nth run on ***<date>***.

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. USING MLOCK() SYSTEM CALLS ON RHEL FOR REAL TIME

The RHEL for Real-Time memory lock (**mlock()**) function enables the real-time calling processes to lock or unlock a specified range of the address space. This range prevents Linux from paging the locked memory when swapping memory space. After you allocate the physical page to the page table entry, references to that page become fast. The **mlock()** system calls include two functions: **mlock()** and **mlockall()**. Similarly, **munlock()** system call includes the **munlock()** and **munlockall()** functions.

6.1. MLOCK() AND MUNLOCK() SYSTEM CALLS

The **mlock()** and **mlockall()** system calls lock a specified memory range and do not page this memory. The following are the **mlock()** system call groups:

- **mlock()** system calls: lock a specified range of address.
- **munlock()** system calls: unlock a specified range of address.

The **mlock()** system calls, lock pages in the address range starting at **addr** and continuing for **len** bytes. When the call returns successfully, all pages that contain a part of the specified address range stay in the memory until unlocked later.

With **mlockall()** system calls, you can lock all mapped pages into the specified address range. Memory locks do not stack. Any page locked by several calls will unlock the specified address range or the entire region with a single **munlock()** system call. With **munlockall()** system calls, you can unlock the entire program space.

The status of the pages contained in a specific range depends on the value in the **flags** argument. The **flags** argument can be 0 or **MLOCK_ONFAULT**.

Memory locks are not inherited by a child process through fork and automatically removed when a process terminates.

WARNING

Use **mlock()** system calls with caution. Excessive use can cause out-of-memory (OOM) errors. When an application is large or if it has a large data domain, the **mlock()** calls can cause thrashing when the system is not able to allocate memory for other tasks.

When using **mlockall()** calls for real-time processes, ensure that you reserve sufficient stack pages.

6.2. USING MLOCK() SYSTEM CALLS TO LOCK PAGES

The real-time **mlock()** system calls use the **addr** parameter to specify the start of an address range and **len** to define the length of the address space in bytes. The **alloc_workbuf()** function dynamically allocates a memory buffer and locks it. Memory allocation is done by the **posix_memalig()** function to align the memory area to a page. The function **free_workbuf()** unlocks the memory area.

Prerequisites:

- You have root privileges or the **CAP_IPC_LOCK** capability to use **mlockall()** or **mlock()** on large buffers

Procedure

- To lock pages with **mlock()** system call, run the following command:

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

void *alloc_workbuf(size_t size)
{
  void ptr;
  int retval;

  // alloc memory aligned to a page, to prevent two mlock() in the same page.
  retval = posix_memalign(&ptr, (size_t) sysconf(_SC_PAGESIZE), size);

  // return NULL on failure
  if (retval)
    return NULL;

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

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

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

Verification

The real-time **mlock()** and **munlock()** calls return 0 when successful. In case of an error, they return -1 and set a **errno** to indicate the error.

6.3. USING MLOCKALL() SYSTEM CALLS TO LOCK ALL MAPPED PAGES

To lock and unlock real-time memory with **mlockall()** and **munlockall()** system calls, set the **flags** argument to 0 or one of the constants: **MCL_CURRENT** or **MCL_FUTURE**. With **MCL_FUTURE**, a future system call, such as **mmap2()**, **sbrk2()**, or **malloc3()**, might fail, because it causes the number of locked bytes to exceed the permitted maximum.

Prerequisites

- You have root privileges.

Procedure

- To use **mlockall()** and **munlockall()** real-time system calls :
  - Lock all mapped pages by using **mlockall()** system call:

```
#include <sys/mman.h>
int mlockall (int flags)
```

  - Unlock all mapped pages by using **munlockall()** system call:

```
#include <sys/mman.h>
int munlockall (void)
```

Additional resources

- **capabilities(7)** man page
- **mlock(2)** man page
- **mlock(3)** man page
- **move_pages(2)** man page
- **posix_memalign(3)** man page
- **posix_memalign(3p)** man page

6.4. USING MMAP() SYSTEM CALLS TO MAP FILES OR DEVICES INTO MEMORY

For large memory allocations on real-time systems, the memory allocation (**malloc**) method uses the **mmap()** system call to find addressable memory space. You can allocate and lock memory areas by setting **MAP_LOCKED** in the **flags** parameter.

As **mmap()** allocates memory on a page basis, there are no two locks on the same page, which prevents the double-lock or single-unlock problems.

Prerequisites

- You have root privileges.

Procedure

- To map a specific process address space:

```
#include <sys/mman.h>
#include <stdlib.h>

void *alloc_workbuf(size_t size)
{
 void *ptr;
```

```
 ptr = mmap(NULL, size, PROT_READ | PROT_WRITE,
           MAP_PRIVATE | MAP_ANONYMOUS | MAP_LOCKED, -1, 0);

 if (ptr == MAP_FAILED)
  return NULL;

 return ptr;
}

void
free_workbuf(void *ptr, size_t size)
{
 munmap(ptr, size);
}
```

Verification

- When the **mmap()** function completes successfully, it returns a pointer to the mapped area. On error, it returns the **MAP_FAILED** value and sets a **errno** to indicate the error.
- When the **munmap()** function completes successfully, it returns **0**. On error, it returns **-1** and sets an **errno** to indicate the error.

Additional resources

- the **mmap(2)** man page
- the **mlockall(2)** man page

6.5. PARAMETERS FOR MLOCK() SYSTEM CALLS

The following table lists the **mlock()** parameters.

Table 6.1. mlock parameters

Parameter	Description
addr	Specifies the process address space to lock or unlock. When NULL, the kernel chooses the page-aligned arrangement of data in the memory. If **addr** is not NULL, the kernel chooses a nearby page boundary, which is always above or equal to the value specified in **/proc/sys/vm/mmap_min_addr** file.
len	Specifies the length of the mapping, which must be greater than 0.
fd	Specifies the file descriptor.

Parameter	Description
prot	**mmap** and **munmap** calls define the desired memory protection with this parameter. **prot** takes one or a combination of **PROT_EXEC**, **PROT_READ**, **PROT_WRITE** or **PROT_NONE** values.
flags	Controls the mapping visibility to other processes that map the same file. It takes one of the values: **MAP_ANONYMOUS**, **MAP_LOCKED**, **MAP_PRIVATE** or **MAP_SHARED** values.
MCL_CURRENT	Locks all pages that are currently mapped into a process.
MCL_FUTURE	Sets the mode to lock subsequent memory allocations. These could be new pages required by a growing heap and stack, new memory-mapped files, or shared memory regions.

CHAPTER 7. SETTING CPU AFFINITY ON RHEL FOR REAL TIME

All threads and interrupt sources in the system has a processor affinity property. The operating system scheduler uses this information to determine the threads and interrupts to run on a CPU. Setting processor affinity, along with effective policy and priority settings, achieves the maximum possible performance. Applications always compete for resources, especially CPU time, with other processes. Depending on the application, related threads are often run on the same core. Alternatively, one application thread can be allocated to one core.

Systems that perform multitasking are naturally more prone to indeterminism. Even high priority applications can be delayed from executing while a lower priority application is in a critical section of code. After the low priority application exits the critical section, the kernel safely preempts the low priority application and schedules the high priority application on the processor. Additionally, migrating processes from one CPU to another can be costly due to cache invalidation. RHEL for Real Time includes tools that address some of these issues and allows latency to be better controlled.

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

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

Setting the following typical affinity setups can achieve maximum possible performance:

- Using a single CPU core for all system processes and setting the application to run on the remainder of the cores.
- Configuring a thread application and a specific kernel thread (network softirq or a driver thread) on the same CPU.
- Pairing the producer-consumer threads on each CPU. Producers and consumers are two classes of threads, where producers insert data into the buffer and consumers remove it from the buffer.

The usual good practice for tuning affinities on a real-time system is to determine the number of cores required to run the application and then isolate those cores. You achieve this with the Tuna tool or with the shell scripts to modify the bit mask value, such as the **taskset** command. The **taskset** command changes the affinity of a process and modifying the /**proc**/ file system entry changes the affinity of an interrupt.

7.1. TUNING PROCESSOR AFFINITY USING THE TASKSET COMMAND

On real-time, the **taskset** command helps to set or retrieve the CPU affinity of a running process. The **taskset** command takes **-p** and **-c** options. The **-p** or **--pid** option work an existing process and does not start a new task. The **-c** or **--cpu-list** specify a numerical list of processors instead of a bitmask. The list may contain multiple items, separated by comma, and a range of processors. For example, 0,5,7,9-11.

Prerequisites

- You have root privileges.

Procedure

- To check the process affinity for a specific process:

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

The command prints the affinity of the process with PID 1000. The process is configured to use either CPU 0 or CPU 1.

  - (Optional) To configure a specific CPU to bind a process:

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

  - (Optional) To define more than one CPU affinity:

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

  - (Optional) To configure a priority level and a policy on a specific CPU:

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

For further granularity, you can also specify the priority and policy. In the example, the command runs the **/bin/my-app** application on CPU 5 with **SCHED_FIFO** policy and a priority value of 78.

7.2. SETTING PROCESSOR AFFINITY USING THE SCHED_SETAFFINITY() SYSTEM CALL

You can also set processor affinity using the real-time **sched_setaffinity()** system call.

Prerequisite

- You have root privileges.

Procedure

- To set the processor affinity with **sched_setaffinity()**:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sched.h>

int main(int argc, char **argv)
{
  int i, online=0;
```

```
  ulong ncores = sysconf(_SC_NPROCESSORS_CONF);
  cpu_set_t *setp = CPU_ALLOC(ncores);
  ulong setsz = CPU_ALLOC_SIZE(ncores);

  CPU_ZERO_S(setsz, setp);

  if (sched_getaffinity(0, setsz, setp) == -1) {
    perror("sched_getaffinity(2) failed");
    exit(errno);
  }

  for (i=0; i < CPU_COUNT_S(setsz, setp); i) {
    if (CPU_ISSET_S(i, setsz, setp))
      online;
  }

  printf("%d cores configured, %d cpus allowed in affinity mask\n", ncores, online);
  CPU_FREE(setp);
}
```

7.3. ISOLATING A SINGLE CPU TO RUN HIGH UTILIZATION TASKS

Using the real-time **cpusets** mechanism, you can assign a set of CPUs and memory nodes for **SCHED_DEADLINE** tasks. In a task set which includes high and low CPU utilizing tasks, isolating a CPU to run the high utilization task and scheduling small utilization tasks on different sets of CPU, enables all tasks to meet the assigned **runtime**.

Prerequisites

- Root privileges

Procedure

1. Create two directories named as **cpuset**:

```
# cd /sys/fs/cgroup/cpuset/
# mkdir cluster
# mkdir partition
```

2. Disable the load balance of the root **cpuset** to create two new root domains in the **cpuset** directory:

```
# echo 0 > cpuset.sched_load_balance
```

3. In the cluster **cpuset**, schedule the low utilization tasks to run on CPU 1 to 7, verify memory size, and name the CPU as exclusive:

```
# cd cluster/
# echo 1-7 > cpuset.cpus
# echo 0 > cpuset.mems
# echo 1 > cpuset.cpu_exclusive
```

4. Move all low utilization tasks to the cpuset directory:

```
# ps -eLo lwp | while read thread; do echo $thread > tasks ; done
```

5. Create a partition named as **cpuset** and assign the high utilization task:

```
# cd ../partition/
# echo 1 > cpuset.cpu_exclusive
# echo 0 > cpuset.mems
# echo 0 > cpuset.cpus
```

6. Set the shell to the cpuset and start the deadline workload:

```
# echo $$ > tasks
# /root/d &
```

With this setup, the task isolated in the partitioned **cpuset** directory does not interfere with the task in the cluster **cpuset** directory. This enables all real-time tasks to meet the scheduler deadline.

7.4. 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 preemption 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 8. 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 file system 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 (the **atime** attribute). If you need to use a journaling file system, consider disabling **atime**.

8.1. DISABLING ATIME

Disabling the **atime** attribute increases performance and decreases power usage by limiting the number of writes to the file-system journal.

Procedure

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…
```

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…
```

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.

8.2. ADDITIONAL RESOURCES

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

CHAPTER 9. 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** kernel parameter 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.

9.1. DISABLING GRAPHICS CONSOLE LOGGING TO GRAPHICS ADAPTER

The **teletype** (**tty**) 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

- Remove the **console=tty0** option from the kernel configuration:

```
# grubby --update-kernel=ALL --remove-args="console=tty0"
```

9.2. 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 10. 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.

10.1. 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.

10.2. 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.

10.3. 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.

10.4. 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

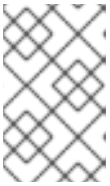
1. View the available clock sources.

```
# 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.

10.5. 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

- the **time(1)** man page

10.6. SYNCHRONIZING THE TSC TIMER ON OPTERON CPUS

The current generation of AMD64 Opteron processors can be susceptible to a large **gettimeofday** skew. This skew occurs when both **cpufreq** and the **Time Stamp Counter** (TSC) are in use. RHEL for Real Time provides a method to prevent this skew by forcing all processors to simultaneously change to the same frequency. As a result, the TSC on a single processor never increments at a different rate than the TSC on another processor.

Prerequisites

- Root permissions

Procedure

1. Enable the **clocksource=tsc** and **powernow-k8.tscsync=1** kernel options:

```
# grubby --update-kernel=ALL --args="clocksource=tsc powernow-k8.tscsync=1"
```

   This forces the use of TSC and enables simultaneous core processor frequency transitions.

2. Restart the machine.

Additional resources

- **gettimeofday(2)** man page

10.7. 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.

```
$ {EDITOR} 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 11. CONTROLLING POWER MANAGEMENT TRANSITIONS

You can control power management transitions to improve latency.

Prerequisites

- Root permissions for the system.

11.1. 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.

11.2. CONFIGURING POWER MANAGEMENT STATES

You can control power management transitions by configuring power management states. More specifically, you can write a value to the **/dev/cpu_dma_latency** file to change the maximum response time for processes, in microseconds. Or you can reference this file in an application or a script.

Prerequisites

- Administrator privileges

Procedure

1. Create the **cpu_dma_latency** file.

```
$ touch /dev/cpu_dma_latency
```

2. Open the file in a text editor.

```
$ sudo {EDITOR} /dev/cpu_dma_latency
```

3. Add the following program lines to the file.

```
#include <errno.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <unistd.h>

FILE *fp;

void start_low_latency() {
   char target = '0';

   fp= fopen("/dev/cpu_dma_latency", "r+");
   if (fp == NULL) {
      fprintf(stderr, "Failed to open PM QOS file: %s", strerror(errno));
      exit(errno);
   }
   fputc(target, fp);
}

void stop_low_latency() {
   if (fp == NULL)
   fclose(fp);
}

int main() {
   start_low_latency();
   //do some latency-sensitive tasks here
   stop_low_latency();
   return 0;
}
```

4. Compile the program.

```
$ sudo gcc $/dev/cpu_dma_latency -o ${output_name_of_choice}
```

5. Run the program.

```
$ sudo gcc $/dev/cpu_dma_latency -o ${output_name_of_choice}
```

CHAPTER 12. 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.

12.1. 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.

12.2. 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.

12.3. 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 13. 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.

13.1. INTERRUPT AND PROCESS BINDING

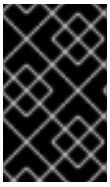
Isolating interrupts (IRQs) from user processes on different dedicated CPUs can minimize or eliminate latency in real-time environments.

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 13.1. Example of the CPU Mask for given CPUs

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

13.2. 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
```

13.3. 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

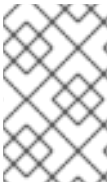
1. Open **/etc/sysconfig/irqbalance** in your preferred text editor and find the section of the file titled **IRQBALANCE_BANNED_CPUS**.

```
# 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.

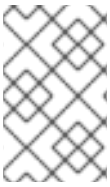


NOTE

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 13.2. 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**.

13.4. 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

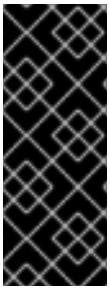
1. Perform an activity that will trigger the specified interrupt.

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.

13.5. 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.

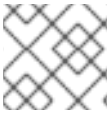
    ```
    # taskset -c 0,4,7-11 /usr/local/bin/my_embedded_process
    ```

    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

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

CHAPTER 14. 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.

14.1. PREREQUISITES

- Root permissions on the system.

14.2. 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

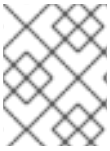
1. Display the current value of **/proc/sys/vm/panic_on_oom**.

```
# 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.

14.3. 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.

14.4. 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

- 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 15. IMPROVING LATENCY USING THE TUNA CLI

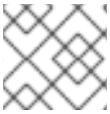
You can use the **tuna** CLI to improve latency on your system. The options used with the **tuna** command determine the method invoked to improve latency.

15.1. PREREQUISITES

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

15.2. 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.

15.3. 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
```

15.4. 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
```

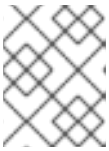
15.5. CHANGING PROCESS SCHEDULING POLICIES AND PRIORITIES USING THE TUNA CLI

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 16. INSTALLING KDUMP

The **kdump** service is installed and activated by default on the new Red Hat Enterprise Linux installations. Learn about **kdump** and how to install **kdump** when it is not enabled by default.

16.1. WHAT IS KDUMP

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

IMPORTANT

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

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 could crash.

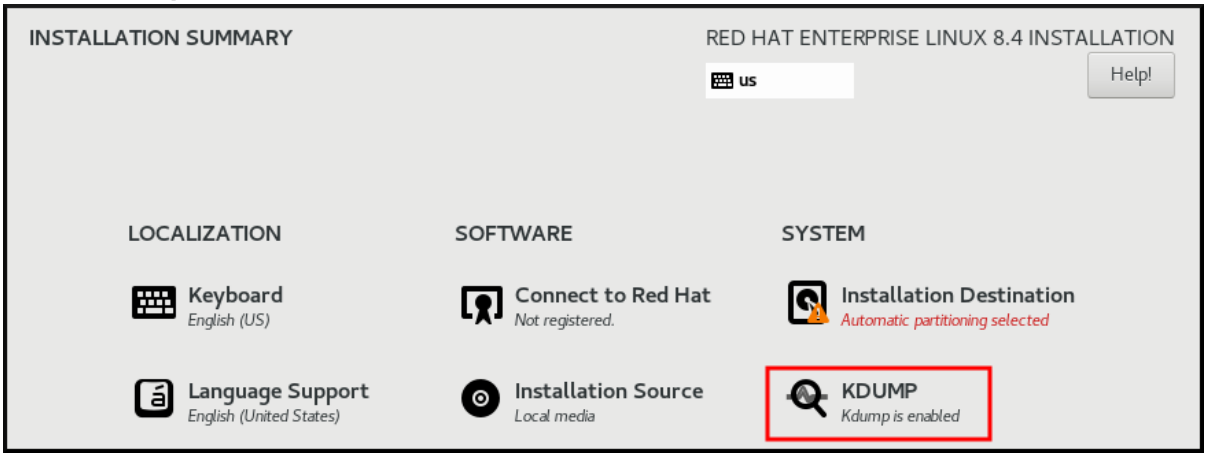
When **kdump** is installed, a default **/etc/kdump.conf** file is created. The file includes the default minimum **kdump** configuration. You can edit this file to customize the **kdump** configuration, but it is not required.

16.2. INSTALLING KDUMP USING ANACONDA

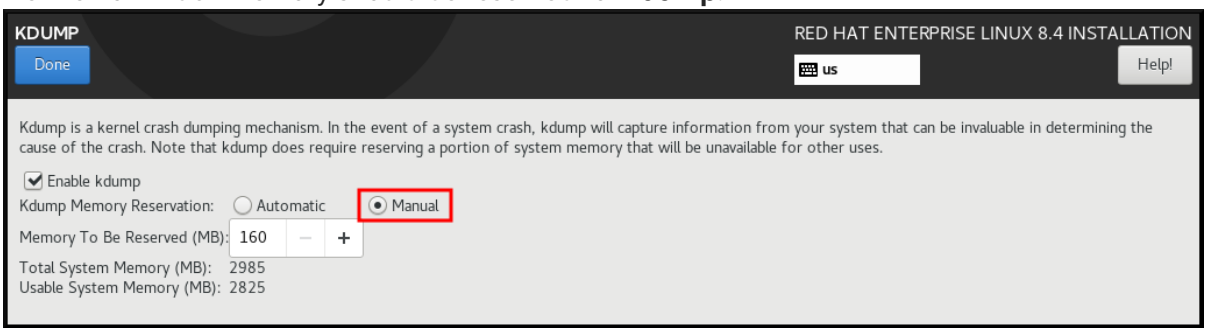
The **Anaconda** installer provides a graphical interface screen for **kdump** configuration during an interactive installation. The installer screen is titled as **KDUMP** and is available from the main **Installation Summary** screen. You can enable **kdump** and reserve the required amount of memory.

Procedure

1. Go to the **Kdump** field.
2. Enable **kdump** if not already enabled.



3. Define how much memory should be reserved for **kdump**.



16.3. INSTALLING KDUMP ON THE COMMAND LINE

Some installation options, such as custom **Kickstart** installations, in some cases do **not** install or enable **kdump** by default. If this is your case, follow the procedure below.

Prerequisites

- An active RHEL subscription
- The **kexec-tools** package
- Fulfilled requirements for **kdump** configurations and targets. For details, see Supported kdump configurations and targets.

Procedure

1. Check whether **kdump** is installed on your system:

```
# rpm -q kexec-tools
```

Output if the package is installed:

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

Output if the package is not installed:

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

2. Install **kdump** and other necessary packages by:

```
# dnf install kexec-tools
```

IMPORTANT

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

CHAPTER 17. CONFIGURING KDUMP ON THE COMMAND LINE

Plan and build your **kdump** environment.

17.1. ESTIMATING THE KDUMP SIZE

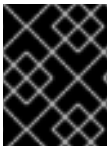
When planning and building your **kdump** environment, it is important to know how much space the crash dump file requires.

The **makedumpfile --mem-usage** command estimates how much space the crash dump file requires. It generates a memory usage report. The report helps you determine the dump level and which pages are safe to be excluded.

Procedure

- Execute the following command to generate a memory usage report:

```
# makedumpfile --mem-usage /proc/kcore


TYPE        PAGES    EXCLUDABLE    DESCRIPTION
----------------------------------------------------------------
ZERO          501635      yes        Pages filled with zero
CACHE         51657       yes        Cache pages
CACHE_PRIVATE 5442        yes        Cache pages + private
USER          16301       yes        User process pages
FREE          77738211    yes        Free pages
KERN_DATA     1333192     no         Dumpable kernel data
```



IMPORTANT

The **makedumpfile --mem-usage** command reports required memory in pages. This means that you must calculate the size of memory in use against the kernel page size.

17.2. CONFIGURING KDUMP MEMORY USAGE

The memory for **kdump** is reserved during the system boot. The memory size is set in the system Grand Unified Bootloader (GRUB) configuration. The memory size depends on the value of the **crashkernel=** option specified in the configuration file and the size of the system physical memory.

The **crashkernel=** option can be defined in multiple ways. You can either specify the **crashkernel=** value or configure the **auto** option. The **crashkernel=auto** parameter reserves memory automatically, based on the total amount of physical memory in the system. When configured, the kernel will automatically reserve an appropriate amount of required memory for the capture 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.

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

Prerequisites

- Root permissions.
- Fulfilled requirements for **kdump** configurations and targets. For details, see Supported kdump configurations and targets.

Procedure

1. Prepare the **crashkernel=** option.
   - For example, to reserve 128 MB of memory, use the following:

     ```
     crashkernel=128M
     ```

   - Alternatively, 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
     ```

     The above example reserves 64 MB of memory if the total amount of system memory is between 512 MB and 2 GB. If the total amount of memory is more than 2 GB, 128 MB is reserved.

   - Offset the reserved memory.
     Some systems require to reserve memory with a certain fixed offset since **crashkernel** reservation is very early, and it wants to reserve some area for special usage. If the offset is set, the reserved memory begins there. To offset the reserved memory, use the following syntax:

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

     In this example, **kdump** reserves 128 MB of memory starting at 16 MB (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. In that case, the offset is always specified last. For example:

     ```
     crashkernel=512M-2G:64M,2G-:128M@16M
     ```

2. Apply the **crashkernel=** option to your boot loader configuration:

   ```
   # grubby --update-kernel=ALL --args="crashkernel=<value>"
   ```

   Replace ***<value>*** with the value of the the **crashkernel=** option that you prepared in the previous step.

Additional resources

- Memory requirements for kdump
- Configuring kernel command-line parameters
- 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)** manual page

17.3. CONFIGURING THE KDUMP TARGET

The crash dump is usually stored as a file in a local file system, written directly to a device. Alternatively, you can set up for the crash dump to be sent over a network using the **NFS** or **SSH** protocols. Only one of these options to preserve a crash dump file can be set at a time. The default behavior is to store it in the **/var/crash/** directory of the local file system.

Prerequisites

- **Root** permissions.
- Fulfilled requirements for **kdump** configurations and targets. For details, see Supported kdump configurations and targets.

Procedure

- To store the crash dump file in **/var/crash/** directory of the local file system, edit the **/etc/kdump.conf** file and specify the path:

```
path /var/crash
```

The option **path /var/crash** represents the path to the file system in which **kdump** saves the crash dump file. When you specify a dump target in the **/etc/kdump.conf** file, then the **path** is relative to the specified dump target.

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.

WARNING

kdump saves the crash dump file in **/var/crash/var/crash** directory, when the dump target is mounted at **/var/crash** and the option **path** is also set as **/var/crash** in the **/etc/kdump.conf** file. For example, in the following instance, the **ext4** file system is already mounted at **/var/crash** and the **path** are set as **/var/crash**:

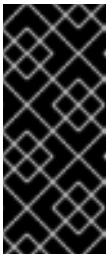
```
# grep -v ^# /etc/kdump.conf | grep -v ^$
ext4 /dev/mapper/vg00-varcrashvol
path /var/crash
core_collector makedumpfile -c --message-level 1 -d 31
```

This results in the **/var/crash/var/crash** path. To solve this problem, use the option **path** / instead of **path /var/crash**

- To change the local directory in which the crash dump is to be saved, as **root**, edit the **/etc/kdump.conf** configuration file as described below.

  1. Remove the hash sign ("#") from the beginning of the **#path /var/crash** line.
  2. Replace the value with the intended directory path. For example:

     ```
     path /usr/local/cores
     ```

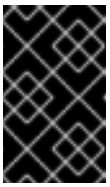
     IMPORTANT

     In RHEL 8, the directory defined as the kdump target using the **path** directive must exist when the **kdump** systemd service is started - otherwise the service fails. This behavior is different from earlier releases of RHEL, where the directory was being created automatically if it did not exist when starting the service.

- To write the file to a different partition, as **root**, edit the **/etc/kdump.conf** configuration file as described below.

  1. Remove the hash sign ("#") from the beginning of the **#ext4** line, depending on your choice.
     - device name (the **#ext4 /dev/vg/lv_kdump** line)
     - file system label (the **#ext4 LABEL=/boot** line)
     - UUID (the **#ext4 UUID=03138356-5e61-4ab3-b58e-27507ac41937** line)
  2. Change the file system type as well as the device name, label or UUID to the desired values. For example:

     ```
     ext4 UUID=03138356-5e61-4ab3-b58e-27507ac41937
     ```

     IMPORTANT

     It is recommended to specify storage devices using a **LABEL=** or **UUID=**. Disk device names such as **/dev/sda3** are not guaranteed to be consistent across reboot.

- To write the crash dump directly to a device, edit the **/etc/kdump.conf** configuration file:

  1. Remove the hash sign ("#") from the beginning of the **#raw /dev/vg/lv_kdump** line.
  2. Replace the value with the intended device name. For example:

     ```
     raw /dev/sdb1
     ```

- To store the crash dump to a remote machine using the **NFS** protocol, edit the **/etc/kdump.conf** configuration file:

  1. Remove the hash sign ("#") from the beginning of the **#nfs my.server.com:/export/tmp** line.
  2. Replace the value with a valid hostname and directory path. For example:

```
nfs penguin.example.com:/export/cores
```

- To store the crash dump to a remote machine using the **SSH** protocol, edit the **/etc/kdump.conf** configuration file:

  1. Remove the hash sign ("#") from the beginning of the **#ssh user@my.server.com** line.
  2. Replace the value with a valid username and hostname.
  3. Include your **SSH** key in the configuration.
     - Remove the hash sign from the beginning of the **#sshkey /root/.ssh/kdump_id_rsa** line.
     - Change the value 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
```

17.4. CONFIGURING THE KDUMP CORE COLLECTOR

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

- Compressing the size of a crash dump file and copying only necessary pages using various dump levels
- Excluding unnecessary crash dump pages
- Filtering the page types to be included in the crash dump.

Syntax

```
core_collector makedumpfile -l --message-level 1 -d 31
```

Options

- **-c**, **-l** or **-p**: specify compress dump file format by each page using either, **zlib** for **-c** option, **lzo** for **-l** option or **snappy** for **-p** option.
- **-d (dump_level)**: excludes pages so that they are not copied to the dump file.
- **--message-level** : specify the message types. You can restrict outputs printed by specifying **message_level** with this option. For example, specifying 7 as **message_level** prints common messages and error messages. The maximum value of **message_level** is 31

Prerequisites

- **Root** permissions
- Fulfilled requirements for **kdump** configurations and targets. For details, see Supported kdump configurations and targets.

Procedure

1. As **root**, edit the **/etc/kdump.conf** configuration file and remove the hash sign ("#") from the beginning of the **#core_collector makedumpfile -l --message-level 1 -d 31**.
2. To enable crash dump file compression, execute:

```
core_collector makedumpfile -l --message-level 1 -d 31
```

The **-l** option specifies the **dump** compressed file format. The **-d** option specifies dump level as 31. The **--message-level** option specifies message level as 1.

Also, consider following examples with the **-c** and **-p** options:

- To compress a crash dump file using **-c**:

```
core_collector makedumpfile -c -d 31 --message-level 1
```

- To compress a crash dump file using **-p**:

```
core_collector makedumpfile -p -d 31 --message-level 1
```

Additional resources

- the **makedumpfile(8)** man page
- The kdump configuration file

17.5. CONFIGURING THE KDUMP DEFAULT FAILURE RESPONSES

By default, when **kdump** fails to create a crash dump file at the configured target location, the system reboots and the dump is lost in the process. To change this behavior, follow the procedure below.

Prerequisites

- Root permissions.
- Fulfilled requirements for **kdump** configurations and targets. For details, see Supported kdump configurations and targets.

Procedure

1. As **root**, remove the hash sign ("#") from the beginning of the **#failure_action** line in the **/etc/kdump.conf** configuration file.
2. Replace the value with a desired action.

```
failure_action poweroff
```

Additional resources

- Configuring the kdump target

17.6. 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.

17.7. 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.
  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.

17.8. 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**

17.9. TESTING THE KDUMP CONFIGURATION

You can test that the crash dump process works and is valid before the machine enters production.

WARNING

The commands below cause the kernel to crash. Use caution when following these steps, and never carelessly use them on active production system.

Procedure

1. Reboot the system with **kdump** enabled.
2. Make sure that **kdump** is running:

```
# systemctl is-active kdump
active
```

3. Force the Linux kernel to crash:

```
echo 1 > /proc/sys/kernel/sysrq
echo c > /proc/sysrq-trigger
```

WARNING

The command above crashes the kernel, and a reboot is required.

Once booted again, the ***address-YYYY-MM-DD-HH:MM:SS*/vmcore** file is created at the location you have specified in the **/etc/kdump.conf** file (by default to **/var/crash/**).

NOTE

This action confirms the validity of the configuration. Also it is possible to use this action to record how long it takes for a crash dump to complete with a representative work-load.

Additional resources

- Configuring the kdump target

CHAPTER 18. ENABLING KDUMP

Enable and start the **kdump** service for all installed kernels or for a specific kernel.

18.1. 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 **crashkernel=auto** command-line parameter to all installed kernels:

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

2. Enable the **kdump** service.

```
# systemctl enable --now kdump.service
```

Verification

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

```
# 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)
```

18.2. 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) configuration file.
   For example:

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

3. Enable the **kdump** service.

```
# systemctl enable --now kdump.service
```

Verification

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

```
# 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)
```

18.3. DISABLING THE KDUMP SERVICE

To disable the **kdump** service at boot time, follow the procedure below.

Prerequisites

- Fulfilled requirements for **kdump** configurations and targets. For details, see Supported kdump configurations and targets.
- All configurations for installing **kdump** are set up according to your needs. For details, see Installing kdump.

Procedure

1. To stop the **kdump** service in the current session:

```
# systemctl stop kdump.service
```

2. To disable the **kdump** service:

```
# systemctl disable kdump.service
```

WARNING

It is recommended to set **kptr_restrict=1**. In that case, the **kdumpctl** service loads the crash kernel regardless of Kernel Address Space Layout (KASLR) being enabled or not.

Troubleshooting step

When **kptr_restrict** is not set to (1), and if KASLR is enabled, the contents of **/proc/kcore** file are generated as all zeros. Consequently, the **kdumpctl** service fails to access the **/proc/kcore** and load the crash kernel.

To work around this problem, the **/usr/share/doc/kexec-tools/kexec-kdump-howto.txt** file displays a warning message, which recommends the **kptr_restrict=1** setting.

To ensure that **kdumpctl** service loads the crash kernel, verify that **kernel.kptr_restrict = 1** is listed in the **sysctl.conf** file.

Additional resources

- Configuring basic system settings in RHEL

CHAPTER 19. 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.

19.1. 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]...
```

19.2. 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.

```
# cat <<-EOF > /etc/systemd/system/mcelog.system.d/priority.conf
```

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.

```
# systemctl daemon-reload
```

4. Restart the service.

```
# systemctl restart mcelog
```

Verification

- Display the service's priority.

```
$ tuna -t mcelog -P
```

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.

19.3. 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.

```
# md sscd
```

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.

```
# systemctl daemon-reload
```

4. Restart the service.

```
# systemctl restart service
```

Verification

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

```
$ tuna -t mcelog -P
```

  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
```

```
:_content-type: REFERENCE
```

19.4. PRIORITY MAP

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

Table 19.1. 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.

19.5. ADDITIONAL RESOURCES

- Working with systemd unit files

CHAPTER 20. NETWORK DETERMINISM TIPS

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.
- Do you need to 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.

There are numerous tools for tuning the network. This section provides information on some of the more useful tools.

20.1. COALESCING INTERRUPTS

In systems that transfer large amounts of data where throughput is a priority, using the default value or increasing coalescence can increase throughput and lower the number of interrupts hitting CPUs. For systems requiring a rapid network response, reducing or disabling coalescence is advised.

To reduce the number of interrupts, packets can be collected and a single interrupt generated for a collection of packets.

Prerequisites

- Administrator privileges.

Procedure

- To enable coalescing interrupts, run the **ethtool** command with the **--coalesce** option.

```
# ethtool -C tun0
```

Verification

Verify that coalescing interrupts are enabled.

```
# ethtool -c tun0
Coalesce parameters for tun0:
Adaptive RX: n/a  TX: n/a
stats-block-usecs: n/a
sample-interval: n/a
pkt-rate-low: n/a
pkt-rate-high: n/a

rx-usecs: n/a
rx-frames: 0
rx-usecs-irq: n/a
rx-frames-irq: n/a

tx-usecs: n/a
```

```
tx-frames: n/a
tx-usecs-irq: n/a
tx-frames-irq: n/a

rx-usecs-low: n/a
rx-frame-low: n/a
tx-usecs-low: n/a
tx-frame-low: n/a

rx-usecs-high: n/a
rx-frame-high: n/a
tx-usecs-high: n/a
tx-frame-high: n/a

CQE mode RX: n/a  TX: n/a
```

20.2. AVOIDING NETWORK CONGESTION

I/O switches can often be subject to back-pressure, where network data builds up as a result of full buffers. You can change pause parameters and avoid network congestion.

Prerequisites

- Administrator privileges

Procedure

- To change pause parameters, run the **ethtool** command with the **-A** option.

```
# ethtool -A enp0s31f6
```

Verification

Verify that the pause parameter changed.

```
# ethtool -a enp0s31f6
Pause parameters for enp0s31f6:
Autonegotiate: on
RX:  on
TX:  on
```

20.3. MONITORING NETWORK PROTOCOL STATISTICS

The **netstat** command can be used to monitor network traffic.

Procedure

To monitor network traffic:

```
$ netstat -s
Ip:
    Forwarding: 1
    30817508 total packets received
    2927 forwarded
```

```
    0 incoming packets discarded
    30813320 incoming packets delivered
    19184491 requests sent out
    181 outgoing packets dropped
    2628 dropped because of missing route
Icmp
    29450 ICMP messages received
    213 input ICMP message failed
    ICMP input histogram:
        destination unreachable: 29431
        echo requests: 19
    10141 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        destination unreachable: 10122
        echo replies: 19
IcmpMsg:
        InType3: 29431
        InType8: 19
        OutType0: 19
        OutType3: 10122
Tcp:
    162638 active connection openings
    89 passive connection openings
    38908 failed connection attempts
    17869 connection resets received
    48 connections established
    8456952 segments received
    9323882 segments sent out
    69885 segments retransmitted
    1143 bad segments received
    56209 resets sent
Udp:
    21929780 packets received
    1319 packets to unknown port received
    712919 packet receive errors
    10134989 packets sent
    712919 receive buffer errors
    180 send buffer errors
    IgnoredMulti: 39231
```

20.4. ADDITIONAL RESOURCES

- the **ethtool(8)** man page
- the **netstat(8)** man page

CHAPTER 21. 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.

21.1. 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
```

21.2. 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***.

21.3. 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
```

21.4. ADDITIONAL RESOURCES

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

CHAPTER 22. ISOLATING CPUS USING TUNED-PROFILES-REAL-TIME

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. Kernel related bound threads are linked to a specific CPU and cannot not be moved).
- Removing interrupts by modifying the **/proc/irq/N/smp_affinity** property of each Interrupt Request (IRQ) number **N** in the system.

By using the **isolated_cores=cpulist** configuration option of the [package]`tuned-profiles-realtime `package, you can automate operations to isolate a CPU.

Prerequisites

- You have administrator privileges.

22.1. 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

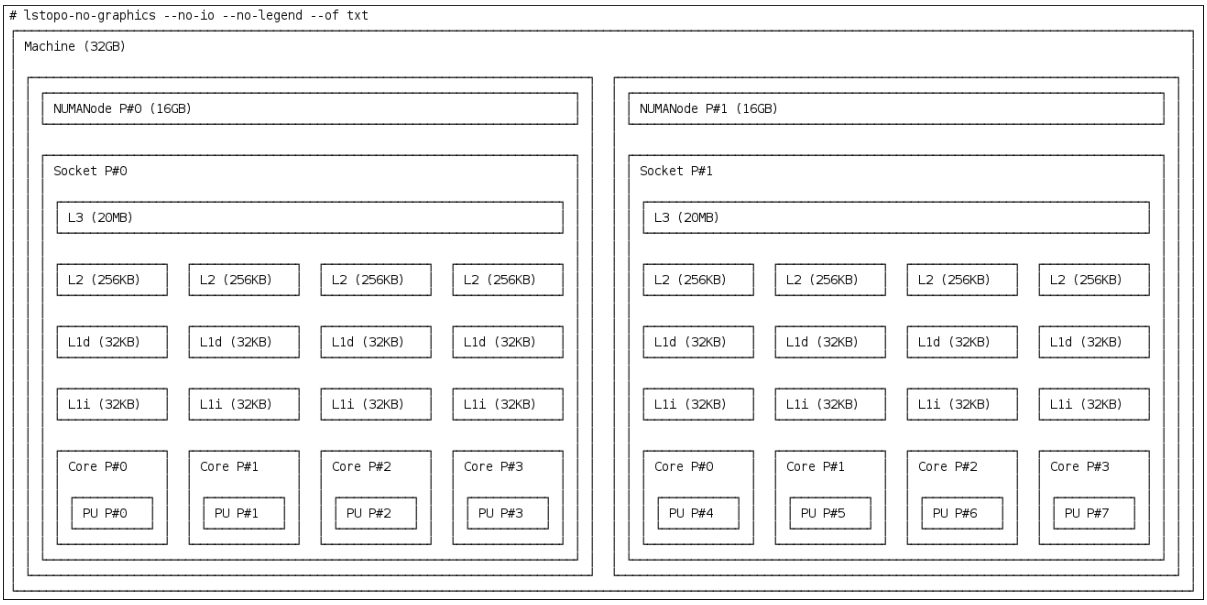
- The **hwloc** package must be installed.

Procedure

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

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

Figure 22.1. 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

- the **hwloc(7)** man page

22.2. 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.

NOTE

In **tuned-profiles-realtime** version 2.19 and higher, the built-in function **calc_isolated_cores** applies the initial CPU setup automatically. The **/etc/tuned/realtime-variables.conf** configuration file includes the default variable content as **isolated_cores=${f:calc_isolated_cores:2}**.

By default, **calc_isolated_cores** reserves one core per socket for housekeeping and isolates the rest. If you must change the default configuration, comment out the **isolated_cores=${f:calc_isolated_cores:2}** line in **/etc/tuned/realtime-variables.conf** configuration file and follow the procedure steps for Isolating CPUs using TuneD's **isolated_cores** option.

Prerequisites

- The **TuneD** and **tuned-profiles-realtime** packages are installed.
- You have root privileges.

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 for changes to take effect.

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
```

22.3. 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 23. 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.

23.1. 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.

23.2. 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 24. REDUCING TCP PERFORMANCE SPIKES

Generating TCP timestamps can result in TCP performance spikes. The **sysctl** command controls the values of TCP related entries, setting the timestamps kernel parameter found at **/proc/sys/net/ipv4/tcp_timestamps**.

Prerequisites

- Administrator privileges.

24.1. TURNING OFF TCP TIMESTAMPS

Turning off TCP timestamps can reduce TCP performance spikes.

Procedure

- Turn off TCP timestamps:

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

The output shows that the value of **net.ip4.tcp_timestamps** options is **0**. That is, TCP timestamps are disabled.

24.2. TURNING ON TCP TIMESTAMPS

Generating timestamps can cause TCP performance spikes. You can reduce TCP performance spikes by disabling TCP timestamps. If you find that generating TCP timestamps is not causing TCP performance spikes, you can enable them.

Procedure

- Enable TCP timestamps.

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

The output shows that the value of **net.ip4.tcp_timestamps** is **1**. That is, TCP timestamps are enabled.

24.3. DISPLAYING THE TCP TIMESTAMP STATUS

You can view the status of TCP timestamp generation.

Procedure

- Display the TCP timestamp generation status:

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

The value **1** indicates that timestamps are being generated. The value **0** indicates timestamps are being not generated.

CHAPTER 25. 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

25.1. 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.

25.2. 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.

25.3. 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.

25.4. ADDITIONAL RESOURCES

- Avoiding RCU Stalls in the real-time kernel

CHAPTER 26. 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

- Administrator privileges.

26.1. USING THE FTRACE UTILITY TO TRACE LATENCIES

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

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
```

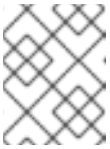
   The files in this directory can only be modified by the root user, because enabling tracing can have an impact on the performance of the system.

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.

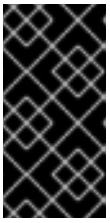
4. The function-trace option is useful because tracing latencies with **wakeup_rt**, **preemptirqsoff**, and so on automatically enables **function tracing**, which may exaggerate the overhead.
   Check if **function** and **function_graph** tracing are enabled:

```
# 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.

26.2. 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.

26.3. 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 preemption or interrupts, and records the maximum amount of time for which preemption 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.

26.4. 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

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 27. APPLICATION TUNING AND DEPLOYMENT

The following sections provide tips about enhancing and developing RHEL for Real Time applications.

NOTE

In general, try to use **POSIX** defined APIs (application programming interfaces). RHEL for Real Time is compliant with **POSIX** standards. Latency reduction in RHEL for Real Time kernel is also based on **POSIX**.

27.1. SIGNAL PROCESSING IN REAL-TIME APPLICATIONS

Traditional **UNIX** and **POSIX** signals have their uses, especially for error handling, but they are not suitable as an event delivery mechanism in real-time applications. This is because the current Linux kernel signal handling code is quite complex, mainly due to legacy behavior and the many 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 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. That is, 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 completes, 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 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. The code paths through these relatively new constructs are much cleaner than the legacy handling code for signals.

Additional resources

- Requirements of the POSIX Signal Model

27.2. SYNCHRONIZING THREADS

The **sched_yield** command is a synchronization mechanism that can allow lower priority threads a chance to run. This type of request is prone to failure when issued from within a poorly-written application.

A higher priority thread can call **sched_yield()** to allow other threads a chance to run. The calling process gets moved to the tail of the queue of processes running at that priority. When this occurs in a situation where there are no other processes running at the same priority, the calling process continues running. If the priority of that process is high, it can potentially create a busy loop, rendering the machine unusable.

When a **SCHED_DEADLINE** task calls **sched_yield()**, it gives up the configured CPU, and the remaining runtime is immediately throttled until the next period. The **sched_yield()** behavior allows the task to wake up at the start of the next period.

The scheduler is better able to determine when, and if, there actually are other threads waiting to run. Avoid using **sched_yield()** on any real-time task.

Procedure

- To call the **sched_yield()** function, run the following code:

```
for(;;) {
        do_the_computation();
        /*
 * Notify the scheduler the end of the computation
         * This syscall will block until the next replenishment
         */
sched_yield();
  }
```

The **SCHED_DEADLINE** task gets throttled by the conflict-based search (CBS) algorithm until the next period (start of next execution of the loop).

Additional resources

- the **pthread.h(P)** man page
- the **sched_yield(2)** man page
- the **sched_yield(3p)** man page
- Techniques that can have its behavior changed when the kernel is replaced .

27.3. REAL-TIME SCHEDULER PRIORITIES

The **systemd** command can be used to set real-time priority for services launched during the boot process. This is described in *Changing the priority of services during booting* .

In the example given in that procedure, some kernel threads can be given a very high priority. This allows the default priorities to 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 that can be used by applications. Use extreme caution when scheduling any application thread above priority 49 because it can prevent essential system services from running, because it can prevent essential system services from running. This can result in unpredictable behavior, including blocked network traffic, blocked virtual memory paging, and data corruption due to blocked filesystem journaling.

If any application threads are scheduled above priority 89, ensure that the threads run only a very short code path. Failure to do so would undermine the low latency capabilities of the RHEL for Real Time kernel.

Setting real-time priority for non-privileged users

By default, only root users are able to change priority and scheduling information. To grant non-privileged users the ability to adjust these settings, the best method is to add the non-privileged user to the **realtime** group.

IMPORTANT

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

27.4. LOADING DYNAMIC LIBRARIES

When developing your real-time application, consider resolving symbols at startup to avoid non-deterministic latencies during program execution. Note that resolving symbols at startup can slow down program initialization.

You can instruct Dynamic Libraries to load at application startup by setting the **LD_BIND_NOW** variable with **ld.so**, the dynamic linker/loader.

For example, the following shell 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 &_
```

Additional resources

- the **ld.so(8)** man page

27.5. ADDITIONAL RESOURCES

- HOWTO: Build an RT-application

CHAPTER 28. APPLICATION TIMESTAMPING

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

You can reduce the cost of reading the clock by selecting a hardware clock that has a reading mechanism, faster than that of the default clock.

In RHEL 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 possible CPU cost.

These benefits are more evident on systems which use hardware clocks with high reading costs.

28.1. POSIX CLOCKS

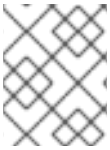
POSIX is a standard for implementing and representing time sources. You can assign a POSIX clock to an application without affecting other applications in the system. This is in contrast to hardware clocks which are selected by the kernel and implemented across the system.

The function used to read a given POSIX clock is **clock_gettime()**, which is defined at **<time.h>**. The kernel counterpart to **clock_gettime()** is a system call. When a user process calls **clock_gettime()**:

1. The corresponding C library (**glibc**) calls the **sys_clock_gettime()** system call.
2. **sys_clock_gettime()** performs the requested operation.
3. **sys_clock_gettime()** returns the result to the user program.

However, the context switch from the user application to the kernel has a CPU 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 context switching to the kernel, thus making it faster to read the clock, support for the **CLOCK_MONOTONIC_COARSE** and **CLOCK_REALTIME_COARSE** POSIX clocks was added, in the form of a virtual dynamic shared object (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. This yields a significant performance gain. Time readings for **_COARSE** clocks have a millisecond (ms) resolution, meaning that time intervals smaller than 1 ms are not recorded. The **_COARSE** variants of the POSIX clocks are suitable for any application that can accommodate millisecond clock resolution.



NOTE

To compare the cost and resolution of reading POSIX clocks with and without the **_COARSE** prefix, see the RHEL for Real Time Reference guide.

28.2. THE _COARSE CLOCK VARIANT IN CLOCK_GETTIME

The following code shows an example of code using the **clock_gettime** function with the **CLOCK_MONOTONIC_COARSE** POSIX clock:

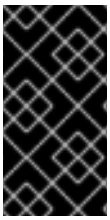
```
#include <time.h>

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 by adding checks 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.

NOTE

The **clock_gettime()** man page provides more information about writing 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

28.3. ADDITIONAL RESOURCES

- **clock_gettime()** man page

CHAPTER 29. IMPROVING NETWORK LATENCY USING TCP_NODELAY

By default, **TCP** uses Nagle's algorithm to collect small outgoing packets to send all at once. This can cause higher rates of latency.

Prerequisites

- Administrator privileges.

29.1. THE EFFECTS OF USING TCP_NODELAY

Applications that require low latency on every packet sent must be run on sockets with the **TCP_NODELAY** option enabled. This sends buffer writes to the kernel as soon as an event occurs.

Note

For **TCP_NODELAY** to be effective, applications must avoid doing small, logically related buffer writes. Otherwise, these small writes cause **TCP** to send these multiple buffers as individual packets, resulting in poor overall performance.

If applications have several buffers that are logically related and must be sent as one packet, apply one of the following workarounds to avoid poor performance:

- Build a contiguous packet in memory and then send the logical packet to **TCP** on a socket configured with **TCP_NODELAY**.
- Create an I/O vector and pass it to the kernel using the **writev** command on a socket configured with **TCP_NODELAY**.
- Use the **TCP_CORK** option. **TCP_CORK** tells **TCP** to wait for the application to remove the cork before sending any packets. This command causes 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 provide abstractions for layers.

When a logical packet has been built in the kernel by the various components in the application, the socket should be uncorked, allowing **TCP** to send the accumulated logical packet immediately.

29.2. ENABLING TCP_NODELAY

The **TCP_NODELAY** option sends buffer writes to the kernel when events occur, with no delays. Enable **TCP_NODELAY** using the **setsockopt()** function.

Procedure

1. Add the following lines to the **TCP** application's **.c** file.

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

2. Save the file and exit the editor.
3. Apply one of the following workarounds to prevent poor performance.

- Build a contiguous packet in memory and then send the logical packet to **TCP** on a socket configured with **TCP_NODELAY**.
- Create an I/O vector and pass it to the kernel using **writev** on a socket configured with **TCP_NODELAY**.

29.3. ENABLING TCP_CORK

The **TCP_CORK** option prevents **TCP** from sending any packets until the socket is "uncorked".

Procedure

1. Add the following lines to the **TCP** application's **.c** file.

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

2. Save the file and exit the editor.
3. After the logical packet has been built in the kernel by the various components in the application, disable **TCP_CORK**.

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

TCP sends the accumulated logical packet immediately, without waiting for any further packets from the application.

29.4. ADDITIONAL RESOURCES

- the **tcp(7)** man page
- the **setsockopt(3p)** man page
- the **setsockopt(2)** man page

CHAPTER 30. PREVENTING RESOURCE OVERUSE BY USING MUTEX

Mutual exclusion (mutex) algorithms are used to prevent overuse of common resources.

30.1. MUTEX OPTIONS

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.

When you initialize a **pthread_mutex_t** object with the standard attributes, a private, non-recursive, non-robust, and non-priority inheritance-capable mutex is created. This object does not provide any of the benfits provided by the **pthreads** API and the RHEL for Real Time kernel.

To benefit from the **pthreads** API and the RHEL for Real Time kernel, create a **pthread_mutexattr_t** object. This object stores the attributes defined for the futex.

NOTE

The terms **futex** and **mutex** are used to describe POSIX thread (**pthread**) mutex constructs.

30.2. CREATING A MUTEX ATTRIBUTE OBJECT

To define any additional capabilities for the **mutex**, create a **pthread_mutexattr_t** object. This object stores the defined attributes for the futex.

Procedure

- Create the mutex attribute object using one of the following:
  - **pthread_mutex_t(*my_mutex*);**
  - **pthread_mutexattr_t(*&my_mutex_attr*);**
  - **pthread_mutexattr_init(*&my_mutex_attr*);**

For more information about advanced mutex attributes, see Advanced mutex attributes.

NOTE

This section does not include a check of the return value of the function. This is a basic safety procedure that you must always perform.

30.3. CREATING A MUTEX WITH STANDARD ATTRIBUTES

When you initialize a **pthread_mutex_t** object with the standard attributes, a private, non-recursive, non-robust, and non-priority inheritance-capable mutex is created.

Procedure

- Create a mutex object under **pthreads** using one of the following:
  - **pthread_mutex_t(*my_mutex*);**
  - **pthread_mutex_init(*&my_mutex*, *&my_mutex_attr*);**
    where ***&my_mutex_attr;*** is a mutex attribute object.

30.4. ADVANCED MUTEX ATTRIBUTES

The following advanced mutex attributes can be stored in a mutex attribute object:

Mutex attributes

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);

Real-time priority inheritance

You can avoid priority inversion problems by using priority inheritance.
pthread_mutexattr_setprotocol(&my_mutex_attr, PTHREAD_PRIO_INHERIT);

Robust mutexes

When a pthread dies, robust mutexes under the pthread are released. However, this comes with 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);

Mutex initialization

Shared mutexes can be used between processes, however, they can create a lot more overhead.
pthread_mutex_init(&my_mutex_attr, &my_mutex);

30.5. CLEANING UP A MUTEX ATTRIBUTE OBJECT

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

Procedure

- Clean up the attribute object using the **_destroy** command.
  pthread_mutexattr_destroy(*&my_mutex_attr*);

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

30.6. ADDITIONAL RESOURCES

- the **futex(7)** man page
- the **pthread_mutex_destroy(P)** man page
- the **pthread_mutexattr_setprotocol(3p)** man page

- the **pthread_mutexattr_setprioceiling(3p)** man page

CHAPTER 31. ANALYZING APPLICATION PERFORMANCE

Perf is a performance analysis tool. It provides a simple command line interface and abstracts 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 a specific system configuration.

Prerequisites

- The **perf** package must be installed on the system.
- Administrator privileges.

31.1. COLLECTING SYSTEM-WIDE STATISTICS

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

Procedure

- Collect system-wide performance statistics.

```
# 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 **-a** option, and the process was terminated after a few seconds. The results show that it collected 0.725 MB of data and stored it to a newly-created **perf.data** file.

Verification

- Ensure that the results file was created.

```
# ls
perf.data
```

31.2. ARCHIVING PERFORMANCE ANALYSIS RESULTS

You can analyze the results of the **perf** on other systems using the **perf archive** command. This may not be necessary, if:

- Dynamic Shared Objects (DSOs), such as binaries and libraries, are already present in the analysis system, such as the ~/**.debug**/ cache.
- Both systems have the same set of binaries.

Procedure

1. Create an archive of the results from the **perf** command.

```
# perf archive
```

2. Create a tarball from the archive.

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

31.3. ANALYZING PERFORMANCE ANALYSIS RESULTS

The data from the **perf record** feature can now be investigated directly using the **perf report** command.

Procedure

- Analyze the results directly from the **perf.data** file or from an archived 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 the kernel or user space of the process.

The report shows information about the module from which the sample was taken:

- A kernel sample that did not take place in a kernel module is marked with the notation **[kernel.kallsyms]**.
- A kernel sample that took place in the kernel module is 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.
- The result **[k]** indicates kernel space.

Finer grained details are available for review, including data appropriate for experienced **perf** developers.

31.4. LISTING PRE-DEFINED EVENTS

There are a range of available options to get the hardware tracepoint activity.

Procedure

- List pre-defined hardware and software events:

```
# perf list
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]...
```

31.5. GETTING STATISTICS ABOUT SPECIFIED EVENTS

You can view specific events using the **perf stat** command.

Procedure

1. 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.

2. View file system activity by running a script. The following shows an example script:

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

3. In another terminal run the **perf stat** command:

```
# 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.

31.6. ADDITIONAL RESOURCES

- **perf help *COMMAND***
- the **perf**(1) man page

CHAPTER 32. STRESS TESTING REAL-TIME SYSTEMS WITH STRESS-NG

The **stress-ng** tool measures the system's capability to maintain a good level of efficiency under unfavorable conditions. The **stress-ng** tool is a stress workload generator to load and stress all kernel interfaces. It includes a wide range of stress mechanisms known as stressors. Stress testing makes a machine work hard and trip hardware issues such as thermal overruns and operating system bugs that occur when a system is being overworked.

There are over 270 different tests. These include CPU specific tests that exercise floating point, integer, bit manipulation, control flow, and virtual memory tests.

NOTE

Use the **stress-ng** tool with caution as some of the tests can impact the system's thermal zone trip points on a poorly designed hardware. This can impact system performance and cause excessive system thrashing which can be difficult to stop.

32.1. TESTING CPU FLOATING POINT UNITS AND PROCESSOR DATA CACHE

A floating-point unit is the functional part of the processor that performs floating point arithmetic operations. Floating point units handle mathematical operations and make floating numbers or decimal calculations simpler.

Using the **--matrix-method** option, you can stress test the CPU floating point operations and processor data cache.

Prerequisites

- Root privileges for the systems

Procedure

- To test the floating point on one CPU for 60 seconds, use the **--matrix** option:

```
# stress-ng --matrix 1 -t 1m
```

- To run multiple stressors on more than one CPUs for 60 seconds, use the **--times** or **-t** option:

```
# stress-ng --matrix 0 -t 1m

stress-ng --matrix 0 -t 1m --times
stress-ng: info:  [16783] dispatching hogs: 4 matrix
stress-ng: info:  [16783] successful run completed in 60.00s (1 min, 0.00 secs)
stress-ng: info:  [16783] for a 60.00s run time:
stress-ng: info:  [16783] 240.00s available CPU time
stress-ng: info:  [16783] 205.21s user time   ( 85.50%)
stress-ng: info:  [16783] 0.32s system time (  0.13%)
stress-ng: info:  [16783] 205.53s total time  ( 85.64%)
stress-ng: info:  [16783] load average: 3.20 1.25 1.40
```

The special mode with 0 stressors, query the available number of CPUs to run, removing the need to specify the CPU number.

The total CPU time required is 4 x 60 seconds (240 seconds), of which 0.13% is in the kernel, 85.50% is in user time, and **stress-ng** runs 85.64% of all the CPUs.

- To test message passing between processes using a POSIX message queue, use the **-mq** option:

```
# stress-ng --mq 0 -t 30s --times --perf
```

The **mq** option configures a specific number of processes to force context switches using the POSIX message queue. This stress test aims for low data cache misses.

32.2. TESTING CPU WITH MULTIPLE STRESS MECHANISMS

The **stress-ng** tool runs multiple stress tests. In the default mode, it runs the specified stressor mechanisms in parallel.

Prerequisites

- Root privileges for the systems

Procedure

- Run multiple instances of CPU stressors as follows:

```
# stress-ng --cpu 2 --matrix 1 --mq 3 -t 5m
```

In the example, **stress-ng** runs two instances of the CPU stressors, one instance of the matrix stressor and three instances of the message queue stressor to test for five minutes.

- To run all stress tests in parallel, use the **–all** option:

```
# stress-ng --all 2
```

In this example, **stress-ng** runs two instances of all stress tests in parallel.

- To run each different stressor in a specific sequence, use the **--seq** option.

```
# stress-ng --seq 4 -t 20
```

In this example, **stress-ng** runs all the stressors one by one for 20 minutes, with the number of instances of each stressor matching the number of online CPUs.

- To exclude specific stressors from a test run, use the **-x** option:

```
# stress-ng --seq 1 -x numa,matrix,hdd
```

In this example, **stress-ng** runs all stressors, one instance of each, excluding **numa**, **hdd** and **key** stressors mechanisms.

32.3. MEASURING CPU HEAT GENERATION

To measure the CPU heat generation, the specified stressors generate high temperatures for a short time duration to test the system's cooling reliability and stability under maximum heat generation. Using the **--matrix-size** option, you can measure CPU temperatures in degrees Celsius over a short time duration.

Prerequisites

- Root privileges for the system.

Procedure

1. To test the CPU behavior at high temperatures for a specified time duration, run the following command:

```
# stress-ng --matrix 0 --matrix-size 64 --tz -t 60

stress-ng: info:  [18351] dispatching hogs: 4 matrix
stress-ng: info:  [18351] successful run completed in 60.00s (1 min, 0.00 secs)
stress-ng: info:  [18351] matrix:
stress-ng: info:  [18351] x86_pkg_temp   88.00 °C
stress-ng: info:  [18351] acpitz   87.00 °C
```

   In this example, the **stress-ng** configures the processor package thermal zone to reach 88 degrees Celsius over the duration of 60 seconds.

2. (Optional) To print a report at the end of a run, use the **--tz** option:

```
# stress-ng --cpu 0 --tz -t 60

stress-ng: info:  [18065] dispatching hogs: 4 cpu
stress-ng: info:  [18065] successful run completed in 60.07s (1 min, 0.07 secs)
stress-ng: info:  [18065] cpu:
stress-ng: info:  [18065] x86_pkg_temp   88.75 °C
stress-ng: info:  [18065] acpitz   88.38 °C
```

32.4. MEASURING TEST OUTCOMES WITH BOGO OPERATIONS

The **stress-ng** tool can measure a stress test throughput by measuring the bogo operations per second. The size of a bogo operation depends on the stressor being run. The test outcomes are not precise, but they provide a rough estimate of the performance.

You must not use this measurement as an accurate benchmark metric. These estimates help to understand the system performance changes on different kernel versions or different compiler versions used to build **stress-ng**. Use the **--metrics-brief** option to display the total number of available bogo operations and the matrix stressor performance on your machine.

Prerequisites

- Root privileges on the systems

Procedure

- To measure test outcomes with bogo operations, use with the **--metrics-brief** option:

```
# stress-ng --matrix 0 -t 60s --metrics-brief

stress-ng: info: [17579] dispatching hogs: 4 matrix
stress-ng: info: [17579] successful run completed in 60.01s (1 min, 0.01 secs)
stress-ng: info: [17579] stressor bogo ops real time usr time sys time   bogo ops/s bogo ops/s
stress-ng: info:  [17579]                 (secs)   (secs)  (secs)  (real time) (usr+sys time)
stress-ng: info:  [17579] matrix  349322   60.00    203.23   0.19      5822.03      1717.25
```

The **--metrics-brief** option displays the test outcomes and the total number of real-time bogo operations run by the **matrix** stressor for 60 seconds.

32.5. GENERATING A VIRTUAL MEMORY PRESSURE

When under memory pressure, the kernel starts writing pages out to swap. You can stress the virtual memory by using the **--page-in** option to force non-resident pages to swap back into the virtual memory. This causes the virtual machine to be heavily exercised. Using the **--page-in** option, you can enable this mode for the **bigheap**, **mmap** and virtual machine (**vm**) stressors. The **--page-in** option, touch allocated pages that are not in core, forcing them to page in.

Prerequisites

- Root privileges for the system.

Procedure

- To stress test a virtual memory, use the **--page-in** option:

```
# stress-ng --vm 2 --vm-bytes 2G --mmap 2 --mmap-bytes 2G --page-in
```

In this example, **stress-ng** tests memory pressure on a system with 4GB of memory, which is less than the allocated buffer sizes, 2 x 2GB of **vm** stressor and 2 x 2GB of **mmap** stressor with **--page-in** enabled.

32.6. TESTING LARGE INTERRUPTS LOADS ON A DEVICE

Running timers at high frequency can generate a large interrupt load. The **–timer** stressor with an appropriately selected timer frequency can force many interrupts per second.

Prerequisites

- Root privileges on the system.

Procedure

- To generate an interrupt load, use the **--timer** option:

```
# stress-ng --timer 32 --timer-freq 1000000
```

In this example, **stress-ng** tests 32 instances at 1MHz.

32.7. GENERATING MAJOR PAGE FAULTS IN A PROGRAM

With **stress-ng**, you can test and analyze the page fault rate by generating major page faults in a page that are not loaded in the memory. On new kernel versions, the **userfaultfd** mechanism notifies the fault finding threads about the page faults in the virtual memory layout of a process.

Prerequisites

- Root privileges on the system.

Procedure

- To generate major page faults on early kernel versions, use:

```
# stress-ng --fault 0 --perf -t 1m
```

- To generate major page faults on new kernel versions, use:

```
# stress-ng --userfaultfd 0 --perf -t 1m
```

32.8. VIEWING CPU STRESS TEST MECHANISMS

The CPU stress test contains methods to exercise a CPU. You can print an output to view all methods using the **which** option.

If you do not specify the test method, by default, the stressor checks all the stressors in a round-robin fashion to test the CPU with each stressor.

Prerequisites

- Root permissions

Procedure

1. Print all available stressor mechanisms, use the **which** option:

```
# stress-ng --cpu-method which

cpu-method must be one of: all ackermann bitops callfunc cdouble cfloat clongdouble
correlate crc16 decimal32 decimal64 decimal128 dither djb2a double euler explog fft
fibonacci float fnv1a gamma gcd gray hamming hanoi hyperbolic idct int128 int64 int32
```

2. Specify a specific CPU stress method using the **--cpu-method** option:

```
# stress-ng --cpu 1 --cpu-method fft -t 1m
```

32.9. USING THE VERIFY MODE

The **verify** mode validates the results when a test is active. It sanity checks the memory contents from a test run and reports any unexpected failures.

All stressors do not have the **verify** mode and enabling one will reduce the bogo operation statistics because of the extra verification step being run in this mode.

Prerequisites

- Root privileges on the system.

Procedure

- To validate a stress test results, use the **--verify** option:

```
# stress-ng --vm 1 --vm-bytes 2G --verify -v
```

  In this example, **stress-ng** prints the output for an exhaustive memory check on a virtually mapped memory using the **vm** stressor configured with **--verify** mode. It sanity checks the read and write results on the memory.

CHAPTER 33. CREATING AND RUNNING CONTAINERS

This section provides information on creating and running containers with the real time kernel.

Prerequisites

- Install **podman** and other container-related utilities.
- Get familiar with administration and management of Linux containers on RHEL 8.
- Install the **kernel-rt** package and other real time-related packages.

33.1. CREATING A CONTAINER

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

Prerequisites

- Administrator priveleges.

Procedure

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

1. Create the directory you want to use for the container. For example:

```
# mkdir cyclictest
```

2. Change into that directory:

```
# cd cyclictest
```

3. 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*.

4. 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
```

5. Build the container image from the directory containing the Dockerfile:

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

33.2. RUNNING A CONTAINER

You can run a container built with a Dockerfile.

Procedure

1. Run a container using the **podman run** command:

```
# 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
```

This example 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, 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 the cyclictest workload to configure the CPU idle time management). If the specified 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 these, refer to the podman-run(1) manual page. To get a specific workload running inside a container, other **podman-run** options may be helpful.

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

33.3. ADDITIONAL RESOURCES

- Building, running, and managing Linux containers on RHEL 9

- Installing RHEL 9 for Real Time

CHAPTER 34. DISPLAYING THE PRIORITY FOR A PROCESS

You can display information about the priority of a process and information about the scheduling policy for a process using the **sched_getattr** attribute.

Prerequisites

- Administrator privileges

34.1. THE CHRT UTILITY

The **chrt** utility checks and adjusts scheduler policies and priorities. It can start new processes with the desired properties or change the properties of a running process.

Additional resources

- the **chrt(1)** man page

34.2. DISPLAYING THE PROCESS PRIORITY USING THE CHRT UTILITY

You can display the current scheduling policy and scheduling priority for a specified process.

Procedure

- Run the **chrt** utility with the **-p** option, specifying a running process.

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

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

34.3. DISPLAYING THE PROCESS PRIORITY USING SCHED_GETSCHEDULER()

Real-time processes use a set of functions to control policy and priority. You can use the **sched_getscheduler()** function to display the scheduler policy for a specified process.

Procedure

1. Create the **get_sched.c** source file and open it in a text editor.

```
$ {EDITOR} get_sched.c
```

2. Add the following lines into the file.

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

```
int main()
{
  int policy;
  pid_t pid = getpid();

  policy = sched_getscheduler(pid);
  printf("Policy for pid %ld is %i.\n", (long) pid, policy);
  return 0;
}
```

The **policy** variable holds the scheduler policy for the specified process.

3. Compile the program.

```
$ gcc get_sched.c -o get_sched
```

4. Run the program with varying policies.

```
$ chrt -o 0 ./get_sched
Policy for pid 27240 is 0.
$ chrt -r 10 ./get_sched
Policy for pid 27243 is 2.
$ chrt -f 10 ./get_sched
Policy for pid 27245 is 1.
```

Additional resources

- the **sched_getscheduler(2)** man page

34.4. DISPLAYING THE VALID RANGE FOR A SCHEDULER POLICY

You can use the **sched_get_priority_min()** and **sched_get_priority_max()** functions to check the valid priority range for a given scheduler policy.

Procedure

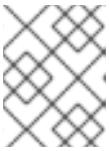
1. Create the **sched_get.c** source file and open it in a text editor.

```
$ {EDITOR} sched_get.c
```

2. Enter the following into the file:

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

int main()
{

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

  printf("Valid priority range for SCHED_FIFO: %d - %d\n",
```

```
        sched_get_priority_min(SCHED_FIFO),
        sched_get_priority_max(SCHED_FIFO));

  printf("Valid priority range for SCHED_RR: %d - %d\n",
        sched_get_priority_min(SCHED_RR),
        sched_get_priority_max(SCHED_RR));
  return 0;
}
```



NOTE

If the specified scheduler policy is not known by the system, the function returns **-1** and **errno** is set to **EINVAL**.

NOTE

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

3. Save the file and exit the editor.
4. Compile the program.

```
$ gcc sched_get.c -o msched_get
```

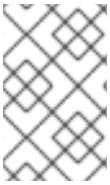
The **sched_get** program is now ready and can be run from the directory in which it is saved.

Additional resources

- the **sched_get_priority_min(2)** man page
- the **sched_get_priority_max(2)** man page

34.5. DISPLAYING THE TIMESLICE FOR A PROCESS

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



NOTE

Though POSIX requires that this function *must* work only with processes that are configured to run with the **SCHED_RR** scheduler policy, the **sched_rr_get_interval()** function can retrieve the timeslice length of any process on Linux.

Timeslice information is returned as a **timespec**. This is the number of seconds and nanoseconds since the base time of 00:00:00 GMT, 1 January 1970:

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

-

Procedure

1. Create the **sched_timeslice.c** source file and open it in a text editor.

```
$ {EDITOR} sched_timeslice.c
```

2. Add the following lines to the **sched_timeslice.c** file.

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

int main()
{
  struct timespec ts;
  int ret;

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

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

  return 0;
}
```

3. Save the file and exit the editor.

4. Compile the program.

```
$ gcc sched_timeslice.c -o sched_timeslice
```

5. Run the program with varying policies and priorities.

```
$ chrt -o 0 ./sched_timeslice
Timeslice: 0.38994072
$ chrt -r 10 ./sched_timeslice
Timeslice: 0.99984800
$ chrt -f 10 ./sched_timeslice
Timeslice: 0.0
```

Additional resources

- the **nice(2)** man page
- the **getpriority(2)** man page
- the **setpriority(2)** man page

34.6. DISPLAYING THE SCHEDULING POLICY AND ASSOCIATED ATTRIBUTES FOR A PROCESS

The **sched_getattr()** function queries the scheduling policy currently applied to the specified process, identified by PID. If PID equals to zero, the policy of the calling process is retrieved.

The **size** argument should reflect the size of the **sched_attr** structure as known to userspace. The kernel fills out **sched_attr::size** to the size of its **sched_attr** structure.

If the input structure is smaller, the kernel returns values outside the provided space. As a result, the system call fails with an **E2BIG** error. The other **sched_attr** fields are filled out as described in The sched_attr structure.

Procedure

1. Create the **sched_timeslice.c** source file and open it in a text editor.

```
$ {EDITOR} sched_timeslice.c
```

2. Add the following lines to the **sched_timeslice.c** file.

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <linux/unistd.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <sys/syscall.h>
#include <pthread.h>

#define gettid() syscall(__NR_gettid)

#define SCHED_DEADLINE    6

/* XXX use the proper syscall numbers */
#ifdef __x86_64__
#define __NR_sched_setattr        314
#define __NR_sched_getattr        315
#endif

struct sched_attr {
    __u32 size;
    __u32 sched_policy;
    __u64 sched_flags;

    /* SCHED_NORMAL, SCHED_BATCH */
    __s32 sched_nice;

    /* SCHED_FIFO, SCHED_RR */
    __u32 sched_priority;

    /* SCHED_DEADLINE (nsec) */
    __u64 sched_runtime;
    __u64 sched_deadline;
    __u64 sched_period;
};

int sched_getattr(pid_t pid,
```

```
        struct sched_attr *attr,
        unsigned int size,
        unsigned int flags)
{
    return syscall(__NR_sched_getattr, pid, attr, size, flags);
}

int main (int argc, char **argv)
{
    struct sched_attr attr;
    unsigned int flags = 0;
    int ret;

    ret = sched_getattr(0, &attr, sizeof(attr), flags);
    if (ret < 0) {
        perror("sched_getattr");
        exit(-1);
    }

    printf("main thread pid=%ld\n", gettid());
    printf("main thread policy=%ld\n", attr.sched_policy);
    printf("main thread nice=%ld\n", attr.sched_nice);
    printf("main thread priority=%ld\n", attr.sched_priority);
    printf("main thread runtime=%ld\n", attr.sched_runtime);
    printf("main thread deadline=%ld\n", attr.sched_deadline);
    printf("main thread period=%ld\n", attr.sched_period);

    return 0;
}
```

3. Compile the **sched_timeslice.c** file.

```
$ gcc sched_timeslice.c -o sched_timeslice
```

4. Check the output of the **sched_timeslice** program.

```
$ ./sched_timeslice
main thread pid=321716
main thread policy=6
main thread nice=0
main thread priority=0
main thread runtime=1000000
main thread deadline=9000000
main thread period=10000000
```

34.7. THE SCHED_ATTR STRUCTURE

The **sched_attr** structure contains or defines a scheduling policy and its associated attributes for a specified thread. The **sched_attr** structure has the following form:

```
struct sched_attr {
 u32 size;
 u32 sched_policy
 u64 sched_flags
```

```
    s32 sched_nice
    u32 sched_priority

    /* SCHED_DEADLINE fields */
            u64 sched_runtime
            u64 sched_deadline
            u64 sched_period
        };
```

sched_attr data structure

size

The thread size in bytes. If the size of the structure is smaller than the kernel structure, additional fields are then assumed to be **0**. If the size is larger than the kernel structure, the kernel verifies all additional fields as **0**.

NOTE

The **sched_setattr()** function fails with **E2BIG** error when **sched_attr** structure is larger than the kernel structure and updates size to contain the size of the kernel structure.

sched_policy

The scheduling policy

sched_flags

Helps control scheduling behavior when a process forks using the **fork()** function. The calling process is referred to as the parent process, and the new process is referred to as the child process. Valid values:

- **0**: The child process inherits the scheduling policy from the parent process.
- **SCHED_FLAG_RESET_ON_FORK: fork()**: The child process does not inherit the scheduling policy from the parent process. Instead, it is set to the default scheduling policy **(struct sched_attr){ .sched_policy = SCHED_OTHER, }**.

sched_nice

Specifies the **nice** value to be set when using **SCHED_OTHER** or **SCHED_BATCH** scheduling policies. The **nice** value is a number in a range from **-20** (high priority) to **+19** (low priority).

sched_priority

Specifies the static priority to be set when scheduling **SCHED_FIFO** or **SCHED_RR**. For other policies, specify priority as **0**.

SCHED_DEADLINE fields must be specified only for deadline scheduling:

- **sched_runtime**: Specifies the **runtime** parameter for deadline scheduling. The value is expressed in nanoseconds.
- **sched_deadline**: Specifies the **deadline** parameter for deadline scheduling. The value is expressed in nanoseconds.
- **sched_period**: Specifies the **period** parameter for deadline scheduling. The value is expressed in nanoseconds.

CHAPTER 35. VIEWING PREEMPTION STATES

Processes using a CPU can give up the CPU they are using, either voluntarily or involuntarily.

35.1. PREEMPTION

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

A process can also involuntarily yield the CPU. This is called preemption and occurs when a higher priority process wants to use the CPU.

Preemption can have a particularly negative impact on system performance, and constant preemption can lead to a state known as thrashing. This problem occurs when processes are constantly preempted, and no process ever runs to completion.

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

35.2. CHECKING THE PREEMPTION STATE OF A PROCESS

You can check the voluntary and involuntary preemption status for a specified process. The statuses are stored in **/proc/*PID*/status**.

Prerequisites

- Administrator privileges.

Procedure

- Display the contents of **/proc/*PID*/status**, where ***PID*** is the ID of the process. The following displays the preemption statuses for the process with PID 1000.

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

CHAPTER 36. SETTING THE PRIORITY FOR A PROCESS WITH THE CHRT UTILITY

You can set the priority for a process using the **chrt** utility.

Prerequisites

- Administrator privileges

36.1. SETTING THE PROCESS PRIORITY USING THE CHRT UTILITY

The **chrt** utility checks and adjusts scheduler policies and priorities. It can start new processes with the desired properties, or change the properties of a running process.

Procedure

- To set the scheduling policy of a process, run the **chrt** command with the appropriate command options and parameters. In the following example, the process ID affected by the command is **1000**, and the priority (**-p**) is **50**.

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

  To start an application with a specified scheduling policy and priority, add the name of the application, and the path to it, if necessary, along with the attributes.

  ```
  # chrt -r -p 50 /bin/my-app
  ```

  For more information on the **chrt** utility options, see The chrt utility options.

36.2. THE CHRT UTILITY OPTIONS

The **chrt** utility options include command options and parameters specifying the process and priority for the command.

Policy options

-f
Sets the scheduler policy to **SCHED_FIFO**.

-o
Sets the scheduler policy to **SCHED_OTHER**.

-r
Sets the scheduler policy to **SCHED_RR** (round robin).

-d
Sets the scheduler policy to **SCHED_DEADLINE**.

-p *n*
Sets the priority of the process to *n*.
When setting a process to SCHED_DEADLINE, you must specify the **runtime**, **deadline**, and **period** parameters.

For example:

```
# chrt -d --sched-runtime 5000000 --sched-deadline 10000000 --sched-period 16666666 0
video_processing_tool
```

where

- **--sched-runtime 5000000** is the run time in nanoseconds.
- **--sched-deadline 10000000** is the relative deadline in nanoseconds.
- **--sched-period 16666666** is the period in nanoseconds.
- **0** is a placeholder for unused priority required by the **chrt** command.

36.3. ADDITIONAL RESOURCES

- the **chrt(1)** man page

CHAPTER 37. SETTING THE PRIORITY FOR A PROCESS WITH LIBRARY CALLS

You can set the priority for a process using the **chrt** utility.

Prerequisites

- Administrator privileges.

37.1. LIBRARY CALLS FOR SETTING PRIORITY

The following library calls are used to set the priority of non-real-time processes.

- **nice**
- **setpriority**

These functions adjust the nice value of a non-real-time process. The **nice** value serves as a suggestion to the scheduler on how to order the list of ready-to-run, non-real-time processes to be run on a processor. The processes at the head of the list run before the ones further down the list.

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

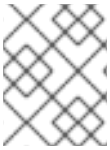
IMPORTANT

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

37.2. SETTING THE PROCESS PRIORITY USING A LIBRARY CALL

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

The **sched_setscheduler()** function requires three parameters, in the form: **sched_setscheduler(pid_t pid, int policy, const struct sched_param *sp);**.



NOTE

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

If the process ID is zero, the **sched_setscheduler()** function acts on the calling process.

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

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

ret = sched_setscheduler(0, SCHED_FIFO, &sp);
```

```
if (ret == -1) {
  perror("sched_setscheduler");
  return 1;
}
```

37.3. SETTING THE PROCESS PRIORITY PARAMETER USING A LIBRARY CALL

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

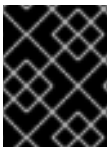
Unlike the **sched_getscheduler()** function, which only returns the scheduling policy, the **sched_getparam()** function returns all scheduling parameters for the given process.

Procedure

Use the following code excerpt that reads the priority of a given real-time process and increments it by two:

```
struct sched_param sp;
int ret;

ret = sched_getparam(0, &sp);
sp.sched_priority += 2;
ret = sched_setparam(0, &sp);
```

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



IMPORTANT

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

37.4. SETTING THE SCHEDULING POLICY AND ASSOCIATED ATTRIBUTES FOR A PROCESS

The **sched_setattr()** function sets the scheduling policy and its associated attributes for an instance ID specified in PID. When pid=0, **sched_setattr()** acts on the process and attributes of the calling thread.

Procedure

- Call **sched_setattr()** specifying the process ID on which the call acts and one of the following real-time scheduling policies:

Real-time scheduling policies

SCHED_FIFO

Schedules a first-in and first-out policy.

SCHED_RR

Schedules a round-robin policy.

SCHED_DEADLINE

Schedules a deadline scheduling policy.

Linux also supports the following non-real-time scheduling policies:

Non-real-time scheduling policies

SCHED_OTHER

Schedules the standard round-robin time-sharing policy.

SCHED_BATCH

Schedules a “batch" style execution of processes.

SCHED_IDLE

Schedules very low priority background jobs. **SCHED_IDLE** can be used only at static priority **0**, and the nice value has no influence for this policy.
This policy is intended for running jobs at extremely low priority (lower than a +19 nice value using **SCHED_OTHER** or **SCHED_BATCH** policies).

37.5. ADDITIONAL RESOURCES

- The sched_attr-structure