Red Hat Enterprise Linux for Real Time 9

Understanding RHEL for Real Time

An introduction to RHEL for Real Time kernel

Last Updated: 2024-03-26

Red Hat Enterprise Linux for Real Time 9 Understanding RHEL for Real Time

An introduction to RHEL for Real Time kernel

Legal Notice

Abstract

Understand the fundamental concepts and associated references on tuning the RHEL for Real Time kernel to maintain low latency and a consistent response time on latency sensitive applications.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

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

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

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

Submitting 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. HARDWARE PLATFORMS FOR RHEL FOR REAL TIME

Configuring the hardware correctly plays a critical role in setting up the real-time environment because hardware impacts the way your system operates. Not all hardware platforms are real-time capable and enable fine tuning. Before performing fine tuning, you must ensure that the potential hardware platform is real-time capable.

Hardware platforms vary based on the vendor. You can test and verify the hardware suitability for real-time with the hardware latency detector (**hwlatdetect**) program. The program controls the latency detector kernel module and helps to detect latencies caused by underlying hardware or firmware behavior.

Any tuning steps required for low latency operation have been completed. Refer to the vendor documentation for instructions to

Prerequisites

- The RHEL-RT packages are installed.
- Any tuning steps required for low latency operation are complete. Refer to the vendor documentation for instructions to reduce or remove any System Management Interrupts (SMIs) that make the system move to System Management Mode (SMM).

> **WARNING**
>
> You must avoid disabling System Management Interrupts (SMIs) completely because it can result in catastrophic hardware failures.

1.1. PROCESSOR CORES

A real-time processor core is a physical Central Processing Unit (CPU) and it executes the machine code. A socket is a connection between the processor and the motherboard of the computer. The socket is the location on the motherboard that the processor is placed into. There are two sets of processors:

- Single core processor that occupies one socket with one available core.
- Quad-core processor that occupies one socket with four available cores.

When designing a real time environment, be aware of the number of available cores, the cache layout among cores, and how the cores are physically connected.

When multiple cores are available, use threads or processes. A program when written without using these constructs, runs on a single processor at a time. A multi-core platform provides advantages through using different cores for different types of operations.

Caches

Caches have a noticeable impact on overall processing time and determinism. Often, the threads of an application need to synchronize access to a shared resource, such as a data structure.

With the **tuna** command line tool (CLI), you can determine the cache layout and bind interacting threads to a core so that they share the cache. Cache sharing reduces memory faults by ensuring that the mutual exclusion primitive (mutex, condition variables, or similar) and the data structure use the same cache.

Interconnects

Increasing the number of cores on systems can cause conflicting demands on the interconnects. This makes it necessary to determine the interconnect topology to help detect the conflicts that occur between the cores on real-time systems.

Many hardware vendors now provide a transparent network of interconnects between cores and memory, known as Non-uniform memory access (NUMA) architecture.

NUMA is a system memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. When you use NUMA, a processor can access its own local memory faster than non-local memory, such as memory on another processor or memory shared between processors. On NUMA systems, understanding the interconnect topology helps to place threads that communicate frequently on adjacent cores.

The **taskset** and **numactl** utilities determine the CPU topology. **taskset** defines the CPU affinity without NUMA resources such as memory nodes and **numactl** controls the NUMA policy for processes and shared memory.

1.2. ADDITIONAL RESOURCES

- Installing RHEL 9 for Real Time

CHAPTER 2. MEMORY MANAGEMENT ON RHEL FOR REAL TIME

Real-time systems use a virtual memory system, where an address referenced by a user-space application translates into a physical address. The translation occurs through a combination of page tables and address translation hardware in the underlying computing system. An advantage of having the translation mechanism in between a program and the actual memory is that the operating system can swap pages when required or on request of the CPU.

In real-time, to swap pages from the storage to the primary memory, the previously used page table entry is marked as invalid. As a result, even under normal memory pressure, the operating system can retrieve pages from one application and give them to another. This can cause unpredictable system behaviors.

Memory allocation implementations include demand paging mechanism and memory lock (**mlock()**) system calls.

NOTE

Sharing data information about CPUs in different cache and NUMA domains might cause traffic problems and bottlenecks.

When writing a multithreaded application, consider the machine topology before designing the data decomposition. Topology is the memory hierarchy, and includes CPU caches and the Non-Uniform Memory Access (NUMA) node.

2.1. DEMAND PAGING

Demand paging is similar to a paging system with page swapping. The system loads pages that are stored in the secondary memory when required or on CPU demand. All memory addresses generated by a program pass through an address translation mechanism in the processor. The addresses then convert from a process-specific virtual address to a physical memory address. This is referred to as virtual memory. The two main components in the translation mechanism are page tables and translation lookaside buffers (TLBs)

Page tables

Page tables are multi-level tables in physical memory that contain mappings for virtual to physical memory. These mappings are readable by the virtual memory translation hardware in the processor.

A page table entry with an assigned physical address, is referred to as the resident working set. When the operating system needs to free memory for other processes, it can swap pages from the resident working set. When swapping pages, any reference to a virtual address within that page creates a page fault and causes page reallocation.

When the system is extremely low on physical memory, the swap process starts to thrash, which constantly steals pages from processes, and never allows a process to complete. You can monitor the virtual memory statistics by looking for the **pgfault** value in the **/proc/vmstat** file.

Translation lookaside buffers

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

A TLB operates on the principle of locality of reference. This means that if code stays in one region of memory for a significant period of time (such as loops or call-related functions) then the TLB references avoid the main memory for address translations. This can significantly speed up processing times.

When writing deterministic and fast code, use functions that maintain locality of reference. This can mean using loops rather than recursion. If recursion are not avoidable, place the recursion call at the end of the function. This is called tail-recursion, which makes the code work in a relatively small region of memory and avoids calling table translations from the main memory.

2.2. MAJOR AND MINOR PAGE FAULTS

RHEL for Real Time allocates memory by breaking physical memory into chunks called pages and then maps them to the virtual memory. Faults in real-time occur when a process needs a specific page that is not mapped or is no longer available in the memory. Hence faults essentially mean unavailability of pages when required by a CPU. When a process encounters a page fault, all threads freeze until the kernel handles this fault. There are several ways to address this problem, but the best solution can be to adjust the source code to avoid page faults.

Minor page faults

Minor page faults in real-time occur when a process attempts to access a portion of memory before it has been initialized. In such scenarios, the system performs operations to fill the memory maps or other management structures. The severity of a minor page fault can depend on the system load and other factors, but they are usually short and have a negligible impact.

Major page faults

Real-time major faults occur when the system has to either synchronize memory buffers with the disk, swap memory pages belonging to other processes, or undertake any other input output (I/O) activity to free memory. This occurs when the processor references a virtual memory address that does not have a physical page allocated to it. The reference to an empty page causes the processor to execute a fault, and instructs the kernel code to allocate a page which increases latency dramatically.

In real-time, when an application shows a performance drop, it is beneficial to check for the process information related to page faults in the **/proc/** directory. For a specific process identifier (PID), using the **cat** command, you can view the **/proc/PID/stat** file for following relevant entries:

- Field 2: the executable file name.
- Field 10: the number of minor page faults.
- Field 12: the number of major page faults.

The following example demonstrates viewing page faults with the **cat** command and a **pipe** function to return only the second, tenth, and twelfth lines of the **/proc/PID/stat** file:

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

In the example output, the process with PID 3366 is bash and it has 5389 minor page faults and no major page faults.

Additional resources

- Linux System Programming by Robert Love

2.3. MLOCK() SYSTEM CALLS

The memory lock (**mlock()**) system calls enable calling processes to lock or unlock a specified range of the address space and prevents Linux from paging the locked memory to swap space. After you allocate the physical page to the page table entry, references to that page are relatively fast. The memory lock system calls fall into **mlock()** and **munlock()** categories.

The **mlock()** and **munlock()** system calls lock and unlock a specified range of process address pages. When successful, the pages in the specified range remain resident in the memory until the **munlock()** system call unlocks the pages.

The **mlock()** and **munlock()** system calls take following parameters:

- **addr**: specifies the start of an address range.
- **len**: specifies the length of the address space in bytes.

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

The **mlockall()** and **munlockall()** system calls locks or unlocks all of the program space.

NOTE

The **mlock()** system call does not ensure that the program will not have a page I/O. It ensures that the data stays in the memory but cannot ensure it stays on the same page. Other functions such as **move_pages** and memory compactors can move data around regardless of the use of **mlock()**.

Memory locks are made on a page basis and do not stack. If two dynamically allocated memory segments share the same page locked twice by **mlock()** or **mlockall()**, they unlock by using a single **munlock()** or **munlockall()** system call. As such, it is important to be aware of the pages that the application unlocks to avoid double-locking or single-unlocking problems.

The following are two most common workarounds to mitigate double-lock or single-unlock problems:

- Tracking the allocated and locked memory areas and creating a wrapper function that verifies the number of page allocations before unlocking a page. This is the resource counting principle used in device drivers.
- Making memory allocations based on page size and alignment to avoid double-lock on a page.

Additional resources

- **capabilities(7)** man page
- **mlock(2)** man page
- **mlock(3)** man page
- **mlockall(2)** man page
- **mmap(2)** man page
- **move_pages(2)** man page

- **posix_memalign(3)** man page
- **posix_memalign(3p)** man page

2.4. SHARED LIBRARIES

The RHEL for Real Time shared libraries are called dynamic shared objects (DSO) and are a collection of pre-compiled code blocks called functions. These functions are reusable in multiple programs and they load at run-time or compile time.

Linux supports the following two library classes:

- Dynamic or shared libraries: exists as separate files outside of the executable file. These files load into the memory and get mapped at run-time.
- Static libraries: are files linked to a program statically at compile time.

The **ld.so** dynamic linker loads the shared libraries required by a program and then executes the code. The DSO functions load the libraries in the memory once and multiple processes can then reference the objects by mapping into the address space of processes. You can configure the dynamic libraries to load at compile time using the **LD_BIND_NOW** variable.

Evaluating symbols before program initialization can improve performance because evaluating at application run-time can cause latency if the memory pages are located on an external disk.

Additional resources

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

2.5. SHARED MEMORY

In RHEL for Real Time, shared memory is a memory space shared between multiple processes. Using program threads, all threads created in one process context can share the same address space. This makes all data structures accessible to threads. With POSIX shared memory calls, you can configure processes to share a part of the address space.

You can use the following supported POSIX shared memory calls:

- **shm_open()**: creates and opens a new or opens an existing POSIX shared memory object.
- **shm_unlink()**: unlinks POSIX shared memory objects.
- **mmap()**: creates a new mapping in the virtual address space of the calling process.

NOTE

The mechanism for sharing a memory region between two processes using System V IPC **shmem()** set of calls is deprecated and is no longer supported on RHEL for Real Time.

Additional resources

- **shm_open(3)** man page
- **shm_overview(7)** man page

- **mmap(2)** man page

CHAPTER 3. 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.

3.1. 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

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

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

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

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

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

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

3.2. 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:

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

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

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

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

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

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

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

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

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

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

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

3.3. 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:

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

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

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

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

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

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

3.4. 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 usr time sys time   bogo ops/s bogo ops/s
stress-ng: info:  [17579]                (secs)   (secs)  (secs)  (real time) (usr+sys time)
stress-ng: info:  [17579] matrix  349322   60.00    203.23   0.19      5822.03      1717.25
```

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

3.5. 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:

  ```
  # 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.

3.6. 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:

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

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

3.7. 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
  ```

3.8. 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
   ```

3.9. 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 4. HARDWARE INTERRUPTS ON RHEL FOR REAL TIME

Real-time systems receive many interrupts over the course of its operation, including a semi-regular "timer" interrupt that periodically performs maintenance and system scheduling decisions. The systems may also receive special kinds of interrupts, such as Nonmaskable Interrupt (NMI) and System Management Interrupt (SMI). Hardware interrupts are used by devices to indicate a change in the physical state of the system that requires attention. For example, a hard disk signaling that it has read a series of data blocks, or when a network device has processed a buffer containing network packets.

When an interrupt occurs in real-time, the system stops active programs are stopped and executes an interrupt handler is executed.

In real-time, hardware interrupts are referenced by an interrupt number. These numbers are mapped back to the piece of hardware that created the interrupt. This enables the system to monitor which device created the interrupt and when it occurred. When an interrupt occurs in real-time, the system stops active programs and executes an interrupt handler. The handler preempts other running programs and system activities. This can slow the entire system and create latencies.

RHEL for Real Time modifies the way interrupts are handled to improve performance and decrease latency. Using the **cat /proc/interrupts** command you can print an output to view the types of hardware interrupts that took place, the number of interrupts received, the target CPU for the interrupt, and the device generating the interrupt.

4.1. LEVEL-SIGNALED INTERRUPTS

In real-time, level-signaled interrupts use a dedicated interrupt line that delivers voltage transitions. The device controller raises an interrupt by asserting a signal on the interrupt request line. The interrupt line sends one of two voltages to represent a binary 1 or binary 0.

When the interrupt signal is sent by the line, it remains in that state until the CPU resets it. The CPU performs a state save, captures the interrupt, and dispatches the interrupt handler. The interrupt handler determines the cause of the interrupt, clears the interrupt by performing necessary services, and restores the state of the device. The Level-signaled interrupts are more reliable and supports multiple devices, though they are complex to implement.

4.2. MESSAGE-SIGNALED INTERRUPTS

In real-time, many systems use message-signaled interrupts (MSI), which send the signal as a dedicated message on a packet or message-based electrical bus. A common example of this type of bus is the Peripheral Component Interconnect Express (PCI Express or PCIe). These devices transmit a message type, which the PCIe host controller interprets as an interrupt message. The host controller then sends the message on to the CPU.

In real-time, depending on the hardware, a PCIe system does one of the following:

- Sends the signal using a dedicated interrupt line between the PCIe host controller and the CPU.
- Sends the message over the CPU HyperTransport bus.

In real-time, PCIe systems can also operate in legacy mode, where legacy interrupt lines are implemented in order to support older operating systems or on boot Linux kernels with the option **pci=nomsi** on the kernel command line.

4.3. NON-MASKABLE INTERRUPTS

In real-time, non-maskable interrupts are hardware interrupts that standard interrupt masking techniques in the system cannot ignore. The NMIs have higher priority than the maskable interrupts. The NMIs occur to signal attention for non recoverable hardware errors.

In real-time, NMIs are also used by some systems as a hardware monitor. When a processor receives NMI, it handles the NMI immediately by calling the NMI handler pointed to by the interrupt vector. If certain conditions are met, such as an interrupt not being triggered after a specified length of time, the NMI handler signals a warning and provides debugging information about the problem. This helps to identify and prevent system lockups.

In real-time, maskable interrupts are hardware interrupts that can be ignored by setting a bit in an interrupt mask register's bit-mask. CPUs can temporarily ignore maskable interrupts during critical processing.

4.4. SYSTEM MANAGEMENT INTERRUPTS

In real-time, system management interrupts (SMIs) offer extended functionality, such as legacy hardware device emulation and can also be used for system management tasks. SMIs are similar to non maskable interrupts (NMIs) in that they use a special electrical signalling line and are generally not maskable. When an SMI occurs, the CPU enters the System Management Mode (SMM). In this mode, a special low-level handler executes to handle the SMIs. The SMM is typically provided directly from the system management firmware, often the BIOS or the EFI.

The real-time SMIs are most often used to provide legacy hardware emulation. A common example is to imitate a diskette drive. When there is no diskette drive attached, the operating system attempts to access the diskette and triggers a SMI. In this scenario, a handler provides the operating system with an emulated device instead. The operating system then treats the emulation as a legacy device.

In real-time, SMIs can adversely affect the system because they take place without the direct involvement of the operating system. A poorly written SMI handling routine may consume many milliseconds of CPU time, and the operating system might not be able to preempt the handler. This can create periodic high latencies in an otherwise well-tuned and highly responsive system. As a vendor may use SMI handlers to manage CPU temperature and fan control, it may not be possible to disable them. In such situations, you must notify the vendor of problems that occur when using these interrupts.

In real-time, you can isolate SMIs using the **hwlatdetect** utility. It is available in the **rt-tests** package. This utility measures the time period during which the CPU is used by an SMI handling routine.

4.5. ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER

The advanced programmable interrupt controller (APIC) developed by Intel Corporation, provides the ability to:

- Handle large amounts of interrupts to route each to a specific set of CPUs.
- Support inter-CPU communication and remove the need for multiple devices to share a single interrupt line.

The real-time APIC represents a series of devices and technologies that work together to generate, route, and handle a large number of hardware interrupts in a scalable and manageable way. It uses a combination of a local APIC built into each system CPU and a number of Input/Output APICs that are connected directly to hardware devices.

In real-time, when a hardware device generates an interrupt, the connected I/O APIC detects and routes the interrupt across the system APIC bus to a specific CPU. The operating system knows the IO-APIC connectes to the devices, and interrupt line within that device. The Advanced Configuration and Power Interface Differentiated System Description Table (ACPI DSDT) includes information about the specific wiring of the host system motherboard and peripheral components and a device provides information about the available interrupt sources. Together, these two sets of data provide information about the overall interrupt hierarchy.

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

CHAPTER 5. RHEL FOR REAL TIME PROCESSES AND THREADS

The RHEL for Real Time key factors in operating systems are minimal interrupt latency and minimal thread switching latency. Although all programs use threads and processes, RHEL for Real Time handles them in a different way compared to the standard Red Hat Enterprise Linux.

In real-time, using parallelism helps achieve greater efficiency in task execution and latency. Parallelism is when multiple tasks or several sub-tasks run at the same time using the multi-core infrastructure of CPU.

5.1. PROCESSES

A real-time process, in simplest terms, is a program in execution. The term process refers to an independent address space, potentially containing multiple threads. When the concept of more than one process running inside one address space was developed, Linux turned to a process structure that shares an address space with another process. This works well, as long as the process data structure is small.

A UNIX®-style process construct contains:

- Address mappings for virtual memory.
- An execution context (PC, stack, registers).
- State and accounting information.

In real-time, each process starts with a single thread, often called the parent thread. You can create additional threads from parent threads using the **fork()** system calls. **fork()** creates a new child process which is identical to the parent process except for the new process identifier. The child process runs independent of the creating process. The parent and child processes can be executed simultaneously. The difference between the **fork()** and **exec()** system calls is that, **fork()** starts a new process which is the copy of the parent process and **exec()** replaces the current process image with the new one.

In real-time, the **fork()** system call, when successful, returns the process identifier of the child process and the parent process returns a non-zero value. On error, it returns an error number.

5.2. THREADS

In real-time, multiple threads can exist within a process. All threads of a process share its virtual address space and system resources. A thread is a schedulable entity that contains:

- A program counter (PC).
- A register context.
- A stack pointer.

In real-time, following are potential mechanisms to create parallelism:

- Using the **fork()** and **exec()** function calls to create new processes. The **fork()** call creates an exact duplicate of a process from which it is called and has a unique process identifier.
- Using the Posix threads (**pthreads**) API to create new threads within an already running process.

You must evaluate the component interaction level before forking real-time threads. Creating a new address space and running it as a new process is beneficial when the components are independent of one another or with less interaction. When components are required to share data or communicate frequently, running the threads within one address space is more efficient.

In real-time, the **fork()** system call, when successful, returns a zero value. On error, it returns an error number.

5.3. ADDITIONAL RESOURCES

- **fork(2)** man page
- **exec(2)** man page

CHAPTER 6. APPLICATION TIMESTAMPING ON RHEL FOR REAL TIME

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.

6.1. 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.

6.2. 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.

6.3. CLOCK_GETTIME() FUNCTION

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

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

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

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

NOTE

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

IMPORTANT

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

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

6.4. ADDITIONAL RESOURCES

- **clock_gettime()** man page

CHAPTER 7. 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.

7.1. 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⇐deadline⇐period** to be true. The relation between the required options is **runtime⇐deadline⇐period**.

7.2. 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 7.1. SCHED_DEADLINE parameters

Parameter	Description
period	**period** is the activation pattern of a real-time task. For example, if a video processing task has 60 frames per second to process, a new frame is queued for service every 16 milliseconds. Therefore, the **period** is 16 milliseconds.
runtime	**runtime** is the amount of CPU execution time allotted to the task to produce an output. In real-time, the maximum execution time, also known as "Worst Case Execution Time" (WCET) is the **runtime**. For example, if a video processing tool can take, in the worst case, five milliseconds to process an image, the **runtime** is five milliseconds.
deadline	**deadline** is the maximum time for the output to be produced. For example, if a task needs to deliver the processed frame within ten milliseconds, the **deadline** is ten milliseconds.

CHAPTER 8. 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.

8.1. 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 shut-down 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.

8.2. 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.

8.3. 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.

Settings	Description	Example commands
available_monitors	Displays the available monitors one per line.	**# cat available_monitors**

Settings	Description	Example commands
available_reactors	Display the available reactors one per line.	**# cat available_reactors**
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 9. AFFINITY IN RHEL FOR REAL TIME

In real-time, every thread and interrupt source in the system has a processor affinity property. The operating system scheduler uses this information to determine which threads and interrupts to run on which CPU.

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

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

Some of the more typical affinity setups include:

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

NOTE

The affinity settings must be designed in conjunction with the program for good expected behavior.

9.1. PROCESSOR AFFINITY

In real-time, the processes by default, can run on any CPU. However, you can configure the processes to run on a predetermined selection of CPUs, by changing the affinity of the process. Child processes inherit the CPU affinities of their parents.

The real-time practice for tuning affinities on a system is to determine the number of cores required to run the application and then isolating those cores. This can be achieved with the Tuna tool, or with shell scripts to modify the bitmask value.

The **taskset** command can be used to change the affinity of a process and modifying the **/proc/** filesystem entry changes the affinity of an interrupt. Using the **taskset** command with the **-p** or **--pid** option and the process identifier (PID) of the process, checks the affinity of a process.

The **-c** or **--cpu-list** option displays the numerical list of cores, instead of as a bitmask. The affinity can be set by specifying the number of the CPU to bind a specific process. For example, for a process that previously used either CPU 0 or CPU 1, you can change the affinity so that it can only run on CPU 1. In addition to the **taskset** command, you can also set the processor affinity can using the **sched_setaffinity()** system call.

Additional resources

- **taskset(1)** man page
- **sched_setaffinity(2)** man page

9.2. SCHED_DEADLINE AND CPUSETS

The kernel’s deadline scheduling class (**SCHED_DEADLINE**) implements early deadline first scheduler (EDF) for sporadic tasks with a constrained deadline. It prioritizes the tasks according to the job deadline: earliest absolute deadline first. In addition to the EDF scheduler, the deadline scheduler also implements the constant bandwidth server (CBS). The CBS algorithm is a resource reservation protocol.

The CBS guarantees that each task receives its run time **(Q)** at every period **(T)**. At the start of every activation of a task, the CBS replenishes the task’s run time. As the job runs, it consumes its **runtime** and if the task runs out of its **runtime**, the task is throttled and de-scheduled. The throttling mechanism prevents a single task from running more than its runtime and helps to avoid the performance problems of other jobs.

In real-time, to avoid the overloading the system with **deadline** tasks, the **deadline** scheduler implements an acceptance test, which is run every time a task is configured to run with the **deadline scheduler**. The acceptance test guarantees that **SCHED_DEADLINE** tasks does not use more CPU time than the specified on the **kernel.sched_rt_runtime_us**/**kernel.sched_rt_period_us** files, which is 950 ms over 1s, by default.

CHAPTER 10. THREAD SYNCHRONIZATION MECHANISMS IN RHEL FOR REAL TIME

In real-time, when two or more threads need access to a shared resource at the same time, the threads coordinate using the thread synchronization mechanism. Thread synchronization ensures that only one thread uses the shared resource at a time. The three thread synchronization mechanisms used on Linux: Mutexes, Barriers, and Condition variables (**condvars**).

10.1. MUTEXES

Mutex derives from the terms mutual exclusion. The mutual exclusion object synchronizes access to a resource. It is a mechanism that ensures only one thread can acquire a mutex at a time.

The **mutex** algorithm creates a serial access to each section of code, so that only one thread executes the code at any one time. Mutexes are created using an attribute object known as the **mutex** attribute object. It is an abstract object, which contains several attributes that depends on the POSIX options you choose to implement. The attribute object is defined with the **pthread_mutex_t** variable. The object stores the attributes defined for the mutex. The **pthread_mutex_init(&my_mutex, &my_mutex_attr)**, **pthread_mutexattr_setrobust()** and **pthread_mutexattr_getrobust()** functions return 0, when successful. On error, they return the error number.

In real-time, you can either retain the attribute object to initialize more mutexes of the same type or you can clean up (destroy) the attribute object. The mutex is not affected in either case. Mutexes include the standard and advanced type of mutexes.

Standard mutexes

The real-time standard mutexes are private, non-recursive, non-robust, and non-priority inheritance capable mutexes. Initializing a **pthread_mutex_t** using **pthread_mutex_init(&my_mutex, &my_mutex_attr)** creates a standard mutex. When using the standard mutex type, your application may not benefit from the advantages provided by the **pthreads** API and the RHEL for Real Time kernel.

Advanced mutexes

Mutexes defined with additional capabilities are called advanced mutexes. Advanced capabilities include priority inheritance, robust behavior of a mutex, and shared and private mutexes. For example, for robust mutexes, initializing the **pthread_mutexattr_setrobust()** function, sets the robust attribute. Similarly, using the attribute **PTHREAD_PROCESS_SHARED**, allows any thread to operate on the mutex, provided the thread has access to its allocated memory. The attribute **PTHREAD_PROCESS_PRIVATE** sets a private mutex.

A non-robust mutex does not release automatically and stays locked until you manually release it.

Additional resources

- **futex(7)** man page
- **pthread_mutex_destroy(P)** man page

10.2. BARRIERS

Barriers operate in a very different way when compared to other thread synchronization methods. The barriers define a point in the code where all active threads stop until all threads and processes reach this barrier. Barriers are used in situations when a running application needs to ensure that all threads have completed specific tasks before execution can continue.

The barrier mutex in real-time, take following two variables:

- The first variable records the **stop** and **pass** state of the barrier.
- The second variable records the total number of threads that enter the barrier.

The barrier sets the state to **pass** only when the specified number of threads reach the defined barrier. When the barrier state is set to **pass**, the threads and processes proceed further. The **pthread_barrier_init()** function allocates the required resources to use the defined barrier and initializes it with the attributes referenced by the **attr** attribute object.

The **pthread_barrier_init()** and **pthread_barrier_destroy()** functions return the zero value, when successful. On error, they return an error number.

10.3. CONDITION VARIABLES

In real-time, condition variables (**condvar**) is a POSIX thread construct that waits for a particular condition to be achieved before proceeding. In general, the signaled condition relates to the state of data that the thread shares with another thread. For example, a **condvar** can be used to signal a data entry into a processing queue and a thread waiting to process that data from the queue. Using the **pthread_cond_init()** function, you can initialize a condition variable.

The **pthread_cond_init()**, **pthread_cond_wait()**, and **pthread_cond_signal()** functions return the zero value, when successful. On error, it returns the error number.

10.4. MUTEX CLASSES

The mentioned mutex options provides guidance on the mutex classes to consider when writing or porting an application.

Table 10.1. Mutex options

Advanced mutexes	Description
Shared mutexes	Defines shared access for multiple threads to acquire a mutex at a given time. Shared mutexes can create latency. The attribute is **PTHREAD_PROCESS_SHARED**.
Private mutexes	Ensures that only the threads created within the same process can access the mutex. The attribute is **PTHREAD_PROCESS_PRIVATE**.
Real-time priority inheritance	Sets the priority level of the lower priority task higher above a current higher priority task. When the task completes, it releases the resource and the task drops back to its original priority permitting the higher priority task to run. The attribute is **PTHREAD_PRIO_INHERIT**.

Advanced mutexes	Description
Robust mutexes	Sets the robust mutexes to release automatically when the owning thread would stop. The value substring **NP** in the string **PTHREAD_MUTEX_ROBUST_NP**, indicates that robust mutexes are non-POSIX or not portable

Additional resources

- **futex(7)** man page

10.5. THREAD SYNCHRONIZATION FUNCTIONS

The mentioned list of function types and the description provides information on the functions to use for thread synchronization mechanisms for the real-time kernel.

Table 10.2. Functions

Function	Description
pthread_mutexattr_init(&my_mutex_attr)	Initiates a mutex with attributes specified by **attr**. If **attr** is NULL, it applies the default mutex attributes.
pthread_mutexattr_destroy(&my_mutex_attr)	Destroys the specified mutex object. You can re-initialize with **pthread_mutex_init()**.
pthread_mutexattr_setrobust()	Specifies the **PTHREAD_MUTEX_ROBUST** attribute of a mutex. The **PTHREAD_MUTEX_ROBUST** attribute defines a thread that can stop without unlocking the mutex. A future call to own this mutex succeeds automatically and returns the value **EOWNERDEAD** to indicate that the previous mutex owner no longer exists.
pthread_mutexattr_getrobust()	Queries the **PTHREAD_MUTEX_ROBUST** attribute of a mutex.
pthread_barrier_init()	Allocates the required resources to use and initialize the barrier with attribute object **attr**. If **attr** is NULL, it applies the default values.
pthread_cond_init()	Initializes a condition variable. The **cond** argument defines the object to initiate with the attributes in the condition variable attribute object **attr**. If **attr** is NULL, it applies the default values.
pthread_cond_wait()	Blocks a thread execution until it receives a signal from another thread. In addition, a call to this function also releases the associated lock on mutex before blocking. The argument **cond** defines the **pthread_cond_t** object for a thread to block on. The**mutex** argument specifies the mutex to unblock.

Function	Description
pthread_cond_signal()	Unblocks at least one of the threads that are blocked on a specified condition variable. The argument **cond** specifies using the **pthread_cond_t** object to unblock the thread.

CHAPTER 11. SOCKET OPTIONS IN RHEL FOR REAL TIME

The real-time socket is a two way data transfer mechanism between two processes on same systems such as the UNIX domain and loopback devices or on different systems such as network sockets.

Transmission Control Protocol (TCP) is the most common transport protocol and is often used to achieve consistent low latency for a service that requires constant communication or to cork the sockets in a low priority restricted environment.

With new applications, hardware features, and kernel architecture optimizations, TCP has to introduce new approaches to handle the changes effectively. The new approaches can cause unstable program behaviors. Because the program behavior changes as the underlying operating system components change, they must be handled with care.

One example of such behavior in TCP is the delay in sending small buffers. This allows sending them as one network packet. Buffering small writes to TCP and sending them all at once generally works well, but it can also create latencies. For real-time applications, the **TCP_NODELAY** socket option disables the delay and sends small writes as soon as they are ready.

The relevant socket options for data transfer are **TCP_NODELAY** and **TCP_CORK**.

11.1. TCP_NODELAY SOCKET OPTION

The **TCP_NODELAY** socket option disables Nagle's algorithm. Configuring **TCP_NODELAY** with the **setsockopt** sockets API function sends multiple small buffer writes as individual packets as soon as they are ready.

Sending multiple logically related buffers as a single packet by building a contiguous packet before sending, achieves better latency and performance. Alternatively, if the memory buffers are logically related but not contiguous, you can create an I/O vector and pass it to the kernel using **writev** on a socket with **TCP_NODELAY** enabled.

The following example illustrates enabling **TCP_NODELAY** through the **setsockopt** sockets API.

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

NOTE

To use **TCP_NODELAY** effectively, avoid small, logically related buffer writes. With **TCP_NODELAY**, small writes make TCP send multiple buffers as individual packets, which may result in poor overall performance.

Additional resources

- **sendfile(2)** man page

11.2. TCP_CORK SOCKET OPTION

The **TCP_CORK** option collects all data packets in a socket and prevents from transmitting them until the buffer fills to a specified limit. This enables applications to build a packet in the kernel space and send data when **TCP_CORK** is disabled. **TCP_CORK** is set on a socket file descriptor using the **setsocketopt()** function. When developing programs, if you must send bulk data from a file, consider using **TCP_CORK** with the **sendfile()** function.

When a logical packet is built in the kernel by various components, enable **TCP_CORK** by configuring it to a value of 1 using the **setsockopt** sockets API. This is known as "corking the socket”. **TCP_CORK** can cause bugs if the cork is not removed at an appropriate time.

The following example illustrates enabling **TCP_CORK** through the **setsockopt** sockets API.

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

In some environments, if the kernel is not able to identify when to remove the cork, you can manually remove it as follows:

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

Additional resources

- **sendfile(2)** man page

11.3. EXAMPLE PROGRAMS USING SOCKET OPTIONS

The **TCP_NODELAY** and **TCP_CORK** socket options significantly influence the behavior of a network connection. **TCP_NODELAY** disables the Nagle’s algorithm on applications that benefit by sending data packets as soon as they are ready. With **TCP_CORK**, you can transfer multiple data packets simultaneously, with no delays between them.

NOTE

To enable the socket options, for example **TCP_NODELAY**, build it with the following code and then set appropriate options.

```
gcc tcp_nodelay_client.c -o tcp_nodelay_client -lrt
```

When you run the **tcp_nodelay_server** and **tcp_nodelay_client** programs without any arguments, the client uses the default socket options. For more information about **tcp_nodelay_server** and **tcp_nodelay_client** programs, see the TCP changes result in latency performance when small buffers are used article.

The example programs provide information about the performance impact these socket options can have on your applications.

Performance impact on a client

You can send small buffer writes to a client without using the **TCP_NODELAY** and **TCP_CORK** socket options. When run without any arguments, the client uses the default socket options.

- To initiate data transfer, define the server TCP port and the number of packets it must process. For example, 10,000 packets in this test.

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

  The code sends 15 packets, each of two bytes, and waits for a response from the server. It adopts the default TCP behavior here

Performance impact on a loopback interface

To enable the socket option, build it using **gcc tcp_nodelay_client.c -o tcp_nodelay_client -lrt** and then set the appropriate options.

Following examples use a loopback interface to demonstrate three variations:

- To send buffer writes immediately, set the **no_delay** option on a socket configured with **TCP_NODELAY**.

  ```
  $ ./tcp_nodelay_client localhost 5001 10000 no_delay

   10000 packets of 30 bytes sent in 1649.771240 ms: 181.843399 bytes/ms using
  TCP_NODELAY
  ```

  TCP sends the buffers right away, disabling the algorithm that combines the small packets. This improves performance but can cause a flurry of small packets to be sent for each logical packet.

- To collect multiple data packets and send them with one system call, configure the **TCP_CORK** socket option.

  ```
  $ ./tcp_nodelay_client localhost 5001 10000 cork

   10000 packets of 30 bytes sent in 850.796448 ms: 352.610779 bytes/ms using TCP_CORK
  ```

  Using the cork technique significantly reduces the time required to send data packets as it combines full logical packets in its buffers and sends fewer overall network packets. You must ensure to remove the **cork** at the appropriate time.

  When developing programs, if you must send bulk data from a file, consider using **TCP_CORK** with the **sendfile()** option.

- To measure performance without using socket options.

  ```
  $ ./tcp_nodelay_client localhost 5001 10000

   10000 packets of 30 bytes sent in 400129.781250 ms: 0.749757 bytes/ms
  ```

  This is the baseline measure when TCP combines buffer writes and waits to check for more data than can optimally fit in the network packet.

Additional resources

- **sendfile(2)** man page

CHAPTER 12. RHEL FOR REAL TIME SCHEDULER

RHEL for Real Time uses the command line utilities help you to configure and monitor process configurations.

12.1. CHRT UTILITY FOR SETTING THE SCHEDULER

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

The **chrt** utility takes the either **--pid** or the **-p** option to specify the process ID (PID).

The **chrt** utility takes the following policy options:

- **-f** or **--fifo**: sets the schedule to **SCHED_FIFO**.
- **-o** or **--other**: sets the schedule to **SCHED_OTHER**.
- **-r** or **--rr**: sets schedule to **SCHED_RR**.
- **-d** or **--deadline**: sets schedule to **SCHED_DEADLINE**.

The following example shows the attributes for a specified process.

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

12.2. PREEMPTIVE SCHEDULING

The real-time preemption is the mechanism to temporarily interrupt an executing task, with the intention of resuming it at a later time. It occurs when a higher priority process interrupts the CPU usage. Preemption can have a particularly negative impact on performance, and constant preemption can lead to a state known as thrashing. This problem occurs when processes are constantly preempted and no process ever gets to run completely. Changing the priority of a task can help reduce involuntary preemption.

You can check for voluntary and involuntary preemption occurring on a single process by viewing the contents of the **/proc/PID/status** file, where PID is the process identifier.

The following example shows the preemption status of a process with PID 1000.

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

12.3. LIBRARY FUNCTIONS FOR SCHEDULER PRIORITY

The real-time processes use a different set of library calls to control policy and priority. The functions require the inclusion of the **sched.h** header file. The symbols **SCHED_OTHER**, **SCHED_RR** and **SCHED_FIFO** must also be defined in the **sched.h** header file.

The table lists the functions that set the policy and priority for the real-time scheduler.

Table 12.1. Library functions for real-time scheduler

Functions	Description
sched_getscheduler()	Retrieves the scheduler policy for a specific process identifier (PID)
sched_setscheduler()	Sets the scheduler policy and other parameters. This function requires three parameters: **sched_setscheduler(pid_t pid**, **int policy**, **const struct sched_param *sp);**
sched_getparam()	Retrieves the scheduling parameters of a scheduling policy.
sched_setparam()	Sets the parameters associated with a scheduling policy that has been already set and can be verified using the **sched_getparam()** function.
sched_get_priority_max()	Returns the maximum valid priority associated with the scheduling policy.
sched_get_priority_min()	Returns the minimum valid priority associated with the scheduling policy .
sched_rr_get_interval()	Displays the allocated **timeslice** for each process.

CHAPTER 13. SYSTEM CALLS IN RHEL FOR REAL TIME

The real-time system call is a function used by application programs to communicate with the kernel. It is a mechanism for programs to order resources from the kernel.

13.1. SCHED_YIELD() FUNCTION

The **sched_yield()** function is designed for a processor to select a process other than the running one. This type of request is prone to failure when issued from within a poorly-written application.

When the **sched_yield()** function is used within processes with real-time priorities, it can display unexpected behavior. The process that calls **sched_yield()** moves to the tail of the queue of processes running at same priority. When there are no other processes running at the same priority, the process that called **sched_yield()** continues to run. If the priority of that process is high, it can potentially create a busy loop, rendering the machine unusable.

In general, do not use **sched_yield()** on real-time processes.

13.2. GETRUSAGE() FUNCTION

The **getrusage()** function retrieves important information from a specified process or its threads. It reports on information such as:

- The number of voluntary and involuntary context switches.
- Major and minor page faults.
- Amount of memory in use.

getrusage() enables you to query an application to provide information relevant to both performance tuning and debugging activities. **getrusage()** retrieves information that would otherwise need to be cataloged from several different files in the /**proc**/ directory and would be hard to synchronize with specific actions or events on the application.

NOTE

Not all the fields contained in the structure filled with **getrusage()** results are set by the kernel. Some of them are kept for compatibility reasons only.

Additional resources

- **getrusage(2)** man page

CHAPTER 14. MEASURING SCHEDULING LATENCY USING TIMERLAT IN RHEL FOR REAL TIME

The **rtla-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.

14.1. CONFIGURING THE TIMERLAT TRACER TO MEASURE SCHEDULING LATENCY

You can configure the **timerlat** tracer by adding **timerlat** in the **curret_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
```

14.2. 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.

14.3. 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
  ```

14.4. 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.

CHAPTER 15. MEASURING SCHEDULING LATENCY USING RTLA-OSNOISE IN RHEL FOR REAL TIME

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 (NMIs)
- Interrupt requests (IRQs)
- Soft interrupt requests (SoftIRQs)
- Other system threads activity
- Hardware-related jobs, such as non maskable high priority system management interrupts (SMIs)

15.1. 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 (NMIs), interrupt requests (IRQs), software interrupt requests (SoftIRQs), 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.

15.2. CONFIGURING THE RTLA-OSNOISE TRACER TO MEASURE SCHEDULING LATENCY

You can configure the **rtla-osnoise** tracer by adding **osnoise** in the **curret_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:

```
# 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 osnoise > current_tracer
```

3. Generate the tracing output:

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

15.3. 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.

15.4. 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.

15.5. 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.

15.6. 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:

```
# rtla osnoise top -P F:1 -c 0-3 -r 900000 -d 1M -q
```

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.

15.7. 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 16. 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.

16.1. 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.

16.2. 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.

16.3. 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.