Red Hat Enterprise Linux for Real Time

9

Optimizing RHEL 9 for Real Time for low latency operation

Optimizing the RHEL for Real Time kernel on Red Hat Enterprise Linux
Red Hat Enterprise Linux for Real Time 9 Optimizing RHEL 9 for Real Time for low latency operation

Optimizing the RHEL for Real Time kernel on Red Hat Enterprise Linux
Abstract

Tune your workstations on the RHEL for Real Time kernel to achieve consistently low latency and a predictable response time on latency-sensitive applications. Perform real-time kernel tuning by managing system resources, measuring latency between events, and recording latency for analysis on applications with strict determinism requirements.
Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE ................................................................. 7

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION ....................................... 8

CHAPTER 1. REAL-TIME KERNEL TUNING IN RHEL 9 ............................................... 9
1. TUNING GUIDELINES 9
2. THREAD SCHEDULING POLICIES 10
3. BALANCING LOGGING PARAMETERS 10
4. IMPROVING PERFORMANCE BY AVOIDING RUNNING UNNECESSARY APPLICATIONS 11
5. NON-UNIFORM MEMORY ACCESS 12
6. ENSURING THAT DEBUGFS IS MOUNTED 12
7. INFINIBAND IN RHHEL FOR REAL TIME 13
8. USING ROCEE AND HIGH-PERFORMANCE NETWORKING 13
9. TUNING CONTAINERS FOR RHHEL FOR REAL-TIME 13

CHAPTER 2. SCHEDULING POLICIES FOR RHHEL FOR REAL TIME ...................... 15
1. SCHEDULER POLICIES 15
2. PARAMETERS FOR SCHED_DEADLINE POLICY 16

CHAPTER 3. SETTING PERSISTENT KERNEL TUNING PARAMETERS ...................... 17
1. MAKING PERSISTENT KERNEL TUNING PARAMETER CHANGES 17

CHAPTER 4. APPLICATION TUNING AND DEPLOYMENT ......................................... 18
1. SIGNAL PROCESSING IN REAL-TIME APPLICATIONS 18
2. SYNCHRONIZING THREADS 18
3. REAL-TIME SCHEDULER PRIORITIES 19
4. LOADING DYNAMIC LIBRARIES 19

CHAPTER 5. SETTING BIOS PARAMETERS FOR SYSTEM TUNING ......................... 21
1. DISABLING POWER MANAGEMENT TO IMPROVE RESPONSE TIMES 21
2. IMPROVING RESPONSE TIMES BY DISABLING ERROR DETECTION AND CORRECTION UNITS 21
3. IMPROVING RESPONSE TIME BY CONFIGURING SYSTEM MANAGEMENT INTERRUPTS 21

CHAPTER 6. RUNTIME VERIFICATION OF THE REAL-TIME KERNEL ..................... 23
1. RUNTIME MONITORS AND REACTORS 23
2. ONLINE RUNTIME MONITORS 23
3. THE USER INTERFACE 23

CHAPTER 7. RUNNING AND INTERPRETING HARDWARE AND FIRMWARE LATENCY TESTS ................................................................. 25
1. RUNNING HARDWARE AND FIRMWARE LATENCY TESTS 25
2. INTERPRETING HARDWARE AND FIRMWARE LATENCY TEST RESULTS 26

CHAPTER 8. RUNNING AND INTERPRETING SYSTEM LATENCY TESTS .................. 29
1. PREREQUISITES 29
2. RUNNING SYSTEM LATENCY TESTS 29

CHAPTER 9. SETTING CPU AFFINITY ON RHHEL FOR REAL TIME ..................... 31
1. TUNING PROCESSOR AFFINITY USING THE TASKSET COMMAND 31
2. SETTING PROCESSOR AFFINITY USING THE SCHED_SETAFFINITY() SYSTEM CALL 32
3. ISOLATING A SINGLE CPU TO RUN HIGH UTILIZATION TASKS 33
4. REDUCING CPU PERFORMANCE SPIKES 34
5. LOWERING CPU USAGE BY DISABLING THE PC CARD DAEMON 35

CHAPTER 10. USING MLOCK() SYSTEM CALLS ON RHHEL FOR REAL TIME .............. 37
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>MLOCK() AND MUNLOCK() SYSTEM CALLS</td>
<td>37</td>
</tr>
<tr>
<td>10.2</td>
<td>USING MLOCK() SYSTEM CALLS TO LOCK PAGES</td>
<td>37</td>
</tr>
<tr>
<td>10.3</td>
<td>USING MLOCKALL() SYSTEM CALLS TO LOCK ALL MAPPED PAGES</td>
<td>38</td>
</tr>
<tr>
<td>10.4</td>
<td>USING MMAP() SYSTEM CALLS TO MAP FILES OR DEVICES INTO MEMORY</td>
<td>39</td>
</tr>
<tr>
<td>10.5</td>
<td>PARAMETERS FOR MLOCK() SYSTEM CALLS</td>
<td>40</td>
</tr>
<tr>
<td>11.1</td>
<td>CONFIGURING THE TIMERLAT TRACER TO MEASURE SCHEDULING LATENCY</td>
<td>42</td>
</tr>
<tr>
<td>11.2</td>
<td>THE TIMERLAT TRACER OPTIONS</td>
<td>42</td>
</tr>
<tr>
<td>11.3</td>
<td>MEASURING TIMER LATENCY WITH RTLA-TIMERLAT-TOP</td>
<td>43</td>
</tr>
<tr>
<td>11.4</td>
<td>THE RTLA TIMERLAT TOP TRACER OPTIONS</td>
<td>43</td>
</tr>
<tr>
<td>12.1</td>
<td>THE RTLA-OSNOISE TRACER</td>
<td>44</td>
</tr>
<tr>
<td>12.2</td>
<td>CONFIGURING THE RTLA-OSNOISE TRACER TO MEASURE SCHEDULING LATENCY</td>
<td>45</td>
</tr>
<tr>
<td>12.3</td>
<td>THE RTLA-OSNOISE OPTIONS FOR CONFIGURATION</td>
<td>45</td>
</tr>
<tr>
<td>12.4</td>
<td>THE RTLA-OSNOISE TRACEPOINTS</td>
<td>46</td>
</tr>
<tr>
<td>12.5</td>
<td>THE RTLA-OSNOISE TRACER OPTIONS</td>
<td>46</td>
</tr>
<tr>
<td>12.6</td>
<td>MEASURING OPERATING SYSTEM NOISE WITH THE RTLA-OSNOISE-TOP TRACER</td>
<td>46</td>
</tr>
<tr>
<td>12.7</td>
<td>THE RTLA-OSNOISE-TOP TRACER OPTIONS</td>
<td>47</td>
</tr>
<tr>
<td>13.1</td>
<td>DISABLING ATIME</td>
<td>49</td>
</tr>
<tr>
<td>13.2</td>
<td>ADDITIONAL RESOURCES</td>
<td>49</td>
</tr>
<tr>
<td>14.1</td>
<td>DISABLING GRAPHICS CONSOLE LOGGING TO GRAPHICS ADAPTER</td>
<td>50</td>
</tr>
<tr>
<td>14.2</td>
<td>DISABLING MESSAGES FROM PRINTING ON GRAPHICS CONSOLE</td>
<td>50</td>
</tr>
<tr>
<td>15.1</td>
<td>HARDWARE CLOCKS</td>
<td>52</td>
</tr>
<tr>
<td>15.2</td>
<td>VIEWING THE AVAILABLE CLOCK SOURCES IN YOUR SYSTEM</td>
<td>52</td>
</tr>
<tr>
<td>15.3</td>
<td>VIEWING THE CLOCK SOURCE CURRENTLY IN USE</td>
<td>52</td>
</tr>
<tr>
<td>15.4</td>
<td>TEMPORARILY CHANGING THE CLOCK SOURCE TO USE</td>
<td>52</td>
</tr>
<tr>
<td>15.5</td>
<td>COMPARING THE COST OF READING HARDWARE CLOCK SOURCES</td>
<td>54</td>
</tr>
<tr>
<td>15.6</td>
<td>SYNCHRONIZING THE TSC TIMER ON OPTERON CPUS</td>
<td>55</td>
</tr>
<tr>
<td>15.7</td>
<td>THE CLOCK_TIMING PROGRAM</td>
<td>55</td>
</tr>
<tr>
<td>16.1</td>
<td>POWER SAVING STATES</td>
<td>57</td>
</tr>
<tr>
<td>16.2</td>
<td>CONFIGURING POWER MANAGEMENT STATES</td>
<td>57</td>
</tr>
<tr>
<td>17.1</td>
<td>INTERRUPT AND PROCESS BINDING</td>
<td>59</td>
</tr>
<tr>
<td>17.2</td>
<td>DISABLING THE IRQBALANCE DAEMON</td>
<td>59</td>
</tr>
<tr>
<td>17.3</td>
<td>EXCLUDING CPUS FROM IRQ BALANCING</td>
<td>60</td>
</tr>
<tr>
<td>17.4</td>
<td>MANUALLY ASSIGNING CPU AFFINITY TO INDIVIDUALIRQS</td>
<td>61</td>
</tr>
<tr>
<td>17.5</td>
<td>BINDING PROCESSES TO CPUS WITH THE TASKSET UTILITY</td>
<td>62</td>
</tr>
<tr>
<td>18.1</td>
<td>PREREQUISITES</td>
<td>64</td>
</tr>
<tr>
<td>18.2</td>
<td>CHANGING THE OUT OF MEMORY VALUE</td>
<td>64</td>
</tr>
<tr>
<td>18.3</td>
<td>PRIORITIZING PROCESSES TO KILL WHEN IN AN OUT OF MEMORY STATE</td>
<td>64</td>
</tr>
<tr>
<td>18.4</td>
<td>DISABLING THE OUT OF MEMORY KILLER FOR A PROCESS</td>
<td>65</td>
</tr>
<tr>
<td>19.1</td>
<td>IMPROVING LATENCY USING THE TUNA CLI</td>
<td>67</td>
</tr>
</tbody>
</table>
19.1. PREREQUISITES
19.2. THE TUNA CLI
19.3. ISOLATING CPUS USING THE TUNA CLI
19.4. MOVING INTERRUPTS TO SPECIFIED CPUS USING THE TUNA CLI
19.5. CHANGING PROCESS SCHEDULING POLICIES AND PRIORITIES USING THE TUNA CLI

CHAPTER 20. SETTING SCHEDULER PRIORITIES ............................................. 71
20.1. VIEWING THREAD SCHEDULING PRIORITIES ............................... 71
20.2. CHANGING THE PRIORITY OF SERVICES DURING BOOTING .......... 71
20.3. CONFIGURING THE CPU USAGE OF A SERVICE ........................... 73
20.4. PRIORITY MAP ........................................................................ 73
20.5. ADDITIONAL RESOURCES ......................................................... 74

CHAPTER 21. NETWORK DETERMINISM TIPS ............................................ 75
21.1. COALESCING INTERRUPTS .......................................................... 75
21.2. AVOIDING NETWORK CONGESTION ........................................ 76
21.3. MONITORING NETWORK PROTOCOL STATISTICS ....................... 76
21.4. ADDITIONAL RESOURCES ......................................................... 77

CHAPTER 22. TRACING LATENCIES WITH TRACE-CMD .......................... 78
22.1. INSTALLING TRACE-CMD .......................................................... 78
22.2. RUNNING TRACE-CMD ............................................................ 78
22.3. TRACE-CMD EXAMPLES .......................................................... 78
22.4. ADDITIONAL RESOURCES ......................................................... 79

CHAPTER 23. ISOLATING CPUS USING TUNED-PROFILES-REAL-TIME ...... 80
23.1. CHOOSING CPUS TO ISOLATE .................................................. 80
23.2. ISOLATING CPUS USING TUNED'S ISOLATED_CORES OPTION .... 81
23.3. ISOLATING CPUS USING THE NOHZ AND NOHZ_FULL PARAMETERS 83

CHAPTER 24. LIMITING SCHED_OTHER TASK MIGRATION .................... 84
24.1. TASK MIGRATION .................................................................. 84
24.2. LIMITING SCHED_OTHER TASK MIGRATION USING THE SCHED_NR_MIGRATE VARIABLE 84

CHAPTER 25. REDUCING TCP PERFORMANCE SPIKES ............................ 85
25.1. TURNING OFF TCP TIMESTAMPs ............................................... 85
25.2. TURNING ON TCP TIMESTAMPs ............................................... 85
25.3. DISPLAYING THE TCP TIMESTAMP STATUS .............................. 85

CHAPTER 26. IMPROVING CPU PERFORMANCE BY USING RCU CALLBACKS 87
26.1. OFFLOADING RCU CALLBACKS ................................................. 87
26.2. MOVING RCU CALLBACKS ..................................................... 87
26.3. RELIEVING CPUS FROM AWAKENING RCU OFFLOAD THREADS .... 88
26.4. ADDITIONAL RESOURCES ......................................................... 88

CHAPTER 27. TRACING LATENCIES USING FTRACE ............................... 89
27.1. USING THE FTRACE UTILITY TO TRACE LATENCIES ................. 89
27.2. FTRACE FILES ................................................................... 91
27.3. FTRACE TRACERS .................................................................. 91
27.4. FTRACE EXAMPLES ................................................................. 92

CHAPTER 28. APPLICATION TIMESTAMPING ......................................... 94
28.1. POSIX CLOCKS ................................................................... 94
28.2. THE _COARSE CLOCK VARIANT IN CLOCK_GETTIME ............... 94
28.3. ADDITIONAL RESOURCES ......................................................... 95
CHAPTER 37. SETTING THE PRIORITY FOR A PROCESS WITH LIBRARY CALLS ........................................ 121
37.1. LIBRARY CALLS FOR SETTING PRIORITY .......................... 121
37.2. SETTING THE PROCESS PRIORITY USING A LIBRARY CALL ........................................ 121
37.3. SETTING THE PROCESS PRIORITY PARAMETER USING A LIBRARY CALL .................. 122
37.4. SETTING THE SCHEDULING POLICY AND ASSOCIATED ATTRIBUTES FOR A PROCESS .. 122
37.5. ADDITIONAL RESOURCES ............................................. 123

CHAPTER 38. SCHEDULING PROBLEMS ON THE REAL-TIME KERNEL AND SOLUTIONS .............. 124
38.1. SCHEDULING POLICIES FOR THE REAL-TIME KERNEL ........................................ 124
38.2. SCHEDULER THROTTLING IN THE REAL-TIME KERNEL ........................................ 124
38.3. THREAD STARVATION IN THE REAL-TIME KERNEL ........................................ 125
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 feedback through Jira (account required)

1. Log in to the Jira website.
2. Click Create in the top navigation bar.
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 Create at the bottom of the dialogue.
CHAPTER 1. REAL-TIME KERNEL TUNING IN RHEL 9

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 tuned so that latency meets those requirements. The RHEL for Real Time kernel provides seamless integration with RHEL 9 and offers clients the opportunity to measure, configure, and record latency times within their organization.

Use the RHEL for Real Time kernel on well-tuned systems, for applications with extremely high determinism requirements. With the kernel system tuning, you can achieve good improvement in determinism. Before you begin, perform general system tuning of the standard RHEL 9 system and then deploy the RHEL for Real Time kernel.

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 particular set of tuning changes. Make the length of your test runs adjustable and run them for longer than a few minutes. You can 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 detect 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 can vary. 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 tuning parameter 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 problems.

- 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.

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

The example above configures the client system to log all kernel messages to the remote machine at @my.remote.logging.server.
Alternatively, you can configure `syslogd` to log all locally generated system messages, by adding the following line to the `/etc/rsyslog.conf` file:

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

**IMPORTANT**

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

**Additional resources**

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

### 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**

- You have root permissions on 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.
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.

   ```bash
   # 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

- [cron(8) man page](https://manpages.debian.org/sid/cron.8.en)

### 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

- [numactl(8) man page](https://manpages.debian.org/sid/numactl.8.en)

### 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:

  ```bash
  # mount | grep '^debugfs
ddebugfs 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 9. 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. TUNING CONTAINERS FOR RHEL FOR REAL-TIME

When testing the real-time workload in a container running on the main RHEL kernel, add the following options to the `podman run` command as necessary:

- `--cpuset-cpus=<cpu_list>` specifies the list of isolated CPU cores to use. If you have more than one CPU, use a comma-separated or a hyphen-separated range of CPUs that a container can use.

- `--cpuset-mems=<number-of-memory-nodes>` specifies Non-Uniform Memory Access (NUMA) memory nodes to use, and, therefore avoids cross-NUMA node memory access.

- `--memory-reservation=<limit> <my_rt_container_image>` verifies that the minimal amount of memory required by the real-time workload running on the container, is available at container start time.

**Procedure**

- Start the real-time workloads in a container:
# podman run --cpuset-cpus=<cpu_list> --cpuset-mems=<number_of_memory_nodes> --memory-reservation=<limit> <my_rt_container_image>

Additional resources

- podman-run(1) man page
CHAPTER 2. SCHEDULING POLICIES FOR RHEL FOR REAL TIME

In real-time, the scheduler is the kernel component that determines the runnable thread to run. Each thread has an associated scheduling policy and a static scheduling priority, known as `sched_priority`. The scheduling is preemptive and therefore the currently running thread stops when a thread with a higher static priority gets ready to run. The running thread then returns to the `waitlist` for its static priority.

All Linux threads have one of the following scheduling policies:

- **SCHED_OTHER** or **SCHED_NORMAL**: is the default policy
- **SCHED_BATCH**: is similar to **SCHED_OTHER**, but with incremental orientation
- **SCHED_IDLE**: is the policy with lower priority than **SCHED_OTHER**.
- **SCHED_FIFO**: is the first in and first out real-time policy.
- **SCHED_RR**: is the round-robin real-time policy.
- **SCHED_DEADLINE**: is a scheduler policy to prioritize tasks according to the job deadline. The job with the earliest absolute deadline runs first.

2.1. SCHEDULER POLICIES

The real-time threads have higher priority than the standard threads. The policies have scheduling priority values that range from the minimum value of 1 to the maximum value of 99.

The following policies are critical to real-time:

- **SCHED_OTHER** or **SCHED_NORMAL** policy
  This is the default scheduling policy for Linux threads. It has a dynamic priority that is changed by the system based on the characteristics of the thread. **SCHED_OTHER** threads have nice values between 20, which is the highest priority and 19, which is the lowest priority. The default nice value for **SCHED_OTHER** threads is 0.

- **SCHED_FIFO** policy
  Threads with **SCHED_FIFO** run with higher priority over **SCHED_OTHER** tasks. Instead of using nice values, **SCHED_FIFO** uses a fixed priority between 1, which is the lowest and 99, which is the highest. A **SCHED_FIFO** thread with a priority of 1 always schedules first over a **SCHED_OTHER** thread.

- **SCHED_RR** policy
  The **SCHED_RR** policy is similar to the **SCHED_FIFO** policy. The threads of equal priority are scheduled in a round-robin fashion. **SCHED_FIFO** and **SCHED_RR** threads run until one of the following events occurs:
  - The thread goes to sleep or waits for an event.
  - A higher-priority real-time thread gets ready to run.
    Unless one of the above events occurs, the threads run indefinitely on the specified processor, while the lower-priority threads remain in the queue waiting to run. This might cause the system service threads to be resident and prevent being swapped out and fail the filesystem data flushing.
• **SCHED_DEADLINE** policy

  The **SCHED_DEADLINE** policy specifies the timing requirements. It schedules each task according to the task’s deadline. The task with the earliest deadline first (EDF) schedule runs first.

  The kernel requires runtime $\leq$ deadline $\leq$ period to be true. The relation between the required options is runtime $\leq$ deadline $\leq$ period.

### 2.2. PARAMETERS FOR SCHED_DEADLINE POLICY

Each **SCHED_DEADLINE** task is characterized by **period**, **runtime**, and **deadline** parameters. The values for these parameters are integers of nanoseconds.

<table>
<thead>
<tr>
<th>Table 2.1. SCHED_DEADLINE parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameter</strong></td>
</tr>
<tr>
<td>period</td>
</tr>
<tr>
<td>runtime</td>
</tr>
<tr>
<td>deadline</td>
</tr>
</tbody>
</table>
CHAPTER 3. 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.

3.1. 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

- You have root permissions on the system.

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 4. 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.

4.1. 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

4.2. 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:

```c
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

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

### 4.3. 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.

### 4.4. 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.

```bash
#!/bin/sh
LD_BIND_NOW=1
export LD_BIND_NOW
chrt --fifo 1 _/opt/myapp/myapp-server &
```

Additional resources

- `ld.so(8)` man page
CHAPTER 5. SETTING BIOS PARAMETERS FOR SYSTEM TUNING

The BIOS plays a key role in the functioning of the system. By configuring the BIOS parameters correctly you can significantly improve the system performance.

NOTE

Every system and BIOS vendor uses different terms and navigation methods. For more information about BIOS settings, see the BIOS documentation or contact the BIOS vendor.

5.1. 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.

5.2. 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.

5.3. IMPROVING RESPONSE TIME BY CONFIGURING SYSTEM MANAGEMENT INTERRUPTIONS

System Management Interrupts (SMIs) are a hardware vendor’s 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 6. RUNTIME VERIFICATION OF THE REAL-TIME KERNEL

Runtime verification is a lightweight and rigorous method to check the behavioral equivalence between system events and their formal specifications. Runtime verification has monitors integrated in the kernel that attach to tracepoints. If a system state deviates from defined specifications, the runtime verification program activates reactors to inform or enable a reaction, such as capturing the event in log files or a system shutdown to prevent failure propagation in an extreme case.

6.1. RUNTIME MONITORS AND REACTORS

The runtime verification (RV) monitors are encapsulated inside the RV monitor abstraction and coordinate between the defined specifications and the kernel trace to capture runtime events in trace files. The RV monitor includes:

- Reference Model is a reference model of the system
- Monitor Instance(s) is a set of instance for a monitor, such as a per-CPU monitor or a per-task monitor.
- Helper functions that connect the monitor to the system.

In addition to verifying and monitoring a system at runtime, you can enable a response to an unexpected system event. The forms of reaction can vary from capturing an event in the trace file to initiating an extreme reaction, such as a shutdown to avoid a system failure on safety critical systems.

Reactors are reaction methods available for RV monitors to define reactions to system events as required. By default, monitors provide a trace output of the actions.

6.2. ONLINE RUNTIME MONITORS

Runtime verification (RV) monitors are classified into following types:

- Online monitors capture events in the trace while the system is running. Online monitors are synchronous if the event processing is attached to the system execution. This will block the system during the event monitoring. Online monitors are asynchronous, if the execution is detached from the system and is run on a different machine. This however requires saved execution log files.

- Offline monitors process traces that are generated after the events have occurred. Offline runtime verification capture information by reading the saved trace log files generally from a permanent storage. Offline monitors can work only if you have the events saved in a file.

6.3. THE USER INTERFACE

The user interface is located at `/sys/kernel/tracing/rv` and resembles the tracing interface. The user interface includes the mentioned files and folders.

<table>
<thead>
<tr>
<th>Settings</th>
<th>Description</th>
<th>Example commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>available_monitors</td>
<td>Displays the available monitors one per line.</td>
<td># cat available_monitors</td>
</tr>
<tr>
<td>Settings</td>
<td>Description</td>
<td>Example commands</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>available_reactors</td>
<td>Display the available reactors one per line.</td>
<td># cat available_reactors</td>
</tr>
</tbody>
</table>
| enabled_monitors  | Displays enabled monitors one per line. You can enable more than one monitor at the same time. Writing a monitor name with a '!' prefix disables the monitor and truncating the file disables all enabled monitors. | # cat enabled_monitors  
|                  |                                                                             | # echo wip > enabled_monitors                         |
|                  |                                                                             | # echo '!wip'>> enabled_monitors                      |
| monitors/        | The monitors/ directory resembles the events directory on the tracefs file system with each monitor having its own directory inside monitors/. | # cd monitors/wip/                                    |
| monitors/MONITOR/reactors | Lists available reactors with the select reaction for a specific MONITOR inside '[]'. The default is the no operation (nop) reactor. Writing the name of a reactor integrates it to a specific MONITOR. | # cat monitors/wip/reactors |
| monitoring_on    | Initiates the tracing_on and the tracing_off switcher in the trace interface. Writing 0 stops the monitoring and 1 continues the monitoring. The switcher does not disable enabled monitors but stops the per-entity monitors from monitoring the events. |                                                     |
| reacting_on      | Enables reactors. Writing 0 disables reactions and 1 enables reactions.      |                                                     |
| monitors/MONITOR/desc | Displays the Monitor description |                                                     |
| monitors/MONITOR/enable | Displays the current status of the Monitor. Writing 0 disables the Monitor and 1 enables the Monitor. |                                                     |
CHAPTER 7. 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 realtime-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.

7.1. 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 realtime-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.

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

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

**Additional resources**

- [hwlatdetect man page](#).
- [Interpreting hardware and firmware latency tests](#).

### 7.2. INTERPRETING HARDWARE AND FIRMWARE LATENCY TEST RESULTS

The hardware latency detector (**hwlatdetect**) uses the tracer mechanism to detect latencies introduced by the hardware architecture or BIOS/EFI firmware. By checking the latencies measured by **hwlatdetect**, you can determine if a potential hardware is suitable to support the RHEL for Real Time kernel.

**Examples**

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

```bash
# 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 example 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:

```bash
# 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----
```

The output shows that during the consecutive reads of the system `clocksource`, there were 10 delays that showed up in the 15-18 us range.

NOTE

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

Understanding the results

The table for Testing method, parameters, and results, lists the parameters and the latency values detected by the `hwlatdetect` utility.

Table 7.1. Testing method, parameters, and results

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test duration</td>
<td>10 seconds</td>
<td>The duration of the test in seconds</td>
</tr>
<tr>
<td>detector</td>
<td>tracer</td>
<td>The utility that runs the detector thread</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Latency threshold</td>
<td>10us</td>
<td>The maximum allowable latency</td>
</tr>
<tr>
<td>Sample window</td>
<td>1000000us</td>
<td>1 second</td>
</tr>
<tr>
<td>Sample width</td>
<td>500000us</td>
<td>0.05 seconds</td>
</tr>
<tr>
<td>Non-sampling period</td>
<td>500000us</td>
<td>0.05 seconds</td>
</tr>
<tr>
<td>Output File</td>
<td>None</td>
<td>The file to which the output is saved.</td>
</tr>
</tbody>
</table>

**Results**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Latency</td>
<td>18us</td>
<td>The highest latency during the test that exceeded the Latency threshold. If no sample exceeded the Latency threshold, the report shows Below threshold.</td>
</tr>
<tr>
<td>Samples recorded</td>
<td>10</td>
<td>The number of samples recorded by the test.</td>
</tr>
<tr>
<td>Samples exceeding threshold</td>
<td>10</td>
<td>The number of samples recorded by the test where the latency exceeded the Latency threshold.</td>
</tr>
<tr>
<td>SMIs during run</td>
<td>0</td>
<td>The number of System Management Interrupts (SMIs) that occurred during the test run.</td>
</tr>
</tbody>
</table>

**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 8. RUNNING AND INTERPRETING SYSTEM LATENCY TESTS

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

8.1. PREREQUISITES

- The RHEL for Real Time package group is installed.
- You have root permissions on the system.

8.2. 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.
- You have root permissions on 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

```bash
# rteval --summarize rteval-<date>-N.tar.bz2
```
CHAPTER 9. 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. By setting processor affinity, along with effective policy and priority settings, you can achieve 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, such as 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 can 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.

9.1. 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 can contain more than one items, separated by comma, and a range of processors. For example, 0,5,7,9-11.

Prerequisites

- You have root permissions on the system.
Procedure

- To verify 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 set up to use 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.

9.2. 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 permissions on the system.

Procedure

- To set the processor affinity with sched_setaffinity():

```c
#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 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);

---

9.3. ISOLATING A SINGLE CPU TO RUN HIGH UTILIZATION TASKS

Using the real-time **cpuset** 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**

- You have root permissions on the system.

**Procedure**

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

   ```
   # cd /sys/fs/cgroup/cpu
   # 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:
5. Create a partition named as `cpuset` and assign the high utilization task:

   ```bash
   # 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:

   ```bash
   # 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.

### 9.4. REDUCING CPU PERFORMANCE SPIKES

A common source of latency spikes is when multiple CPUs contend on common locks in the kernel timer tick handler. The usual lock responsible for the contention is `xtime_lock`, which is used by the timekeeping system and the Read-Copy-Update (RCU) structure locks. By using `skew_tick=1`, you can offset the timer tick per CPU to start at a different time and avoid potential lock conflicts.

The `skew_tick` kernel command line parameter might prevent latency fluctuations on moderate to large systems with large core-counts and have latency-sensitive workloads.

#### Prerequisites

- You have administrator permissions.

#### Procedure

1. Enable the `skew_tick=1` parameter with `grubby`.

   ```bash
   # grubby --update-kernel=ALL --args="skew_tick=1"
   ```

2. Reboot for changes to take effect.

   ```bash
   # reboot
   ```

**NOTE**

Enabling `skew_tick=1` causes a significant increase in power consumption and, therefore, you must enable the `skew` boot parameter only if you are running latency sensitive real-time workloads and consistent latency is an important consideration over power consumption.

#### Verification

Display the `/proc/cmdline` file and ensure `skew_tick=1` is specified. The `/proc/cmdline` file shows the parameters passed to the kernel.
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. Therefore, the additional background noise can lead to higher preemption costs to real-time tasks and other undesirable impacts on determinism.

**Prerequisites**

- You have 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 **pcscd** daemon.

2. If the **pcscd** 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 **pcscd** 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
   ```
2. Ensure that the value for the **Active** parameter is **inactive (dead)**.
CHAPTER 10. 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.

10.1. 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.

10.2. 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_memalign() 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:

```c
#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.

**10.3. 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 permissions on the system.

**Procedure**

- To use `mlockall()` and `munlockall()` real-time system calls:
  - Lock all mapped pages by using `mlockall()` system call:
    ```c
    #include <sys/mman.h>
    int mlockall (int flags)
    ```
  - Unlock all mapped pages by using `munlockall()` system call:
    ```c
    #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

### 10.4. 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 memory space. You can assign and lock memory areas by setting `MAP_LOCKED` in the `flags` parameter. As `mmap()` assigns memory on a page basis, it avoids two locks on the same page, which prevents the double-lock or single-unlock problems.

**Prerequisites**

- You have root permissions on the system.

**Procedure**

- To map a specific process-address space:
  ```c
  #include <sys/mman.h>
  #include <stdlib.h>
  
  void *alloc_workbuf(size_t size)
  {
      void *ptr;
      
      ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, 
  ```
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

- `mmap(2)` man page
- `mlockall(2)` man page

### 10.5. PARAMETERS FOR MLOCK() SYSTEM CALLS

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

#### Table 10.1. mlock parameters

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>prot</strong></td>
<td><code>mmap</code> and <code>munmap</code> calls define the desired memory protection with this parameter. <strong>prot</strong> takes one or a combination of <code>PROT_EXEC</code>, <code>PROT_READ</code>, <code>PROT_WRITE</code> or <code>PROT_NONE</code> values.</td>
</tr>
<tr>
<td><strong>flags</strong></td>
<td>Controls the mapping visibility to other processes that map the same file. It takes one of the values: <code>MAP_ANONYMOUS</code>, <code>MAP_LOCKED</code>, <code>MAP_PRIVATE</code> or <code>MAP_SHARED</code> values.</td>
</tr>
<tr>
<td><strong>MCL_CURRENT</strong></td>
<td>Locks all pages that are currently mapped into a process.</td>
</tr>
<tr>
<td><strong>MCL_FUTURE</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
CHAPTER 11. MEASURING SCHEDULING LATENCY USING TIMERLAT IN RHEL FOR REAL TIME

The rtlata-timerlat tool is an interface for the timerlat tracer. The timerlat tracer finds sources of wake-up latencies for real-time threads. The timerlat tracer creates a kernel thread per CPU with a real-time priority and these threads set a periodic timer to wake up and go back to sleep. On a wake up, timerlat finds and collects information, which is useful to debug operating system timer latencies. The timerlat tracer generates an output and prints the following two lines at every activation:

- The timerlat tracer periodically prints the timer latency seen at timer interrupt requests (IRQs) handler. This is the first output seen at the hardirq context before a thread activation.
- The second output is the timer latency of a thread. The ACTIVATION ID field displays the interrupt requests (IRQs) performance to its respective thread execution.

11.1. CONFIGURING THE TIMERLAT TRACER TO MEASURE SCHEDULING LATENCY

You can configure the timerlat tracer by adding timerlat in the current_tracer file of the tracing system. The current_tracer file is generally mounted in the /sys/kernel/tracing directory. The timerlat tracer measures the interrupt requests (IRQs) and saves the trace output for analysis when a thread latency is more than 100 microseconds.

Procedure

1. List the current tracer:

   ```
   # cat /sys/kernel/tracing/current_tracer
   nop
   ```

   The no operations (nop) is the default tracer.

2. Add the timerlat tracer in the current_tracer file of the tracing system:

   ```
   # cd /sys/kernel/tracing/
   # echo timerlat > current_tracer
   ```

3. Generate a tracing output:

   ```
   # cat trace
   # tracer: timerlat
   ```

Verification

- Enter the following command to check if timerlat is enabled as the current tracer:

   ```
   # cat /sys/kernel/tracing/current_tracer
   timerlat
   ```

11.2. THE TIMERLAT TRACER OPTIONS
The timerlat tracer is built on top of osnoise tracer. Therefore, you can set the options in the /osnoise/config directory to trace and capture information for thread scheduling latencies.

timerlat options

cpus
Sets CPUs for a timerlat thread to execute on.

timerlat_period_us
Sets the duration period of the timerlat thread in microseconds.

stop_tracing_us
Stops the system tracing if a timer latency at the irq context is more than the configured value. Writing 0 disables this option.

stop_tracing_total_us
Stops the system tracing if the total noise is more than the configured value. Writing 0 disables this option.

print_stack
Saves the stack of the interrupt requests (IRQs) occurrence. The stack saves the IRQs occurrence after the thread context event, or if the IRQs handler is more than the configured value.

11.3. MEASURING TIMER LATENCY WITH RTLA-TIMERLAT-TOP

The rtla-timerlat-top tracer displays a summary of the periodic output from the timerlat tracer. The tracer output also provides information about each operating system noise and events, such as osnoise, and tracepoints. You can view this information by using the -t option.

Procedure

- To measure timer latency:

```
# rtla timerlat top -s 30 -T 30 -t
```

11.4. THE RTLA TIMERLAT TOP TRACER OPTIONS

By using the rtla timerlat top --help command, you can view the help usage on options for the rtla-timerlat-top tracer.

timerlat-top-tracer options

-p, --period us
Sets the timerlat tracer period in microseconds.

-i, --irq us
Stops the trace if the interrupt requests (IRQs) latency is more than the argument in microseconds.

-T, --thread us
Stops the trace if the thread latency is more than the argument in microseconds.

-t, --trace
Saves the stopped trace to the timerlat_trace.txt file.

-s, --stack us
Saves the stack trace at the interrupt requests (IRQs), if a thread latency is more than the argument.
An ultra-low latency is an environment that is optimized to process high volumes of data packets with low tolerance for delay. Providing exclusive resources to applications, including the CPU, is a prevalent practice in ultra-low-latency environments. For example, for high performance network processing in network functions virtualization (NFV) applications, a single application has the CPU power limit set to run tasks continuously.

The Linux kernel includes the real-time analysis (rtla) tool, which provides an interface for the operating system noise (osnoise) tracer. The operating system noise is the interference that occurs in an application as a result of activities inside the operating system. Linux systems can experience noise due to:

- Non maskable interrupts (NMI)
- Interrupt requests (IRQ)
- Soft interrupt requests (SoftIRQ)
- Other system threads activity
- Hardware-related jobs, such as non maskable high priority system management interrupts (SMI)

### 12.1. THE RTLA-OSNOISE TRACER

The Linux kernel includes the real-time analysis (rtla) tool, which provides an interface for the operating system noise (osnoise) tracer. The rtla-osnoise tracer creates a thread that runs periodically for a specified given period. At the start of a period, the thread disables interrupts, starts sampling, and captures the time in a loop.

The rtla-osnoise tracer provides the following capabilities:

- Measure how much operating noise a CPU receives.
- Characterize the type of operating system noise occurring in the CPU.
- Print optimized trace reports that help to define the root cause of unexpected results.
- Saves an interference counter for each interference source. The interference counter for non maskable interrupts (NMI), interrupt requests (IRQ), software interrupt requests (SoftIRQ), and threads increase when the tool detects the entry events for these interferences.

The rtla-osnoise tracer prints a run report with the following information about the noise sources at the conclusion of the period:

- Total amount of noise.
- The maximum amount of noise.
- The percentage of CPU that is allocated to the thread.
- The counters for the noise sources.
12.2. CONFIGURING THE RTLA-OSNOISE TRACER TO MEASURE SCHEDULING LATENCY

You can configure the `rtla-osnoise` tracer by adding `osnoise` in the `current_tracer` file of the tracing system. The `current_tracer` file is generally mounted in the `/sys/kernel/tracing/` directory. The `rtla-osnoise` tracer measures the interrupt requests (IRQs) and saves the trace output for analysis when a thread latency is more than 20 microseconds for a single noise occurrence.

Procedure

1. List the current tracer:

```bash
# cat /sys/kernel/tracing/current_tracer
nop
```

The no operations (nop) is the default tracer.

2. Add the `timerlat` tracer in the `current_tracer` file of the tracing system:

```bash
# cd /sys/kernel/tracing/
# echo osnoise > current_tracer
```

3. Generate the tracing output:

```bash
# cat trace
# tracer: osnoise
```

12.3. THE RTLA-OSNOISE OPTIONS FOR CONFIGURATION

The configuration options for the `rtla-osnoise` tracer is available in the `/sys/kernel/tracing/` directory.

Configuration options for `rtla-osnoise`

- `osnoise/cpus`
  Configures the CPUs for the `osnoise` thread to run on.

- `osnoise/period_us`
  Configures the period for a `osnoise` thread run.

- `osnoise/runtime_us`
  Configures the run duration for a `osnoise` thread.

- `osnoise/stop_tracing_us`
  Stops the system tracing if a single noise is more than the configured value. Setting 0 disables this option.

- `osnoise/stop_tracing_total_us`
  Stops the system tracing if the total noise is more than the configured value. Setting 0 disables this option.

- `tracing_thresh`
  Sets the minimum delta between two `time()` call reads to be considered as noise, in microseconds. When set to 0, `tracing_thresh` uses the default value, which is 5 microseconds.
12.4. THE RTLA-OSNOISE TRACEPOINTS

The **rtla-osnoise** includes a set of **tracepoints** to identify the source of the operating system noise (**osnoise**).

Trace points for **rtla-osnoise**

**osnoise:sample_threshold**
- Displays a noise when the noise is more than the configured threshold (**tolerance_ns**).

**osnoise:nmi_noise**
- Displays noise and the noise duration from non maskable interrupts (NMIs).

**osnoise:irq_noise**
- Displays noise and the noise duration from interrupt requests (IRQs).

**osnoise:softirq_noise**
- Displays noise and the noise duration from soft interrupt requests (SoftIRQs).

**osnoise:thread_noise**
- Displays noise and the noise duration from a thread.

12.5. THE RTLA-OSNOISE TRACER OPTIONS

The **osnoise/options** file includes a set of **on** and **off** configuration options for the **rtla-osnoise** tracer.

Options for **rtla-osnoise**

**DEFAULTS**
- Resets the options to the default value.

**OSNOISE_WORKLOAD**
- Stops the **osnoise** workload dispatch.

**PANIC_ON_STOP**
- Sets the **panic()** call if the tracer stops. This option captures a **vmcore** dump file.

**OSNOISE_PREEMPT_DISABLE**
- Disables preemption for **osnoise** workloads, which allows only interrupt requests (IRQs) and hardware-related noise.

**OSNOISE_IRQ_DISABLE**
- Disables interrupt requests (IRQs) for **osnoise** workloads, which allows only non maskable interrupts (NMIs) and hardware-related noise.

12.6. MEASURING OPERATING SYSTEM NOISE WITH THE RTLA-OSNOISE-TOP TRACER

The **rtla osnoise-top** tracer measures and prints a periodic summary from the **osnoise** tracer along with the information about the occurrence counters of the interference source.

Procedure

1. Measure the system noise:
The command output displays a periodic summary with information about the real-time priority, the assigned CPUs to run the thread, and the period of the run in microseconds.

12.7. THE RTLA-OSNOISE-TOP TRACER OPTIONS

By using the `rtla osnoise top --help` command, you can view the help usage on the available options for the `rtla-osnoise-top` tracer.

Options for `rtla-osnoise-top`

- **-a, --auto us**
  Sets the automatic trace mode. This mode sets some commonly used options while debugging the system. It is equivalent to use `-s us -T 1` and `-t`.

- **-p, --period us**
  Sets the `osnoise` tracer duration period in microseconds.

- **-r, --runtime us**
  Sets the `osnoise` tracer runtime in microseconds.

- **-s, --stop us**
  Stops the trace if a single sample is more than the argument in microseconds. With `-t`, the command saves the trace to the output.

- **-S, --stop-total us**
  Stops the trace if the total sample is more than the argument in microseconds. With `-T`, the command saves a trace to the output.

- **-T, --threshold us**
  Specifies the minimum delta between two time reads to be considered noise. The default threshold is 5 us.

- **-q, --quiet**
  Prints only a summary at the end of a run.

- **-c, --cpus cpu-list**
  Sets the `osnoise` tracer to run the sample threads on the assigned `cpu-list`.

- **-d, --duration time[s|m|h|d]**
  Sets the duration of a run.

- **-D, --debug**
  Prints debug information.

- **-t, --trace[=file]**
  Saves the stopped trace to `[file|osnoise_trace.txt]` file.

- **-e, --event sys:event**
  Enables an event in the trace (`-t`) session. The argument can be a specific event, for example `-e sched:sched_switch`, or all events of a system group, such as `-e sched system group`.

- **--filter <filter>**
  Filters the previous `-e sys:event` system event with a filter expression.

- **--trigger <trigger>**
  Enables a trace event trigger to the previous `-e sys:event` system event.
-P, --priority o:prio|r:prio|f:prio|d:runtime:period
   Sets the scheduling parameters to the osnoise tracer threads.

-h, --help
   Prints the help menu.
CHAPTER 13. MINIMIZING OR AVOIDING SYSTEM SLOWDOWNS DUE TO JOURNALING

The order in which journal changes are written to disk might 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 the **ext2** file system. 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**, records the time a file was last accessed (the **atime** attribute). If you need to use a journaling file system, consider disabling **atime**.

### 13.1. 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.

   ```bash
   /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.

   ```bash
   /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.

### 13.2. ADDITIONAL RESOURCES

- **mkfs.ext2**(8) man page
- **mkfs.xfs**(8) man page
- **mount**(8) man page
CHAPTER 14. 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.

By disabling the graphics console output from logging on and by controlling the messages that print on the graphics console, you can improve latency on sensitive workloads.

14.1. 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 about 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"

14.2. 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 15. 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.

15.1. 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.

15.2. 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.

  ```bash
  # 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.

15.3. 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.

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

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

15.4. 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 the 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**

- You have root permissions on the system.

**Procedure**

1. View the available clock sources.

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

   As an example, consider 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.

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

**NOTE**

The changes apply to 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.

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

  The example uses HPET as the current clock source in the system.
15.5. 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

- You have 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.
   ```bash
   # cd clock_test
   ```

2. View the available clock sources in your system.
   ```bash
   # 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.
   ```bash
   # 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.
   ```bash
   # time ./clock_timing
   real 0m0.601s
   user 0m0.592s
   sys 0m0.002s
   ```
   The example shows the following parameters:
   - **real** - The total time spent beginning from program invocation until the process ends. real includes user and kernel times, and will usually be larger than the sum of the latter two. If this process is interrupted by an application with higher priority, or by a system event such as a hardware interrupt (IRQ), this time spent waiting is also computed under real.
   - **user** - The time the process spent in user space performing tasks that did not require kernel intervention.
• sys – The time spent by the kernel while performing tasks required by the user process. These tasks include opening files, reading and writing to files or I/O ports, memory allocation, thread creation, and network related activities.

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

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

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

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

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

Additional resources

• time(1) man page

15.6. 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

• You have root permissions on the system.

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

15.7. 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:

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

   for(i=0; i<1000000; 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 16. CONTROLLING POWER MANAGEMENT TRANSITIONS

You can control power management transitions to improve latency.

Prerequisites

- You have root permissions on the system.

16.1. 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.

16.2. CONFIGURING POWER MANAGEMENT STATES

You can control power management transitions by configuring power management states with one of the following ways:

- Write a value to the `/dev/cpu_dma_latency` file to change the maximum response time for processes in microseconds and hold the file descriptor open until low latency is required.

- Reference the `/dev/cpu_dma_latency` file in an application or a script.

Prerequisites

- You have administrator privileges.

Procedure

- Specify latency tolerance by writing a 32-bit number that represents a maximum response time in microseconds in `/dev/cpu_dma_latency` and keep the file descriptor open through the low-latency operation. A value of 0 disables C-state completely.

For example:

```python
import os
import os.path
import signal
import sys
if not os.path.exists('/dev/cpu_dma_latency'):
    print("no PM QOS interface on this system!")
    sys.exit(1)
```
fd = os.open('/dev/cpu_dma_latency', os.O_WRONLY)
os.write(fd, b'0\0\0\0')
print("Press ^C to close /dev/cpu_dma_latency and exit")
signal.pause()
except KeyboardInterrupt:
    print("closing /dev/cpu_dma_latency")
os.close(fd)
sys.exit(0)

NOTE

The Power Management Quality of Service interface (pm_qos) interface is only active while it has an open file descriptor. Therefore, any script or program you use to access /dev/cpu_dma_latency must hold the file open until power-state transitions are allowed.
CHAPTER 17. 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.

17.1. 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>
<thead>
<tr>
<th>CPUs</th>
<th>Bitmask</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000000000000000000000000001</td>
<td>1</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0, 1</td>
<td>00000000000000000000000000000011</td>
<td>3</td>
<td>0x00000011</td>
</tr>
</tbody>
</table>

17.2. 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`.

   ```bash
   # 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.

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

**Verification**

- Check that the `irqbalance` status is inactive.

   ```bash
   # systemctl status irqbalance
   ```

**17.3. 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`.

   ```bash
   # IRQBALANCE_BANNED_CPUS
   # 64 bit bitmask which allows you to indicate which cpu's should
   # be skipped when relancing 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 17.2. Examples

<table>
<thead>
<tr>
<th>CPUs</th>
<th>Bitmask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000001</td>
</tr>
<tr>
<td>8 - 15</td>
<td>0000ff00</td>
</tr>
<tr>
<td>8 - 15, 33</td>
<td>00000001,0000ff00</td>
</tr>
</tbody>
</table>

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`.

17.4. 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.

### 17.5. 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 run 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 want to change. In this example, the process with a PID of 7013 is being instructed to run only on CPU 0.
    ```
    # taskset -p 1 7013
NOTE
You can combine the listed options.

Additional resources
- `taskset(1)` man page
- `numactl(8)` man page
CHAPTER 18. 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.

18.1. PREREQUISITES

- You have root permissions on the system.

18.2. 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`.

   ```bash
   # 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`.

   ```bash
   # 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`.

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

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

18.3. 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.
   ```bash
   # cat /proc/12465/oom_score
   79872
   ```

2. Display the contents of `oom_adj` for the process.
   ```bash
   # cat /proc/12465/oom_adj
   13
   ```

3. Edit the value in `oom_adj`.
   ```bash
   # echo -5 > /proc/12465/oom_adj
   ```

**Verification steps**

1. Display the current `oom_score` for the process.
   ```bash
   # cat /proc/12465/oom_score
   78
   ```

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

**18.4. 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 19. IMPROVING LATENCY USING THE TUNA CLI

You can use the **tuna** CLI to improve latency on your system. The tuna CLI, for RHEL 9, includes the command line interface, which is based on the **argparse** parsing module. The interface provides the following capabilities:

- A more standardized menu of commands and options
- With the interface, you can use predefined inputs and **tuna** ensures that the inputs are of the right type
- Generates usage help messages automatically, on how to use parameters and provides error messages with invalid arguments

19.1. PREREQUISITES

- The **tuna** and the **python-linux-procfs** packages are installed.
- You have root permissions on the system.

19.2. THE TUNA CLI

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

The **tuna** 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 now has a set of commands, which formerly were the action options. These commands are:

- **isolate**
  Move all threads and IRQs away from the **CPU-LIST**.
- **include**
  Configure all threads to run on a **CPU-LIST**.
- **move**
  Move specific entities to the **CPU-LIST**.
- **spread**
  Spread the selected entities over the **CPU-LIST**.
- **priority**
  Set the thread scheduler tunables, such as **POLICY** and **RTPRIO**.
- **run**
  Fork a new process and run the command.
- **save**
  Save **kthreads sched tunables** to **FILENAME**.
- **apply**
  Apply changes defined in the profile.
- **show_threads**
  Display a thread list.
Display the IRQ list.

display

Display the existing profile list.

show_configs

Provide help about selected entities.

what_is

Start the graphical user interface (GUI).

gui

You can view the commands with the tuna -h command. For each command, there are optional arguments, which you can view with the tuna <command> -h command. For example, with the tuna isolate -h command, you can view the options for isolate.

19.3. 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 tuna and the python-linux-procsfs packages are installed.
- You have root permissions on the system.

Procedure

- Isolate one or more CPUs.

  # tuna isolate --cpus=<cpu_list>

  cpu_list is a comma-separated list or a range of CPUs to isolate.

  For example:

  # tuna isolate --cpus=0,1

  or

  # tuna isolate --cpus=0-5

19.4. 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 tuna and python-linux-procsfs packages are installed.
- You have root permissions on the system.
Procedure

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

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

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

For example:
```
# tuna show_irqs --irqs=128
```

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

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

*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 or a range of CPUs.

For example:
```
# tuna move --irqs=128 --cpus=3
```

Verification

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

```
# tuna show_irqs --irqs=128
```

19.5. 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 *tuna* and *python-linux-procsfs* packages are installed.
- You have root permissions on the system.

**NOTE**

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

Procedure

1. View the information for a thread.

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

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

```
# tuna show_threads --threads=42369,42416,43859
```

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

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

- `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 priority FIFO:1 --threads=42369,42416,43859
```

**Verification**

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

```
# tuna show_threads --threads=42369,42416,43859
```
CHAPTER 20. 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.

20.1. 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
  ...
  ```

20.2. 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
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

- You have administrator privileges.
- A service that runs on boot.

Procedure

For an existing service:

1. Create a supplementary service configuration directory file for the service.
   
   ```sh
   # cat <<-EOF > /etc/systemd/system/mcelog.system.d/priority.conf
   
   [SERVICE]
   CPUSchedulingPolicy=fifo
   CPUSchedulingPriority=20
   EOF
   ```

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

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

3. Reload the systemd scripts configuration.

   ```sh
   # systemctl daemon-reload
   ```

4. Restart the service.

   ```sh
   # systemctl restart mcelog
   ```

Verification

- Display the service’s priority.

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

The output shows the configured priority of the service.

For example:

```sh
thread  ctxt_switches
pid SCHED_ rtpri affinity voluntary nonvoluntary cmd
826  FIFO 20 0,1,2,3 13 0 mcelog
```
20.3. CONFIGURING THE CPU USAGE OF A SERVICE

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

Prerequisites

- You have 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
   ```

20.4. PRIORITY MAP

Scheduler priorities are defined in groups, with some groups dedicated to particular kernel functions.
### Table 20.1. Thread priority table

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

### 20.5. ADDITIONAL RESOURCES

- Working with systemd unit files
CHAPTER 21. 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.

21.1. 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**

- You have 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
```
21.2. 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

- You have administrator privileges.

Procedure

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

  ```bash
  # ethtool -A enp0s31f6
  ```

Verification

Verify that the pause parameter changed.

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

21.3. MONITORING NETWORK PROTOCOL STATISTICS

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

Procedure

To monitor network traffic:

  ```bash
  $ 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

21.4. ADDITIONAL RESOURCES

- **ethtool(8)** man page
- **netstat(8)** man page
CHAPTER 22. TRACING LATENCIES WITH TRACE-CMD

The `trace-cmd` utility is a front end to the `ftrace` utility. By using `trace-cmd`, you can enable `ftrace` actions, without the need to write to the `/sys/kernel/debug/tracing/` directory. `trace-cmd` does not add any overhead on its installation.

Prerequisites

- You have administrator privileges.

22.1. INSTALLING TRACE-CMD

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

Prerequisites

- You have administrator privileges.

Procedure

- Install the `trace-cmd` utility.

```
# dnf install trace-cmd
```

22.2. RUNNING TRACE-CMD

You can use the `trace-cmd` utility to access all `ftrace` functionalities.

Prerequisites

- You have 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`.

22.3. 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
  ```

### 22.4. ADDITIONAL RESOURCES

- *trace-cmd*(1) man page
CHAPTER 23. 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 `tuned-profiles-realtime` package, you can automate operations to isolate a CPU.

Prerequisites

- You have administrator privileges.

23.1. CHOOSING CPUs TO ISOLATE

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

- If you have a multi-threaded application where threads need to communicate with one another by sharing cache, they 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 can 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 are installed.

Procedure

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

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

### 23.2. 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.
In tuned-profiles-realtime version 2.19 and later, 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 permissions on the system.

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 real-time 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:
23.3. 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 24. LIMITING SCHED_OTHER TASK MIGRATION

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

Prerequisites

- You have administrator privileges.

24.1. 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.

24.2. 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 25. 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**

- You have administrator privileges.

25.1. TURNING OFF TCP TIMESTAMPS

Turning off TCP timestamps can reduce TCP performance spikes.

**Procedure**

- Turn off TCP timestamps:

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

  The output shows that the value of `net.ipv4.tcp_timestamps` options is 0. That is, TCP timestamps are disabled.

25.2. 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.

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

  The output shows that the value of `net.ipv4.tcp_timestamps` is 1. That is, TCP timestamps are enabled.

25.3. DISPLAYING THE TCP TIMESTAMP STATUS

You can view the status of TCP timestamp generation.

**Procedure**

- Display the TCP timestamp generation status:

  ```bash
  # 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 26. 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

- You have administrator privileges.
- The tuna package is installed

26.1. 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 thread to a housekeeping CPU to relieve CPU 3 from being assigned RCU callback jobs.

26.2. 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.

### 26.3. 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.

### 26.4. ADDITIONAL RESOURCES

- [Avoiding RCU Stalls in the real-time kernel](#)
CHAPTER 27. 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 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

- You have administrator privileges.

27.1. 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_preemptirqoff 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.


   ```
   # 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 \texttt{wakeup\_rt}, \texttt{preemptirqsoff}, and so on automatically enables \textbf{function tracing}, which may exaggerate the overhead. Check if \texttt{function} and \texttt{function\_graph} tracing are enabled:

\begin{verbatim}
# cat /sys/kernel/debug/tracing/options/function-trace
1
\end{verbatim}

- A value of \texttt{1} indicates that \texttt{function} and \texttt{function\_graph} tracing are enabled.
- A value of \texttt{0} indicates that \texttt{function} and \texttt{function\_graph} tracing are disabled.

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

\begin{verbatim}
# echo 0 > /sys/kernel/debug/tracing/options/function-trace
# echo 1 > /sys/kernel/debug/tracing/options/function-trace
\end{verbatim}

\textbf{IMPORTANT}

When using the \texttt{echo} command, ensure you place a space character in between the value and the \texttt{>} character. At the shell prompt, using \texttt{0>}, \texttt{1>}, and \texttt{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 \texttt{/debugfs/tracing/} directory.

For example:

The \texttt{irqsoff}, \texttt{preemptoff}, \texttt{preemptirqsoff}, and \texttt{wakeup} tracers continuously monitor latencies. When they record a latency greater than the one recorded in \texttt{tracing\_max\_latency} the trace of that latency is recorded, and \texttt{tracing\_max\_latency} is updated to the new maximum time. In this way, \texttt{tracing\_max\_latency} always shows the highest recorded latency since it was last reset.

- To reset the maximum latency, echo \texttt{0} into the \texttt{tracing\_max\_latency} file:

\begin{verbatim}
# echo 0 > /sys/kernel/debug/tracing/tracing_max_latency
\end{verbatim}

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

\begin{verbatim}
# echo 200 > /sys/kernel/debug/tracing/tracing_max_latency
\end{verbatim}

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:

\begin{verbatim}
# echo 200 > /sys/kernel/debug/tracing/tracing\_thresh
\end{verbatim}

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.

27.2. 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.

27.3. 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.

### 27.4. FTRACE EXAMPLES

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

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

- Trace only the `schedule` function:
  
  ```
  # echo schedule > /sys/kernel/debug/tracing/set_ftrace_filter
  ```

- Trace all functions that end with `lock`:
  
  ```
  # echo "*lock" > /sys/kernel/debug/tracing/set_ftrace_filter
  ```

- Trace all functions that start with `spin_`:
  
  ```
  # echo "spin_*" > /sys/kernel/debug/tracing/set_ftrace_filter
  ```

- Trace all functions with `cpu` in the name:
  
  ```
  # echo "cpu" > /sys/kernel/debug/tracing/set_ftrace_filter
  ```
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:

```c
#include <time.h>

main()
```
You can improve upon the example above by adding checks to verify the return code of \texttt{clock_gettime()}, to verify the value of the \texttt{rc} variable, or to ensure the content of the \texttt{ts} structure is to be trusted.

\textbf{NOTE}

The \texttt{clock_gettime()} man page provides more information about writing more reliable applications.

\textbf{IMPORTANT}

Programs using the \texttt{clock_gettime()} function must be linked with the \texttt{rt} library by adding \texttt{-lrt} to the \texttt{gcc} command line.

\texttt{$ gcc clock_timing.c -o clock_timing -lrt$}

\subsection*{28.3. ADDITIONAL RESOURCES}

- \texttt{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
- You have 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.

```c
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.

   ```c
   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.

   ```c
   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

• tcp(7) man page

• setsockopt(3p) man page

• 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 benefits 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. This is a basic safety procedure that you must always perform.

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.

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.

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

Real-time priority inheritance

You can avoid priority inversion problems by using priority inheritance.

```c
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.

```c
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.

```c
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.

  ```c
  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

- futex(7) man page
- pthread_mutex_destroy(P) man page
- pthread_mutexattr_setprotocol(3p) man page
- pthread_mutexattr_setprioceiling(3p) man page
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.
- You have 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]
```
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`
- `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**

- You have root permissions on 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
  ```
The special mode with 0 stressors, query the available 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:

  ```bash
  # 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

- You have root privileges on the systems

Procedure

- Run multiple instances of CPU stressors as follows:

  ```bash
  # 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:

  ```bash
  # 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.

  ```bash
  # 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:

  ```bash
  # 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**

- You have root privileges on the system.

**Procedure**

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

   ```bash
   # 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:

   ```bash
   # 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 available bogo operations and the matrix stressor performance on your machine.

**Prerequisites**

- You have root privileges on the system.

**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 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 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**

- You have root privileges on the system.

**Procedure**

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

```bash
# 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**

- You have root permissions on the system.

**Procedure**

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

```bash
# 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**
- You have root permissions 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**
- You have root permissions on the system.

**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

- You have root permissions 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 about 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.
- 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

- You have administrator privileges.

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!

4. Create the following Dockerfile:

   # vim Dockerfile

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:

```bash
# 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

<table>
<thead>
<tr>
<th>T</th>
<th>P</th>
<th>I</th>
<th>C</th>
<th>Min</th>
<th>Act</th>
<th>Avg</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>95</td>
<td>1000</td>
<td>3209</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>95</td>
<td>1500</td>
<td>2137</td>
<td>1</td>
<td>2</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2000</td>
<td>1601</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2500</td>
<td>1280</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>3000</td>
<td>1066</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>3500</td>
<td>913</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>4000</td>
<td>798</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>4500</td>
<td>709</td>
<td>1</td>
<td>2</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>

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 cyclic test 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

- You have 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

- `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.

```sh
# 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.

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

2. Add the following lines into the file.

   ```c
   #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

- 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:

```c
#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", 
```
```c
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

- sched_get_priority_min(2) man page
- 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

- `nice(2)` man page
- `getpriority(2)` man page
- `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.
   
   ```bash
   $ {EDITOR} sched_timeslice.c
   ```

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

   ```c
   #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=1000000

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

- You have 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
```
You can set the priority for a process using the `chrt` utility.

**Prerequisites**
- You have 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 about 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

- `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

- You have administrator privileges.

37.1. LIBRARY CALLS FOR SETTING PRIORITY

Real-time processes use a different set of library calls to control policy and 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.

**IMPORTANT**

The functions require the inclusion of the `sched.h` header file. Ensure you always check the return codes from functions.

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:

```c
struct sched_param sp = { .sched_priority = 50 };  
int ret;  
ret = sched_setscheduler(0, SCHED_FIFO, &sp);  
if (ret == -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:

```c
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
CHAPTER 38. SCHEDULING PROBLEMS ON THE REAL-TIME KERNEL AND SOLUTIONS

Scheduling in the real-time kernel might have consequences sometimes. By using the information provided, you can understand the problems on scheduling policies, scheduler throttling, and thread starvation states on the real-time kernel, as well as potential solutions.

38.1. SCHEDULING POLICIES FOR THE REAL-TIME KERNEL

The real-time scheduling policies share one main characteristic: they run until a higher priority thread interrupts the thread or the threads wait, either by sleeping or performing I/O.

In the case of SCHED_RR, the operating system interrupts a running thread so that another thread of equal SCHED_RR priority can 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 example, 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 and can potentially crash.

On the RHEL for Real Time kernel, interrupt handlers run as threads with a SCHED_FIFO priority. The 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 the programs waiting for data signaled by those interrupts to starve and fail.

38.2. SCHEDULER THROTTLING IN THE REAL-TIME KERNEL

The real-time kernel includes a safeguard mechanism to enable allocating bandwidth for use by the real-time tasks. The safeguard mechanism is known as real-time scheduler throttling.

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 the 95% CPU time slot, the remaining real-time tasks on that CPU will not run. Only the non real-time tasks use the remaining 5% of CPU time. The default values can have the following performance impacts:

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

The real-time scheduler throttling is controlled by the following parameters in the /proc file system:

The /proc/sys/kernel/sched_rt_period_us parameter

Defines the period in µs (microseconds), which is 100% of the CPU bandwidth. The default value is 1,000,000 µs, which is 1 second. Changes to the period’s value must be carefully considered because a period value that is either very high or low can cause problems.

The /proc/sys/kernel/sched_rt_runtime_us parameter

Defines the total bandwidth available for all real-time tasks. The default value is 950,000 µs (0.95 s), which is 95% of the CPU bandwidth. Setting the value to -1 configures the real-time tasks to 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.

### 38.3. THREAD STARVATION IN THE REAL-TIME KERNEL

Thread starvation occurs when a thread is on a CPU run queue for longer than the starvation threshold and does not make progress. A common cause of thread starvation is to run a fixed-priority polling application, such as `SCHED_FIFO` or `SCHED_RR` bound to a CPU. Since the polling application does not block for I/O, this can prevent other threads, such as `kworkers`, from running on that CPU.

An early attempt to reduce thread starvation is called as real-time throttling. In real-time throttling, each CPU has a portion of the execution time dedicated to non real-time tasks. The default setting for throttling is on with 95% of the CPU for real-time tasks and 5% reserved for non real-time tasks. This works if you have a single real-time task causing starvation but does not work if there are multiple real-time tasks assigned to a CPU. You can work around the problem by using:

**The stalld mechanism**

The `stalld` mechanism is an alternative for real-time throttling and avoids some of the throttling drawbacks. `stalld` is a daemon to periodically monitor the state of each thread in the system and looks for threads that are on the run queue for a specified length of time without being run. `stalld` temporarily changes that thread to use the `SCHED_DEADLINE` policy and allocates the thread a small slice of time on the specified CPU. The thread then runs, and when the time slice is used, the thread returns to its original scheduling policy and `stalld` continues to monitor thread states.

Housekeeping CPUs are CPUs that run all daemons, shell processes, kernel threads, interrupt handlers, and all work that can be dispatched from an isolated CPU. For housekeeping CPUs with real-time throttling disabled, `stalld` monitors the CPU that runs the main workload and assigns the CPU with the `SCHED_FIFO` busy loop, which helps to detect stalled threads and improve the thread priority as required with a previously defined acceptable added noise. `stalld` can be a preference if the real-time throttling mechanism causes an unreasonable noise in the main workload.

With `stalld`, you can more precisely control the noise introduced by boosting starved threads. The shell script `/usr/bin/throttlectl` automatically disables real-time throttling when `stalld` is run. You can list the current throttling values by using the `/usr/bin/throttlectl show` script.

**Disabling real-time throttling**

The following parameters in the `/proc` filesystem control real-time throttling:

- The `/proc/sys/kernel/sched_rt_period_us` parameter specifies the number of microseconds in a period and defaults to 1 million, which is 1 second.

- The `/proc/sys/kernel/sched_rt_runtime_us` parameter specifies the number of microseconds that can be used by a real-time task before throttling occurs and it defaults to 950,000 or 95% of the available CPU cycles. You can disable throttling by passing a value of `-1` into the `sched_rt_runtime_us` file by using the `echo -1 > /proc/sys/kernel/sched_rt_runtime_us` command.