Red Hat Enterprise Linux for Real Time 7

Tuning Guide

Advanced tuning procedures to optimize latency in RHEL for Real Time

Last Updated: 2019-09-05

Red Hat Enterprise Linux for Real Time 7 Tuning Guide

Advanced tuning procedures to optimize latency in RHEL for Real Time

Jaroslav Klech
Red Hat Customer Content Services
jklech@redhat.com

Marie Doleželová
Red Hat Customer Content Services

Jana Heves
Red Hat Customer Content Services

Maxim Svistunov
Red Hat Customer Content Services

Radek Bíba
Red Hat Customer Content Services

David Ryan
Red Hat Customer Content Services

Cheryn Tan
Red Hat Customer Content Services

Lana Brindley
Red Hat Customer Content Services

Alison Young
Red Hat Customer Content Services

Legal Notice

Abstract

This book contains advanced tuning procedures for Red Hat Enterprise Linux for Real Time. For installation instructions, see the Red Hat Enterprise Linux for Real Time Installation Guide .

Table of Contents

PREFACE

This book details tuning information about Red Hat Enterprise Linux for Real Time.

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

For most applications running under a Linux environment, basic performance tuning can improve latency sufficiently. For those industries where latency not only needs to be low, but also accountable and predictable, Red Hat has now developed a 'drop-in' kernel replacement that provides this. Red Hat Enterprise Linux for Real Time provides seamless integration with Red Hat Enterprise Linux 7 and offers clients the opportunity to measure, configure, and record latency times within their organization.

You will need to have the Red Hat Enterprise Linux for Real Time kernel installed before you begin the tuning procedures in this book. If you have not yet installed the Red Hat Enterprise Linux for Real Time kernel, or need help with installation issues, read the Red Hat Enterprise Linux for Real Time Installation Guide.

CHAPTER 1. BEFORE YOU START TUNING YOUR RED HAT ENTERPRISE LINUX FOR REAL TIME SYSTEM

Red Hat Enterprise Linux for Real Time is designed to be used on well-tuned systems for applications with extremely high determinism requirements. Kernel system tuning offers the vast majority of the improvement in determinism. For example, in many workloads thorough system tuning improves consistency of results by around 90%. This is why we typically recommend that customers first perform the Chapter 2, *General System Tuning* of standard Red Hat Enterprise Linux before using Red Hat Enterprise Linux for Real Time.

Things to Remember While You Are Tuning Your Red Hat Enterprise Linux for Real Time Kernel

1. Be Patient

   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 tunings that work best for your system.

   Additionally, always make long test runs. Changing some tuning parameters then doing a five minute test run is not a good validation of a set of tunes. Make the length of your test runs adjustable and run them for longer than a few minutes. Try to narrow down to a few different tuning sets with test runs of a few hours, then run those sets for many hours or days at a time, to try and catch corner-cases of max latencies or resource exhaustion.

2. Be Accurate

   Build a measurement mechanism into your application, so that you can accurately gauge how a particular set of tuning changes affect the application's performance. Anecdotal evidence (for example, "The mouse moves more smoothly") is usually wrong and varies from person to person. Do hard measurements and record them for later analysis.

3. Be Methodical

   It is very tempting to make multiple changes to tuning variables between test runs, but doing so means that you do not have a way to narrow down which tune affected your test results. Keep the tuning changes between test runs as small as you can.

4. Be Conservative

   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.

5. Be Smart

   Use the tools you have available. The Tuna graphical 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** tools can help locate latency issues.

6. Be Flexible

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

Scheduling Policies

Linux uses three main 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.

1.1. RUNNING LATENCY TESTS AND INTERPRETING THEIR RESULTS

To verify that the potential hardware platform is suitable for real-time operations, you should run some latency and performance tests with the Real Time kernel. These tests can highlight BIOS or system tuning (including partitioning) issues that might be experienced under a load.

1.1.1. Preliminary Steps

Procedure 1.1. To successfully test your system and interpret the results:

1. Check the vendor documentation for any tuning steps required for low latency operation.

   This step aims 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 is running firmware and not running operating system code, meaning any timers that expire while in SMM will have to wait until the system transitions back into normal operation. This can cause unexplained latencies since SMIs cannot be blocked by Linux and the only indication that we actually took an SMI may 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.

2. Ensure that RHEL-RT and **rt-tests** package is installed.

This step verifies that you have tuned the system properly.

3. Run the **hwlatdetect** program.

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

   Generally, you do not need to run any sort of load on the system while running **hwlatdetect**, since the program is looking for latencies introduced by hardware architecture or BIOS/EFI firmware.

   A typical output of **hwlatdetect** looks like this:

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

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

   The above result represents a system that was tuned to minimize system interruptions from firmware.

   However, not all systems can be tuned to minimize system interruptions as shown below:

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

hwlatdetect was using the **tracer** mechanism as the **detector** for unexplained latencies. Previous versions used a kernel module rather than **ftrace tracer**.

parameters report a latency and how the detection was run. The default latency threshold was 10 microseconds (10 us), the sample window was 1 second, the sampling window was 0.5 seconds.

As a result, **tracer** ran a **detector** thread that ran for one half of each second of the specified duration.

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

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

The inner loop comparison checks that **t0 - t1** does not exceed the specified threshold (10 us default). The outer loop comparison checks the time between the bottom of the loop and the top **t1 - t0**. The time between consecutive reads of the timestamp register should be dozens of nanoseconds (essentially a register read, a comparison and a conditional jump) so any other delay between consecutive reads is introduced by firmware or by the way the system components were connected.

NOTE

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

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

1.1.2. Testing the System Real-time Performance under Load

RHEL-RT provides the **rteval** utility to test the system real-time performance under load. **rteval** starts a heavy system load of **SCHED_OTHER** tasks and 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** then starts the **cyclictest** measurement program. This program starts the **SCHED_FIFO** real-time thread on each online core and then measures 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 for the **rteval** run are written to the **XML** file along with the boot log for the system. Then the **rteval-<date>-N.tar.bz2** file is generated. **N** is a counter for the Nth run on **<date>**. A report, generated from the **XML** file, similar to the below, will be printed to the screen:

```
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 above brings details on the hardware, length of the run, options used, and the timing results, both per-cpu and system-wide. You can regenerate the report by running the **# rteval --summarize rteval-<date>-n.tar.bz2** command.

CHAPTER 2. GENERAL SYSTEM TUNING

This chapter contains general tuning that can be performed on a standard Red Hat Enterprise Linux installation. It is important that these are performed first, in order to better see the benefits of the Red Hat Enterprise Linux for Real Time kernel.

It is recommended that you read these sections first. They contain background information on how to modify tuning parameters and will help you perform the other tasks in this book:

- Section 2.1, “Using the Tuna Interface”
- Section 2.2, “Setting Persistent Tuning Parameters”

When are you ready to begin tuning, perform these steps first, as they will provide the greatest benefit:

- Section 2.3, “Setting BIOS Parameters”
- Section 2.4, “Interrupt and Process Binding”
- Section 2.5, “File System Determinism Tips”

When you are ready to start some fine-tuning on your system, then try the other sections in this chapter:

- Section 2.6, “Using Hardware Clocks for System Timestamping”
- Section 2.7, “Avoid Running Extra Applications”
- Section 2.8, “Swapping and Out of Memory Tips”
- Section 2.9, “Network Determinism Tips”
- Section 2.10, “**syslog** Tuning Tips”
- Section 2.11, “The PC Card Daemon”
- Section 2.12, “Reduce TCP Performance Spikes”
- Section 3.7, “Reducing the TCP Delayed ACK Timeout”

When you have completed all the tuning suggestions in this chapter, move on to Chapter 3, *Realtime-Specific Tuning*

2.1. USING THE TUNA INTERFACE

Throughout this book, instructions are given for tuning the Red Hat Enterprise Linux for Real Time kernel directly. The Tuna interface is a tool that assists you with making changes. It has a graphical interface, or can be run through the command shell.

Tuna can be used to change attributes of threads (scheduling policy, scheduler priority and processor affinity) and interrupts (processor affinity). The tool is designed to be used on a running system, and changes take place immediately. This allows any application-specific measurement tools to see and analyze system performance immediately after the changes have been made.

2.2. SETTING PERSISTENT TUNING PARAMETERS

This book contains many examples on how to specify kernel tuning parameters. Unless stated otherwise, the instructions will cause the parameters to remain in effect until the system reboots or they are explicitly changed. This approach is effective for establishing the initial tuning configuration.

Once you have decided what tuning configuration works for your system, you can make them persistent across reboots. The method you choose depends on the type of parameter you are setting.

Procedure 2.1. Editing the /etc/sysctl.conf File

For any parameter that begins with **/proc/sys/**, including it in the **/etc/sysctl.conf** file will make the parameter persistent.

1. Open the **/etc/sysctl.conf** file in your chosen text editor.

2. Remove the **/proc/sys/** prefix from the command and replace the central / character with a **.** character.

   For example: the command **echo 0 > /proc/sys/kernel/hung_task_panic** will become **kernel.hung_task_panic**.

3. Insert the new entry into the **/etc/sysctl.conf** file with the required parameter.

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

4. Run **# sysctl -p** to refresh with the new configuration.

```
~]# sysctl -p
...[output truncated]...
kernel.hung_task_panic = 0
```

Procedure 2.2. Editing the /etc/rc.d/rc.local File

> **WARNING**
>
> The **/etc/rc.d/rc.local** mechanism should not be used for production startup code. It is a holdover from the SysV Init days of startup scripts and is executed now by the **systemd** service. It should only be used for testing of startup code, since there is no way to control ordering or dependencies.

1. Adjust the command as per the Procedure 2.1, “Editing the **/etc/sysctl.conf** File” instructions.

2. Insert the new entry into the **/etc/rc.d/rc.local** file with the required parameter.

2.3. SETTING BIOS PARAMETERS

Because every system and BIOS vendor uses different terms and navigation methods, this section contains only general information about BIOS settings. If you have trouble locating the setting mentioned, contact the BIOS vendor.

Power Management

Anything that tries to save power by either changing the system clock frequency or by putting the CPU into various sleep states can affect how quickly the system responds to external events.

For best response times, disable power management options in the BIOS.

Error Detection and Correction (EDAC) units

EDAC units are devices used to detect and correct 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 is spent in BIOS, and the more likely that crucial event deadlines will be missed.

Turn EDAC off if possible. Otherwise, switch to the lowest functional level.

System Management Interrupts (SMI)

SMIs are a facility used by hardware vendors ensure the system is operating correctly. The SMI interrupt is usually not serviced by the running operating system, but by code in the BIOS. 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 check to what extent it is safe to disable them.

> **WARNING**
>
> While it is possible to completely disable SMIs, it is strongly recommended that you do not do this. Removing the ability for your system to generate and service SMIs can result in catastrophic hardware failure.

2.4. INTERRUPT AND PROCESS BINDING

Real-time environments need to minimize or eliminate latency when responding to various events. Ideally, interrupts (IRQs) and user processes can be isolated from one another on different dedicated CPUs.

Interrupts are generally shared evenly between CPUs. This can delay interrupt processing through having to write new data and instruction caches, and often creates conflicts with other processing occurring on the CPU. In order to overcome this problem, time-critical interrupts and processes can be dedicated to a CPU (or a range of CPUs). In this way, the code and data structures needed to process this interrupt will have the highest possible likelihood to be in the processor data and instruction caches. The dedicated process can then run as quickly as possible, while all other non-time-critical processes run on the remainder of the CPUs. This can be particularly important in cases where the speeds involved are in the limits of memory and peripheral bus bandwidth available. Here, 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, in tuning applications for different companies which perform similar functions, the optimal performance tunings were completely different. For one firm, isolating 2 out of 4 CPUs for operating system functions and interrupt handling and dedicating the remaining 2 CPUs purely for application handling was optimal. For another firm, binding the network related application processes onto a CPU which was handling the network device driver interrupt yielded optimal determinism. Ultimately, tuning is often accomplished by trying a variety of settings to discover what works best for your organization.

IMPORTANT

For many of the processes described here, you will 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. It can also be expressed as a decimal or hexadecimal number, depending on the command you are using. For example: The CPU mask for CPU 0 only is **00000000000000000000000000000001** as a bitmask, **1** as a decimal, and **0x00000001** as a hexadecimal. The CPU mask for both CPU 0 and 1 is **00000000000000000000000000000011** as a bitmask, **3** as a decimal, and **0x00000003** as a hexadecimal.

Procedure 2.3. Disabling the **irqbalance** Daemon

This daemon is enabled by default and periodically forces interrupts to be handled by CPUs in an even, fair manner. However in real-time deployments, applications are typically dedicated and bound to specific CPUs, so the **irqbalance** daemon is not required.

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

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

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

```
~]# systemctl stop irqbalance
```

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

```
~]# systemctl disable irqbalance
```

Procedure 2.4. Excluding CPUs from IRQ Balancing

The **/etc/sysconfig/irqbalance** configuration file contains a setting that allows CPUs to be excluded from consideration by the IRQ balacing service. This parameter is named **IRQBALANCE_BANNED_CPUS** and is a 64-bit hexadecimal bit mask, where each bit of the mask represents a CPU core.

For example, if you are running a 16-core system and want to remove CPUs 8 to 15 from IRQ balancing, do the following:

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

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

2. Exclude CPUs 8 to 15 by uncommenting the variable **IRQBALANCE_BANNED_CPUS** and setting its value this way:

   ```
   IRQBALANCE_BANNED_CPUS=0000ff00
   ```

3. This will cause the **irqbalance** process to ignore the CPUs that have bits set in the bitmask; in this case, bits 8 through 15.

4. If you are running a system with up to 64 CPU cores, separate each group of eight hexadecimal digits with a comma:

   ```
   IRQBALANCE_BANNED_CPUS=00000001,0000ff00
   ```

   The above mask excludes CPUs 8 to 15 as well as CPU 33 from IRQ balancing.

NOTE

From Red Hat Enterprise Linux 7.2, the **irqbalance** tool automatically avoids IRQs on CPU cores isolated via the **isolcpus=** kernel parameter if **IRQBALANCE_BANNED_CPUS** is not set in the **/etc/sysconfig/irqbalance** file.

Procedure 2.5. Manually Assigning CPU Affinity to Individual IRQs

1. Check which IRQ is in use by each device by viewing the **/proc/interrupts** file:

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

   This file contains a list of IRQs. 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
   ...[output truncated]...
   ```

2. To instruct an IRQ to run on only one processor, use the **echo** command to write the CPU mask, as a hexadecimal number, to the **smp_affinity** entry of the specific IRQ. In this example, we are instructing the interrupt with IRQ number 142 to run on CPU 0 only:

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

3. This change will only take effect once an interrupt has occurred. To test the settings, generate some disk activity, then check the **/proc/interrupts** file for changes. Assuming that you have caused an interrupt to occur, you will see that the number of interrupts on the chosen CPU have risen, while the numbers on the other CPUs have not changed.

Procedure 2.6. Binding Processes to CPUs Using the**taskset** Utility

The **taskset** utility uses the process ID (PID) of a task to view or set the affinity, or can be used to launch a command with a chosen CPU affinity. In order to set the affinity, **taskset** requires the CPU mask expressed 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.

1. 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
   ```

2. It is also possible to 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
   ```

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

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

4. Lastly, using the **-c** parameter, you can specify a CPU list instead of a CPU mask. For example, in order to use CPU 0, 4 and CPUs 7 to 11, the command line would contain **-c 0,4,7-11**. This invocation is more convenient in most cases.

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. See Section 3.6, “Non-Uniform Memory Access” for more information.

Related Manual Pages

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

- chrt(1)
- taskset(1)
- nice(1)
- renice(1)
- sched_setscheduler(2) for a description of the Linux scheduling scheme.

2.5. FILE SYSTEM DETERMINISM TIPS

The order in which journal changes arrive are sometimes not in the order that they are actually written to disk. The kernel I/O system has the option of reordering the journal changes, usually to try and make best use of available storage space. Journal activity can introduce latency through re-ordering journal changes and committing data and metadata. Often, journaling file systems can do things in such a way that they slow the system down.

The default filesystem used by Red Hat Enterprise Linux 7 is a journaling file system called **xfs**. A much earlier file system called **ext2** does not use journaling. Unless your organization specifically requires journaling, consider using **ext2**. In many of our best benchmark results, we utilize the **ext2** file system and consider it one of the top initial tuning recommendations.

Journaling file systems like **xfs** record the time a file was last accessed (**atime**). If using **ext2** is not a suitable solution for your system, consider disabling **atime** under **xfs** instead. Disabling **atime** increases performance and decreases power usage by limiting the number of writes to the filesystem journal.

Procedure 2.7. Disabling atime

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

   ```
   /dev/mapper/rhel-root       /       xfs    defaults…
   ```

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

   ```
   /dev/mapper/rhel-root       /       xfs    noatime,nodiratime…
   ```

IMPORTANT

Some applications rely on **atime** being updated. Therefore, this option is reasonable only on system 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.

Related Manual Pages

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

- mkfs.ext2(8)
- mkfs.xfs(8)
- mount(8) - for information on **atime**, **nodiratime** and **noatime**

2.6. USING HARDWARE CLOCKS FOR SYSTEM TIMESTAMPING

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. For the list of the available clock sources in your system, view the **/sys/devices/system/clocksource/clocksource0/available_clocksource** file:

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

In the example above, the TSC, HPET and ACPI_PM clock sources are available.

The clock source currently in use can be inspected by reading the **/sys/devices/system/clocksource/clocksource0/current_clocksource** file:

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

Changing Clock Sources

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 all these cases it is possible to override the clock selected by the kernel, provided that you understand the side effects of this override and can create an environment which will not trigger the known shortcomings of the given hardware clock. To do so, select a clock source from the list presented in the **/sys/devices/system/clocksource/clocksource0/available_clocksource** file and write the clock's name into the **/sys/devices/system/clocksource/clocksource0/current_clocksource** file. For example, the following command sets HPET as the clock source in use:

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

NOTE

For a brief description of widely used hardware clocks, and to compare the performance between different hardware clocks, see the Red Hat Enterprise Linux for Real Time Reference guide for Red Hat Enterprise Linux for Real Time.

Configuring Additional Boot Parameters for the TSC Clock

While there is no single clock which is ideal for all systems, TSC is generally the preferred clock source. To optimize the reliability of the TSC clock, you can configure additional parameters when booting the kernel, for example:

- ***idle=poll***: Forces the clock to avoid entering the idle state.
- ***processor.max_cstate=1***: Prevents the clock from entering deeper C-states (energy saving mode), so it does not become out of sync.

Note however that in both cases there will be an increase in energy consumption, as the system will always run at top speed.

Controlling Power Management Transitions

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 (as listed in Configuring Additional Boot Parameters for the TSC Clock), but with a more fine-grained control of power saving states.

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.

1. Open the **/dev/cpu_dma_latency** file. Keep the file descriptor open for the duration of the low-latency operation.

2. Write a 32-bit number to it. This number represents a maximum response time in microseconds. For the fastest possible response time, use ***0***.

   An example **/dev/cpu_dma_latency** file is as follows:

   ```
   static int pm_qos_fd = -1;

   void start_low_latency(void)
   {
    s32_t target = 0;

    if (pm_qos_fd >= 0)
     return;
    pm_qos_fd = open("/dev/cpu_dma_latency", O_RDWR);
    if (pm_qos_fd < 0) {
      fprintf(stderr, "Failed to open PM QOS file: %s",
              strerror(errno));
      exit(errno);
    }
    write(pm_qos_fd, &target, sizeof(target));
   }

   void stop_low_latency(void)
   {
    if (pm_qos_fd >= 0)
      close(pm_qos_fd);
   }
   ```

   The application will first call **start_low_latency()**, perform the required latency-sensitive processing, then call **stop_low_latency()**.

Related Manual Pages

For more information, or for further reading, the following book is related to the information given in this section.

- *Linux System Programming* by Robert Love

2.7. AVOID RUNNING EXTRA APPLICATIONS

These are common practices for improving performance, yet they are often overlooked. Here are some 'extra applications' to look for:

- Graphical desktop

  Do not run graphics where they are not absolutely required, especially on servers. To check if the system is configured to boot into the GUI by default, run the following command:

  ```
  ~]# systemctl get-default
  ```

  If you see **graphical.target**, reconfigure the system to boot into the text mode:

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

- Mail Transfer Agents (MTA, such as Sendmail or Postfix)

  Unless you are actively using Sendmail on the system you are tuning, disable it. If it is required, ensure it is well tuned or consider moving it to a dedicated machine.

  > **IMPORTANT**
  >
  > Sendmail is used to send system-generated messages, which are executed by programs such as cron. This includes reports generated by logging functions like logwatch. You will not be able to receive these messages if sendmail is disabled.

- Remote Procedure Calls (RPCs)
- Network File System (NFS)
- Mouse Services

  If you are not using a graphical interface like Gnome or KDE, then you probably do not need a mouse either. Remove the hardware and uninstall **gpm**.

- Automated tasks

  Check for automated **cron** or **at** jobs that could impact performance.

Remember to also check your third party applications, and any components added by external hardware vendors.

Related Manual Pages

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

- rpc(3)
- nfs(5)
- gpm(8)

2.8. SWAPPING AND OUT OF MEMORY TIPS

Memory Swapping

Swapping pages out to disk can introduce latency in any environment. To ensure low latency, the best strategy is to have enough memory in your systems so that swapping is not necessary. Always size the physical RAM as appropriate for your application and system. Use **vmstat** to monitor memory usage and watch the **si** (swap in) and **so** (swap out) fields. It is optimal that they remain zero as much as possible.

Procedure 2.8. Out of Memory (OOM)

Out of Memory (OOM) refers to a computing state where all available memory, including swap space, has been allocated. Normally this will cause the system to panic and stop functioning as expected. There is a switch that controls OOM behavior in **/proc/sys/vm/panic_on_oom**. When set to **1** the kernel will panic on OOM. The default setting is **0** which instructs the kernel to call a function named **oom_killer** on an OOM. Usually, **oom_killer** can kill rogue processes and the system will survive.

1. The easiest way to change this is to **echo** the new value to **/proc/sys/vm/panic_on_oom**.

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

~]# echo 1 > /proc/sys/vm/panic_on_oom

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

> **NOTE**
>
> It is recommended that you make the $RT; kernel panic on OOM. When the system has encountered an OOM state, it is no longer deterministic.

2. It is also possible to prioritize which processes get killed by adjusting the **oom_killer** score. In **/proc/*PID*/** there are two files named **oom_adj** and **oom_score**. Valid scores for **oom_adj** are in the range -16 to +15. This value is used to calculate the 'badness' of the process using an algorithm that also takes into account how long the process has been running, among other factors. To see the current **oom_killer** score, view the **oom_score** for the process. **oom_killer** will kill processes with the highest scores first.

   This example adjusts the **oom_score** of a process with a PID of 12465 to make it less likely that **oom_killer** will kill it.

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

~]# echo -5 > /proc/12465/oom_adj

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

3. There is also a special value of -17, which disables **oom_killer** for that process. In the example below, **oom_score** returns a value of ***O***, indicating that this process would not be killed.

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

~]# echo -17 > /proc/12465/oom_adj

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

Related Manual Pages

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

- swapon(2)
- swapon(8)
- vmstat(8)

2.9. NETWORK DETERMINISM TIPS

Transmission Control Protocol (TCP)

TCP can have a large effect on latency. TCP adds latency in order to obtain efficiency, control congestion, and to ensure reliable delivery. When tuning, consider the following points:

- Do you need ordered delivery?
- Do you need to guard against packet loss?

  Transmitting packets more than once can cause delays.
- If you must use TCP, consider disabling the Nagle buffering algorithm by using **TCP_NODELAY** on your socket. The Nagle algorithm collects small outgoing packets to send all at once, and can have a detrimental effect on latency.

Network Tuning

There are numerous tools for tuning the network. Here are some of the more useful:

Interrupt Coalescing

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

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

Use the **-C** (**--coalesce**) option with the **ethtool** command to enable.

Congestion

Often, I/O switches can be subject to back-pressure, where network data builds up as a result of full buffers.

Use the **-A** (**--pause**) option with the **ethtool** command to change pause parameters and avoid network congestion.

Infiniband (IB)

Infiniband is a type of communications architecture often used to increase bandwidth and provide quality of service and failover. It can also be used to improve latency through Remote Direct Memory Access (RDMA) capabilities.

Network Protocol Statistics

Use the **-s** (**--statistics**) option with the **netstat** command to monitor network traffic.

See also Section 2.12, “Reduce TCP Performance Spikes” and Section 3.7, “Reducing the TCP Delayed ACK Timeout”.

Related Manual Pages

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

- ethtool(8)
- netstat(8)

2.10. SYSLOG TUNING TIPS

syslog can forward log messages from any number of programs over a network. The less often this occurs, the larger the pending transaction is likely to be. If the transaction is very large an I/O spike can occur. To prevent this, keep the interval reasonably small.

Procedure 2.9. Using syslogd for System Logging.

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

1. To enable remote logging, you will first need to configure the machine that will receive the logs. See https://access.redhat.com/solutions/54363 for details.

2. Once remote logging support is enabled on the remote logging server, each system that will send logs to it must be configured to send its syslog output to the server, rather than writing those logs to the local file system. To do this, edit the **/etc/rsyslog.conf** file on each client system. For each of the various logging rules defined in that file, you can replace the local log file with the address of the remote logging server.

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

   The example above will cause the client system to log all kernel messages to the remote machine at ***@my.remote.logging.server***.

3. It is also possible to configure **syslogd** to log all locally generated system messages, by adding a wildcard line to the **/etc/rsyslog.conf** file:

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

IMPORTANT

Note that **syslogd** does not include built-in rate limiting on its generated network traffic. Therefore, we recommend that remote logging on Red Hat Enterprise Linux for Real Time systems be confined to only those messages that are required to be remotely logged by your organization. For example, kernel warnings, authentication requests, and the like. Other messages are locally logged.

Related Manual Pages

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

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

2.11. THE PC CARD DAEMON

The **pcscd** daemon is used to manage connections to PC and SC smart card readers. Although **pcscd** is usually a low priority task, it can often use more CPU than any other daemon. This additional background noise can lead to higher pre-emption costs to real-time tasks and other undesirable impacts on determinism.

Procedure 2.10. Disabling the**pcscd** Daemon

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

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

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

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

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

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

2.12. REDUCE TCP PERFORMANCE SPIKES

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

- Turn timestamps off with the following command:

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

- Turn timestamps on with the following command:

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

- Print the current value with the following command:

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

  The value ***1*** indicates that timestamps are on, the value ***0*** indicates they are off.

2.13. SYSTEM PARTITIONING

One of the key techniques for real-time tuning is partitioning the system. This means isolating a group of CPU cores for exclusive use of one or more real-time applications running on the system. For best results, partitioning should take into account the CPU topology so that related threads are placed on cores contained on the same Non-Uniform Memory Access (NUMA) node to maximize sharing of second and third-level caches. The **lscpu** and **tuna** utilities are used to determine the system CPU topology. The Tuna GUI allows you to dynamically isolate CPUs and move threads and interrupts from one CPU to another so that performance impacts can be measured.

Once a partitioning strategy has been determined based on the system layout and the structure of the application, the next step is to set the system to be partitioned automatically upon boot. For that, use the utilities provided by the tuned-profiles-realtime package. This package is installed by default when the Red Hat Enterprise Linux for Real Time packages are installed. To install tuned-profiles-realtime manually, run the following command as root:

```
~]# yum install tuned-profiles-realtime
```

The tuned-profiles-realtime package provides the **tuned** real-time profile that allows partitioning and other tunings at boot time with no additional user input required. Two configuration files control the behavior of the profile:

- **/etc/tuned/realtime-variables.conf**
- **/usr/lib/tuned/realtime/tuned.conf**

The **realtime-variables.conf** file specifies the group of CPU cores to be isolated. To isolate a group of CPU cores from the system, use the ***isolated_cores*** option as in the following example:

```
# Examples:
# isolated_cores=2,4-7
# isolated_cores=2-23
#
isolated_cores=1-3,5,9-14
```

In the example above, the profile places the CPUs 1, 2, 3, 5, 9, 10, 11, 12, 13, and 14 into an isolated CPU category; the only threads on these CPUs are kernel threads specifically bound to the cores. These kernel threads are only run when a specific condition is raised, such as the migration thread or the watchdog thread.

Once the ***isolated_cores*** variable is set, activate the profile with the **tuned-adm** command:

```
~]# tuned-adm profile realtime
```

The profile uses the **bootloader** plug-in. When activated, this plug-in adds the following boot parameters to the Linux kernel command line:

isolcpus

specifies CPUs listed in the **realtime-variables.conf** file

nohz

turns off the timer tick on an idle CPU; set to **off** by default

nohz_full

turns off the timer tick on a CPU when there is only one runnable task on that CPU; needs **nohz** to be set to **on**

intel_pstate=disable

prevents the Intel idle driver from managing power state and CPU frequency

nosoftlockup

prevents the kernel from detecting soft lockups in user threads

In the above example, the kernel boot command-line parameters look as follows:

```
isolcpus=1-3,5,9-14 nohz=on nohz_full=1-3,5,9-14 intel_pstate=disable nosoftlockup
```

The profile runs the **script.sh** shell script specified in the **[script]** section of **tuned.conf**. The script adjusts the following entries of the **sysfs** virtual file system:

- **/sys/bus/workqueue/devices/writeback/cpumask**
- **/sys/devices/system/machinecheck/machinecheck*/ignore_ce**

The **workqueue** entry above is set to the inverse of the isolated CPUs mask, while the second entry turns off machine check exceptions.

The script also sets the following variables in the **/etc/sysctl.conf** file:

```
kernel.hung_task_timeout_secs = 600
kernel.nmi_watchdog = 0
kernel.sched_rt_runtime_us = 1000000
vm.stat_interval = 10
```

The script uses the **tuna** interface to move any non-bound thread on the isolated CPU numbers off of the isolated CPUs.

For further tuning, copy the default **/usr/lib/tuned/realtime/script.sh** and modify it, then change the **tuned.conf** JSON file to point to the modified script.

2.14. REDUCE CPU PERFORMANCE SPIKES

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

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

CHAPTER 3. REALTIME-SPECIFIC TUNING

Once you have completed the optimization in Chapter 2, *General System Tuning* you are ready to start Red Hat Enterprise Linux for Real Time specific tuning. You must have the Red Hat Enterprise Linux for Real Time kernel installed for these procedures.

IMPORTANT

Do not attempt to use the tools in this section without first having completed Chapter 2, *General System Tuning*. You will not see a performance improvement.

When are you ready to begin Red Hat Enterprise Linux for Real Time tuning, perform these steps first, as they will provide the greatest benefit:

- Section 3.1, "Setting Scheduler Priorities"

When you are ready to start some fine-tuning on your system, then try the other sections in this chapter:

- Section 3.2, "Using **kdump** and **kexec** with the Red Hat Enterprise Linux for Real Time Kernel"
- Section 3.3, "TSC Timer Synchronization on Opteron CPUs"
- Section 3.4, "Infiniband"
- Section 3.6, "Non-Uniform Memory Access"
- Section 3.7, "Reducing the TCP Delayed ACK Timeout"

This chapter also includes information on performance monitoring tools:

- Section 3.9, "Using the **ftrace** Utility for Tracing Latencies"
- Section 3.10, "Latency Tracing Using **trace-cmd**"

Section 3.11, "Using **sched_nr_migrate** to Limit **SCHED_OTHER** Task Migration."

When you have completed all the tuning suggestions in this chapter, move on to Chapter 4, *Application Tuning and Deployment*

3.1. SETTING SCHEDULER PRIORITIES

The 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. It is possible that it will cause the system to become unresponsive and other unpredictable behavior if crucial kernel processes are prevented from running as needed. Ultimately the correct settings are workload-dependent.

Priorities are defined in groups, with some groups dedicated to certain kernel functions:

Table 3.1. Priority Map

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

Procedure 3.1. Using **systemd** to Set Priorities

- 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 following kernel boot.

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

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

3.1.1. Changing the priority of service during boot process

systemd makes it possible to set up real-time priority for services launched during the boot process.

The *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:

CPUSchedulingPolicy=

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

- other
- batch
- idle
- fifo
- rr

CPUSchedulingPriority=

Sets the CPU scheduling priority for 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.

Example 3.1. Changing the priority of the mcelog service

The following example uses the **mcelog** service. To change the priority of the **mcelog** service:

1. Create a supplementary **mcelog** service configuration directory file at **/etc/systemd/system/mcelog.system.d/priority.conf** as follows:

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

2. Insert the following:

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

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

```
# systemctl daemon-reload
```

4. Restart the **mcelog** service:

```
# systemctl restart mcelog
```

5. Display the **mcelog** priority set by **systemd** issue the following:

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

The output of this command should now be similar to the following:

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

For more information about changing the **systemd** *unit configuration directives* refer to the Modifying Existing Unit Files chapter of the System Administrator's Guide.

3.1.2. Configuring the CPU usage of a service

systemd makes it possible to specify which CPUs services are allowed to run on.

To do so, systemd uses the CPUAffinity= *unit configuration directive*.

Example 3.2. Configuring the CPU usage of the mcelog service

The following example restricts the **mcelog** service to run on CPUs 0 and 1:

1. Create a supplementary **mcelog** service configuration directory file at **/etc/systemd/system/mcelog.system.d/affinity.conf** as follows:

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

2. Insert the following:

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

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

```
# systemctl daemon-reload
```

4. Restart the **mcelog** service:

```
# systemctl restart mcelog
```

5. Display which CPUs the **mcelog** service is restricted to:

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

The output of this command should now be similar to the following:

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

For more information about changing the **systemd** *unit configuration directives*, refer to the Modifying Existing Unit Files chapter of the System Administrator's Guide.

3.2. USING KDUMP AND KEXEC WITH THE RED HAT ENTERPRISE LINUX FOR REAL TIME KERNEL

kdump is a reliable kernel crash dumping mechanism because the crash dump is captured from the context of a freshly booted kernel and not from the context of the crashed kernel. **kdump** uses a mechanism called **kexec** to boot into a second kernel whenever the system crashes. This second kernel, often called the crash kernel, boots with very little memory and captures the dump image.

If **kdump** is enabled on your system, the standard boot kernel will reserve a small section of system RAM and load the **kdump** kernel into the reserved space. When a kernel panic or other fatal error occurs, **kexec** is used to boot into the **kdump** kernel without going through BIOS. The system reboots to the **kdump** kernel that is confined to the memory space reserved by the standard boot kernel, and this kernel writes a copy or image of the system memory to the storage mechanism defined in the configuration files. Because **kexec** does not go through the BIOS, the memory of the original boot is retained, and the crash dump is much more detailed. Once this is done, the kernel reboots, which resets the machine and brings the boot kernel back up.

There are three required procedures for enabling **kdump** under Red Hat Enterprise Linux 7. First, ensure that the required RPM packages are installed on the system. Second, create the minimum configuration and modifies the **GRUB** command line using the **rt-setup-kdump** tool. Third, use a graphical system configuration tool called **system-config-kdump** to create and enable a detailed **kdump** configuration.

1. **Installing Required kdump Packages**
   The **rt-setup-kdump** tool is part of the rt-setup package. You also need kexec-tools and system-config-kdump:

   ```
   ~]# yum install rt-setup kexec-tools system-config-kdump
   ```

2. **Creating a Basic kdump Kernel withrt-setup-kdump**

   a. Run the **rt-setup-kdump** tool as **root**:

      ```
      ~]# rt-setup-kdump --grub
      ```

      The ***--grub*** parameter adds the necessary changes to all the real-time kernel entries listed in the **GRUB** configuration.

   b. Restart the system to set up the reserved memory space. You can then turn on the **kdump** init script and start the **kdump** service:

      ```
      ~]# systemctl enable kdump
      ~]# systemctl start kdump
      ```

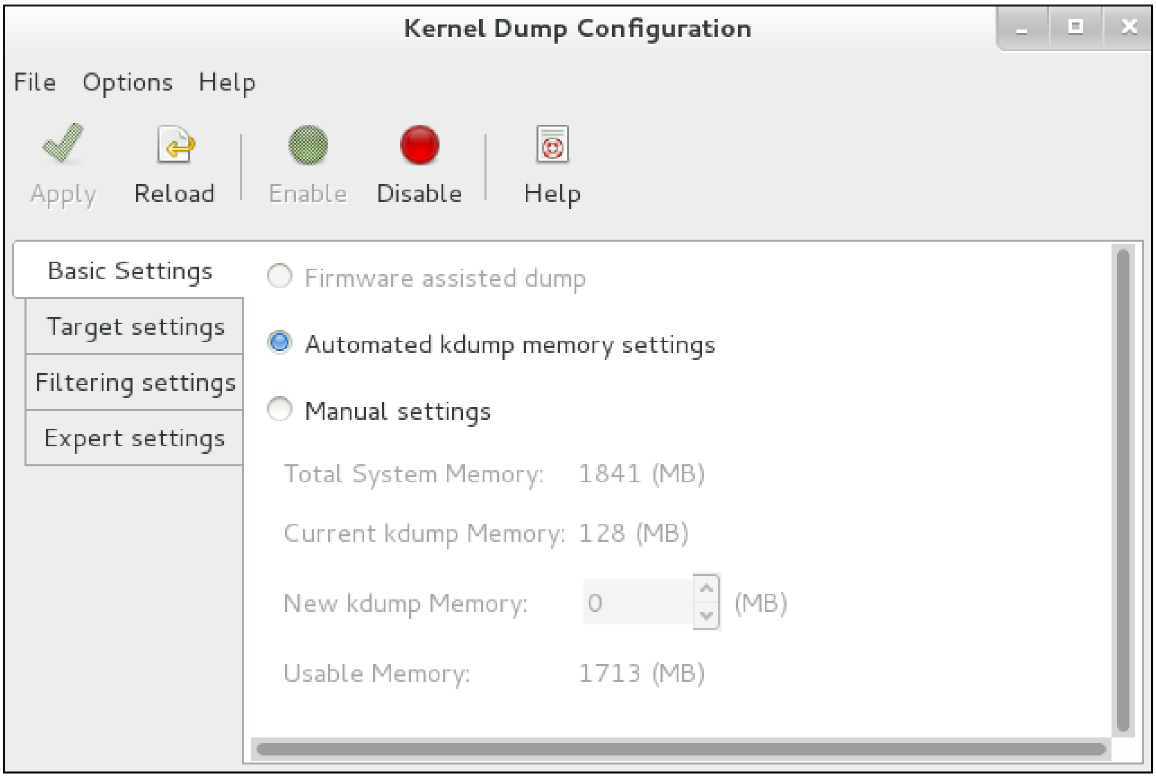
3. **Enabling kdump with system-config-kdump**

   a. Select the **Kernel crash dumps** system tool from the **Applications → System Tools** menu, or use the following command at the shell prompt:

      ```
      ~]# system-config-kdump
      ```

   b. The **Kernel Dump Configuration** window displays. On the toolbar, click the button labeled **Enable**. The Red Hat Enterprise Linux for Real Time kernel supports the ***crashkernel=auto*** parameter which automatically calculates the amount of memory necessary to accommodate the **kdump** kernel.

      By design, on Red Hat Enterprise Linux 7 systems with less than 4GB of RAM, the ***crashkernel=auto*** does not reserve any memory for the **kdump** kernel. In this case, it is necessary to manually set the amount of memory required. You can do so by entering your required value in the **New kdump memory** field on the **Basic Settings** tab:



○ Firmware assisted dump

● Automated kdump memory settings

○ Manual settings

Total System Memory: 1841 (MB)

Current kdump Memory: 128 (MB)

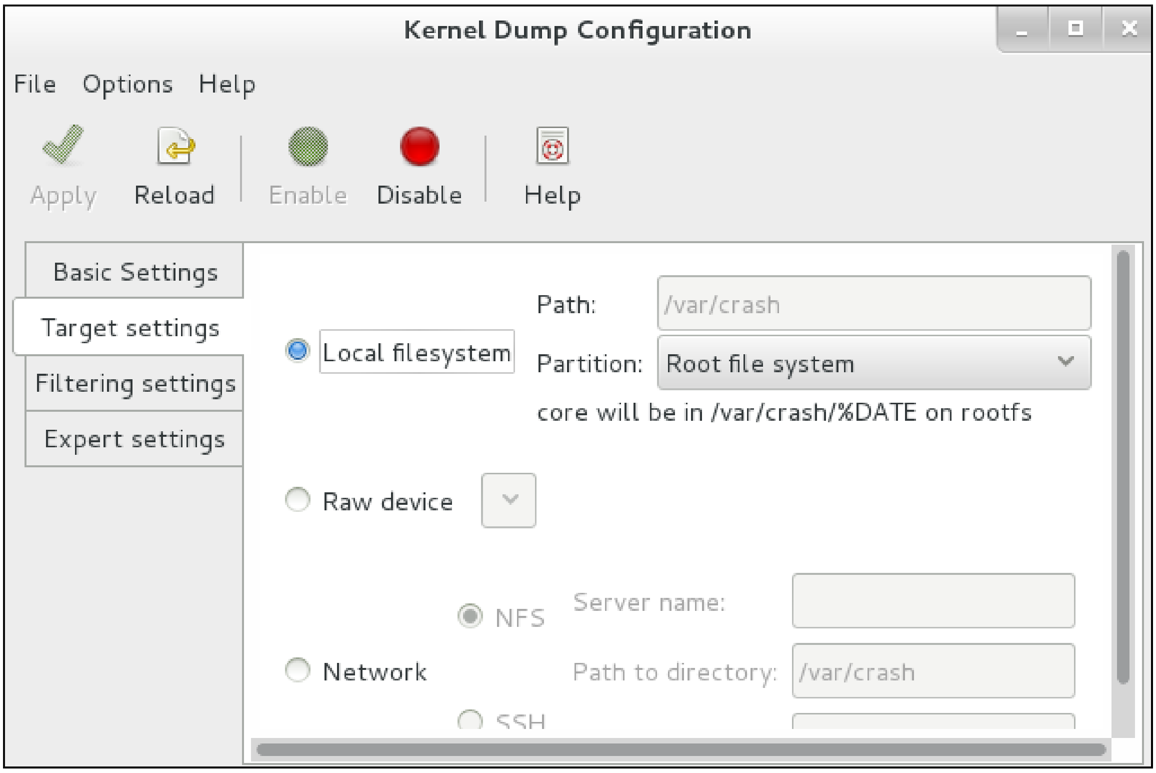
New kdump Memory: 0 (MB)

Usable Memory: 1713 (MB)

NOTE

An alternative way of allocating memory for the **kdump** kernel is by manually setting the ***crashkernel=<value>*** parameter in the **GRUB** configuration.

c. Click the **Target Settings** tab, and specify the target location for your dump file. It can be either stored as a file in a local file system, written directly to a device, or sent over a network using the NFS (Network File System) or SSH (Secure Shell) protocol.



● Local filesystem

Path: /var/crash

Partition: Root file system

core will be in /var/crash/%DATE on rootfs

○ Raw device

○ Network

● NFS

○ SSH

Server name:

Path to directory: /var/crash

To save your settings, click the **Apply** button on the toolbar.

d. Reboot the system to ensure that **kdump** is properly started. If you want to check that the **kdump** is working correctly, you can simulate a panic using **sysrq**:

```
~]# echo c > /proc/sysrq-trigger
```

This will cause the kernel to panic, and the system will boot into the **kdump** kernel. Once your system has been brought back up, you can check the log file at the specified location.

NOTE

Some hardware needs to be reset during the configuration of the **kdump** kernel. If you have any problems getting the **kdump** kernel to work, edit the **/etc/sysconfig/kdump** file and add **reset_devices=1** to the **KDUMP_COMMANDLINE_APPEND** variable.

IMPORTANT

On IBM LS21 machines, the following warning message can occur when attempting to boot the kdump kernel:

```
irq 9: nobody cared (try booting with the "irqpoll" option) handlers:
[<ffffffff811660a0>] (acpi_irq+0x0/0x1b)
turning off IO-APIC fast mode.
```

Some systems will recover from this error and continue booting, while some will freeze after displaying the message. This is a known issue. If you see this error, add the line **acpi=noirq** as a boot parameter to the kdump kernel. Only add this line if this error occurs as it can cause boot problems on machines not affected by this issue.

Related Manual Pages

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

- kexec(8)
- **/etc/kdump.conf**

3.3. TSC TIMER SYNCHRONIZATION 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. Red Hat Enterprise Linux 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.

Procedure 3.2. Enabling TSC Timer Synchronization

1. Open the **/etc/default/grub** file in your preferred text editor and append the parameters ***clocksource=tsc powernow-k8.tscsync=1*** to the **GRUB_CMDLINE_LINUX** variable. This forces the use of TSC and enables simultaneous core processor frequency transitions.

```
GRUB_CMDLINE_LINUX="rd.md=0 rd.lvm=0 rd.dm=0 $([ -x /usr/sbin/rhcrashkernel-param ]
&& /usr/sbin/rhcrashkernel-param || :) rd.luks=0 vconsole.keymap=us rhgb quiet
clocksource=tsc powernow-k8.tscsync=1"
```

2. You will need to restart your system for the changes to take effect.

Related Manual Pages

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

- gettimeofday(2)

3.4. INFINIBAND

Infiniband is a type of communications architecture often used to increase bandwidth and provide quality of service and failover. It can also be used to improve latency through Remote Direct Memory Access (RDMA) capabilities.

Support for Infiniband under Red Hat Enterprise Linux for Real Time does not differ from the support offered under Red Hat Enterprise Linux 7.

NOTE

For more information see Douglas Ledford's article on Getting Started with Infiniband.

3.5. ROCEE AND HIGH PERFORMANCE NETWORKING

RoCEE (RDMA over Converged Enhanced Ethernet) is a protocol that implements Remote Direct Memory Access (RDMA) over 10 Gigabit 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 10 Gigabit Ethernet infrastructure, resulting in less CPU overhead and reduced infrastructure costs.

Support for RoCEE and HPN under Red Hat Enterprise Linux for Real Time does not differ from the support offered under Red Hat Enterprise Linux 7.

NOTE

For more information on how to set up ethernet networks, see the Networking Guide.

3.6. NON-UNIFORM MEMORY ACCESS

Non-Uniform Memory Access (NUMA) is a design used to allocate memory resources to a specific CPU. This can improve access time and results in fewer memory locks. Although this appears as though it would be useful for reducing latency, NUMA systems have been known to interact badly with real-time applications, as they can cause unexpected event latencies.

As mentioned in Procedure 2.6, “Binding Processes to CPUs Using the **taskset** Utility” 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 .

Related Manual Pages

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

- numactl(8)

3.7. REDUCING THE TCP DELAYED ACK TIMEOUT

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

Quick ACK

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

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

Delayed ACK

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

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

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

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

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

NOTE

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

3.8. USING DEBUGFS

The **debugfs** file system is specially designed for debugging and making information available to users. It must be mounted for use with **ftrace** and **trace-cmd**, and it is mounted automatically in Red Hat Enterprise Linux 7 under the **/sys/kernel/debug/** directory.

You can verify that **debugfs** is mounted by running the following command:

```
~]# mount | grep ^debugfs
```

3.9. USING THE FTRACE UTILITY FOR TRACING LATENCIES

One of the diagnostic facilities provided with the Red Hat Enterprise Linux for Real Time kernel is **ftrace**, which is used by developers to analyze and debug latency and performance issues that occur outside of user-space. The **ftrace** utility has a variety of options that allow you to use the utility in a number of different ways. It can be used to trace context switches, measure the time it takes for a high-priority task to wake up, the length of time interrupts are disabled, or list all the kernel functions executed during a given period.

Some tracers, such as the function tracer, will produce exceedingly large amounts of data, which can turn trace log analysis into a time-consuming task. However, it is possible to instruct the tracer to begin and end only when the application reaches critical code paths.

The **ftrace** utility can be set up once the **trace** variant of the Red Hat Enterprise Linux for Real Time kernel is installed and in use.

Procedure 3.3. Using the ftrace Utility

1. In the **/sys/kernel/debug/tracing/** directory, there is a file named **available_tracers**. This file contains all the available tracers for **ftrace**. To see the list of available tracers, use the **cat** command to view the contents of the file:

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

   The user interface for **ftrace** is a series of files within **debugfs**. The **ftrace** files are also located in the **/sys/kernel/debug/tracing/** directory. Enter it:

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

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

   #### Ftrace Files

   The main files within this directory are:

   trace

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

trace_pipe

Like "trace" but is used to read the trace live. It is a producer / consumer trace, where each read will consume the event that is read. But this can be used to see 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 a 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.

Tracers

Depending on how the kernel was configured, not all tracers may be available for a given kernel. For the Red Hat Enterprise Linux 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.

function

One of the most widely applicable tracers. Traces the function calls within the kernel. Can cause noticeable overhead depending on the quantity of functions traced. Creates little overhead when not active.

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 that 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. 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 will hide 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. Records the time that it takes from the current highests priority task to wake up to the time it is scheduled. Only records the time for real time tasks.

preemptirqsoff

Traces the areas that disable pre-emption or interrupts, and records the maximum amount of time for which pre-emption or interrupts were disabled.

preemptoff

Similar to the **preemptirqsoff** tracer but traces only the maximum interval for which pre-emption was disabled.

irqsoff

Similar to the **preemptirqsoff** tracer but traces only the maximum interval for which interrupts were disabled.

nop

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

2. To manually start a tracing session, first select the tracer you wish to use from the list in **available_tracers** and then use the **echo** command to insert the name of the tracer into **/sys/kernel/debug/tracing/current_tracer**:

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

3. To check if **function** and **function_graph** tracing is enabled, use the **cat** command to view the **/sys/kernel/debug/tracing/options/function-trace** file. A value of ***1*** indicates that this is enabled, and ***0*** indicates that it has been disabled.

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

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

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

IMPORTANT

When using the **echo** command, ensure you place a space character in between the value and the **>** character. At the shell prompt, using **0>, 1>, and 2>** (without a space character) refers to standard input, standard output and standard error. Using them by mistake could result in unexpected trace output.

The **function-trace** option is useful because tracing latencies with **wakeup_rt**, **preemptirqsoff** and so on automatically enables function tracing, which may exaggerate the overhead.

4. Adjust details and parameters of the tracers by changing the values for the various files in the **/debugfs/tracing/** directory. Some examples are:

   The irqsoff, preemptoff, preempirqsoff, and wakeup tracers continuously monitor latencies. When they record a latency greater than the one recorded in **tracing_max_latency** the trace of that latency is recorded, and **tracing_max_latency** is updated to the new maximum time. In this way, **tracing_max_latency** will always show the highest recorded latency since it was last reset.

   To reset the maximum latency, **echo *0*** into the **tracing_max_latency** file. To see only latencies greater than a set amount, **echo** the amount in microseconds:

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

   When the tracing threshold is set, it overrides the maximum latency setting. When a latency is recorded that is greater than the threshold, it will be recorded regardless of the maximum latency. When reviewing the trace file, only the last recorded latency is shown.

   To set the threshold, **echo** the number of microseconds above which latencies must be recorded:

   ```
   ~]# echo 200 > /sys/kernel/debug/tracing/tracing_thresh
   ```

5. View the trace logs:

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

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

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

7. Function tracing can be filtered by altering the settings in the **/sys/kernel/debug/tracing/set_ftrace_filter** file. If no filters are specified in the file, all functions are traced. Use the **cat** to view the current filters:

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

8. To change the filters, **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.

   The ***** wildcard can also be used 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.

   Some examples of filters are:

   - 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
```

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.

3.10. LATENCY TRACING USING TRACE-CMD

trace-cmd is a front-end tool to **ftrace**. It can enable the **ftrace** interactions described earlier without needing to write into the **/sys/kernel/debug/tracing/** directory. It can be installed without the special tracing kernel variants, and it does not add any overhead when it is installed.

1. To install the **trace-cmd** tool, enter the following command as **root**:

```
~]# yum install trace-cmd
```

2. To start the utility, type **trace-cmd** at the shell prompt, along with the options you require, using the following syntax:

```
~]# trace-cmd command
```

Some examples of commands are:

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

  Enable and start recording functions executing within the kernel while *myapp* runs. It records functions from all CPUS and all tasks, even those not related to *myapp*.

-
  ```
  ~]# trace-cmd report
  ```

  Display the result.

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

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

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

  Enable all the IRQ events.

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

Start the **wakeup_rt** tracer.

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

  Start the **preemptirqsoff** tracer but disable function tracing in doing so. Note: the version of trace-cmd in Red Hat Enterprise Linux 7 turns off **ftrace_enabled** instead of using the **function-trace** option. You can enable it again with **trace-cmd start -p function**.

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

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

> NOTE
>
> See the trace-cmd(1) man page for a complete list of commands and options. All the individual commands also have their own man pages, trace-cmd-*command*. For further information about event tracing and function tracer, see Appendix A, *Event Tracing* and Appendix B, *Detailed Description of Ftrace* .

3. In this example, the **trace-cmd** utility will trace a single trace point:

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

3.11. USING SCHED_NR_MIGRATE TO LIMIT SCHED_OTHER 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 set 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 that is needed to disable interrupts. If there are a large number of tasks that need to be moved, it will occur while interrupts are disabled, so no timer events or wakeups will happen simultaneously. This can cause severe latencies for real-time tasks when the **sched_nr_migrate** is set to a large value.

Procedure 3.4. Adjusting the Value of the sched_nr_migrate Variable

1. Increasing the **sched_nr_migrate** variable gives high performance from **SCHED_OTHER** threads that spawn lots of tasks, at the expense of real-time latencies. For low real-time task latency at the expense of **SCHED_OTHER** task performance, the value must be lowered. The default value is 8.

2. To adjust the value of the **sched_nr_migrate** variable, you can **echo** the value directly to **/proc/sys/kernel/sched_nr_migrate**:

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

3.12. REAL TIME THROTTLING

Real Time Scheduling Issues

The two real-time scheduling policies in Red Hat Enterprise Linux for Real Time share one main characteristic: they run until they are preempted by a higher priority thread or until they "wait", either by sleeping or performing I/O. In the case of **SCHED_RR**, a thread may be preempted by the operating system so that another thread of equal **SCHED_RR** priority may run. In any 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 quite easy to write an application which monopolizes 100% of a given CPU. At first glance this sounds like it might be a good idea, but in reality it causes lots of headaches for the operating system. The OS 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. If a core is monopolized by a **SCHED_FIFO** thread, it cannot perform the housekeeping tasks and eventually the entire system becomes unstable, potentially causing a crash.

On the Red Hat Enterprise Linux for Real Time kernel, interrupt handlers run as threads with a **SCHED_FIFO** priority (default: 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 and cause programs waiting for data signaled by those interrupts to be starved and fail.

Real Time Scheduler Throttling

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

/proc/sys/kernel/sched_rt_period_us

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

/proc/sys/kernel/sched_rt_runtime_us

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

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

The impact of the default values is two-fold: rogue real-time tasks will not lock up the system by not allowing non-realtime tasks to run and, on the other hand, real-time tasks will have at most 95% of CPU time available from them, probably affecting their performance.

the **RT_RUNTIME_GREED** feature

Although the Real Time throttling mechanism works for the purpose of avoiding real-time tasks that can cause the system hang, an advanced user may want to allow the real-time task to continue running in the absence of non-realtime tasks starving, that is, avoiding the system going idle.

When enabled, this feature checks if non-realtime tasks are starving before throttling the real-time task. If the real-time task becomes throttled, it will be unthrottled as soon as the system goes idle, or when the next period starts, whichever comes first.

Enable **RT_RUNTIME_GREED** with the following command:

```
# echo RT_RUNTIME_GREED > /sys/kernel/debug/sched_features
```

To keep all CPUs with the same rt_runtime, disable the **NO_RT_RUNTIME_SHARE** logic:

```
# echo NO_RT_RUNTIME_SHARE > /sys/kernel/debug/sched_features
```

With these two options set, the user will guarantee some runtime for non-rt-tasks on all CPUs, while keeping real-time tasks running as much as possible.

References

From the kernel documentation, which is available in the kernel-rt-doc package:

- **/usr/share/doc/kernel-rt-doc-3.10.0/Documentation/scheduler/sched-rt-group.txt**

3.13. ISOLATING CPUS USING TUNED-PROFILES-REALTIME

To give application threads the most execution time possible, you can isolate CPUs, which means removing as many extraneous tasks off a CPU as possible. Isolating CPUs generally involves:

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

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

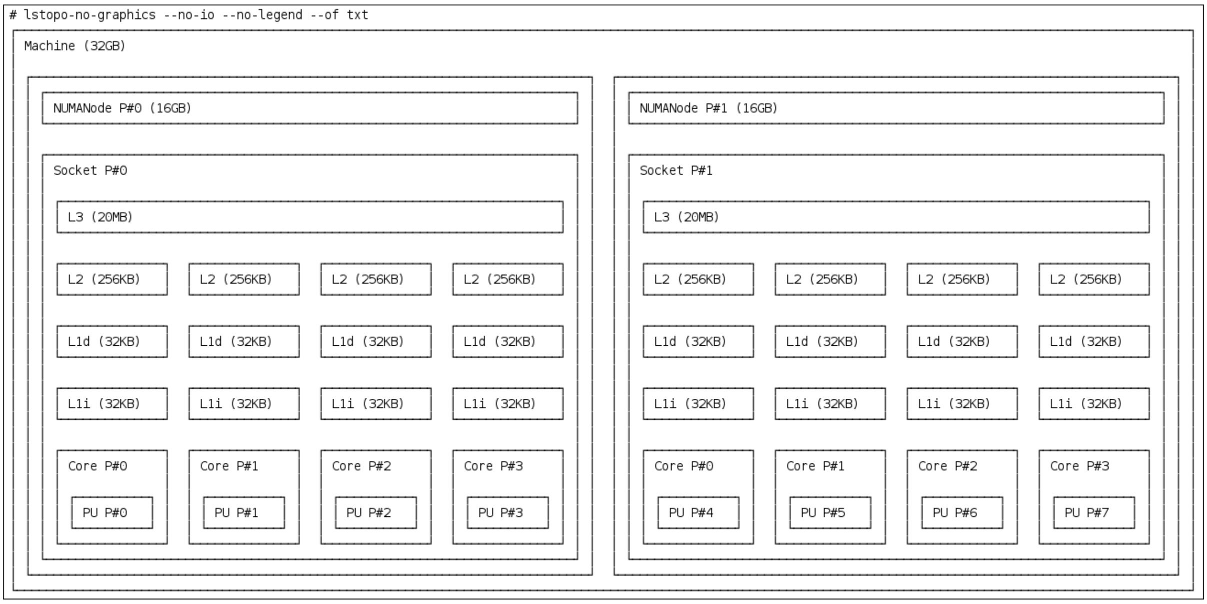
Choosing CPUs to Isolate

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

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

The hwloc package provides commands useful for getting information about CPUs, including **lstopo-no-graphics** and **numactl**:

- To show the layout of available CPUs in physical packages, use the **lstopo-no-graphics --no-io --no-legend --of txt** command:



```
# lstopo-no-graphics --no-io --no-legend --of txt
Machine (32GB)

  NUMANode P#0 (16GB)                                NUMANode P#1 (16GB)

  Socket P#0                                         Socket P#1

    L3 (20MB)                                          L3 (20MB)

    L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)     L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)

    L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)     L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)

    L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)     L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)

    Core P#0    Core P#1    Core P#2    Core P#3       Core P#0    Core P#1    Core P#2    Core P#3
      PU P#0      PU P#1      PU P#2      PU P#3         PU P#4      PU P#5      PU P#6      PU P#7
```

Figure 3.1. Showing the layout of CPUs using**lstopo-no-graphics**

The above 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** command, which produces graphical output.

- For further information about the CPUs, such as the distance between nodes, use the **numactl --hardware** command:

```
~]# 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
```

For more information about utilities provided by the hwloc package, see the **hwloc(7)** manpage.

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 Red Hat Enterprise Linux for Real Time is to use the **tuned** daemon and its tuned-profiles-realtime package.

To specify the ***isolcpus*** boot parameter, follow these steps:

1. Install the tuned package and the tuned-profiles-realtime package:

```
~]# yum install tuned tuned-profiles-realtime
```

2. In file **/etc/tuned/realtime-variables.conf**, set the configuration option **isolated_cores=*cpulist***, where *cpulist* is the list of CPUs that you want to isolate. The list is separated with commas and can contain single CPU numbers or ranges, for example:

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

   The above line would isolate CPUs 0, 1, 2, 3, 5, and 7.

   > **Example 3.3. Isolating CPUs with Communicating threads**
   >
   > In a two socket system with 8 cores, where NUMA node zero has cores 0-3 and NUMA node one has cores 4-8, to allocate two cores for a multi-threaded application, add this line:
   >
   > ```
   > isolated_cores=4,5
   > ```
   >
   > Once the **tuned-profiles-realtime** profile is activated, the ***isolcpus=4,5*** parameter will be added to the boot command line. This will prevent any user-space threads from being assigned to CPUs 4 and 5.

   > **Example 3.4. Isolating CPUs with Non-communicating threads**
   >
   > If you wanted to pick CPUs from different NUMA nodes for unrelated applications, you could specify:
   >
   > ```
   > isolated_cores=0,4
   > ```
   >
   > This would prevent any user-space threads from being assigned to CPUs 0 and 4.

3. Activate the **tuned** profile using the **tuned-adm** utility and then reboot:

   ```
   ~]# tuned-adm profile realtime
   ~]# reboot
   ```

4. Upon reboot, verify that the selected CPUs have been isolated by searching for the ***isolcpus*** parameter at the boot command line:

   ```
   ~]$ cat /proc/cmdline | grep isolcpus
   BOOT_IMAGE=/vmlinuz-3.10.0-394.rt56.276.el7.x86_64 root=/dev/mapper/rhel_foo-root ro
   crashkernel=auto rd.lvm.lv=rhel_foo/root rd.lvm.lv=rhel_foo/swap console=ttyS0,115200n81
   isolcpus=0,4
   ```

Isolating CPUs Using the *nohz* and *nohz_full* Parameters

The ***nohz*** and ***nohz_full*** kernel boot parameters can also be specified in **/etc/tuned/realtime-variables.conf** to isolate CPUs:

nohz=on

May be used to reduce timer activity on a particular set of CPUs. The ***nohz*** parameter is mainly used to reduce timer interrupts happening on idle CPUs. This helps battery life by allowing the idle CPUs to run in reduced power mode. While not being directly useful for real-time response time, the ***nohz*** parameter does not directly hurt real-time response time and is required to activate the next parameter which *does* have positive implications for real-time performance.

nohz_full=*cpulist*

The ***nohz_full*** parameter is used to treat a list of CPUs differently, with respect to timer ticks. If a CPU is listed as a nohz_full CPU and there is only one runnable task on the CPU, then the kernel will stop sending timer ticks to that CPU, so more time may be spent running the application and less time spent servicing interrupts and context switching.

For more information on these parameters, see Configuring kernel tick time

3.14. OFFLOADING RCU CALLBACKS

The Read-Copy-Update (RCU) system is a lockless mechanism for mutual exclusion 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.

RCU callbacks can be offloaded using the ***rcu_nocbs*** and ***rcu_nocb_poll*** kernel parameters.

- 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, which can be moved to a housekeeping CPU, relieving CPU 3 from doing RCU callbacks job.

  To move RCU callback threads to the housekeeping CPU, use the **tuna -t rcu* -c *X* -m** command, where *X* denotes the housekeeping CPU. For example, in a system where CPU 0 is the housekeeping CPU, all RCU callback threads can be moved to CPU 0 using this command:

  ```
  ~]# tuna -t rcu* -c 0 -m
  ```

  This relieves all CPUs other than CPU 0 from doing RCU work.

- Although the RCU offload threads can perform the RCU callbacks on another CPU, each CPU is responsible for awakening the corresponding RCU offload thread. To relieve each CPU from the responsibility of awakening their RCU offload threads, set the ***rcu_nocb_poll*** kernel parameter:

  ```
  rcu_nocb_poll
  ```

  With ***rcu_nocb_poll*** set, the RCU offload threads will be periodically raised by a timer to check if there are callbacks to run.

A common use case for these two options is:

1. Using ***rcu_nocbs=cpulist*** to allow the user to move all RCU offload threads to a housekeeping CPU;

2. Setting ***rcu_nocb_poll*** to relieve each CPU from the responsibility awakening their RCU offload threads.

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

For more information about RCU tuning on real-time, see Avoiding RCU Stalls in the real-time kernel .

CHAPTER 4. APPLICATION TUNING AND DEPLOYMENT

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

NOTE

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

Further Reading

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

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 for use in real-time applications as an event delivery mechanism. The reason for this is that the current Linux kernel signal handling code is quite complex, due mainly to legacy behavior and the multitude of APIs that need to be supported. This complexity means that the code paths that are taken when delivering a signal are not always optimal, and quite long latencies can be experienced by applications.

The original motivation behind UNIX™ signals was to multiplex one thread of control (the process) between different "threads" of execution. Signals behave somewhat like operating system interrupts - when a signal is delivered to an application, the application's context is saved and it starts executing a previously registered signal handler. Once the signal handler has completed, the application returns to executing where it was when the signal was delivered. This can get complicated in practice.

Signals are too non-deterministic to trust them in a real-time application. A better option is to use POSIX Threads (pthreads) to distribute your workload and communicate between various components. You can coordinate groups of threads using the pthreads mechanisms of mutexes, condition variables and barriers and trust that the code paths through these relatively new constructs are much cleaner than the legacy handling code for signals.

Further Reading

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

RTWiki's Build an RT Application

Ulrich Drepper's Requirements of the POSIX Signal Model

4.2. USING SCHED_YIELD AND OTHER SYNCHRONIZATION MECHANISMS

The **sched_yield** system call is used by a thread allowing other threads a chance to run. Often when **sched_yield** is used, the thread can go to the end of the run queues, taking a long time to be scheduled again, or it can be rescheduled straight away, creating a busy loop on the CPU. The scheduler is better able to determine when and if there are actually other threads wanting to run. Avoid using **sched_yield** on any RT task.

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

Related Manual Pages

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

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

4.3. MUTEX OPTIONS

Procedure 4.1. Standard Mutex Creation

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

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

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

   ```
   pthread_mutex_t my_mutex;

   pthread_mutex_init(&my_mutex, NULL);
   ```
3. In this case, your application will not benefit from the advantages provided by the pthreads API and the Red Hat Enterprise Linux for Real Time kernel. There are a number of mutex options that must be considered when writing or porting an application.

Procedure 4.2. Advanced Mutex Options

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

> **IMPORTANT**
>
> For the sake of brevity, these examples do not include a check of the return value of the function. This is a basic safety procedure and one that you must always perform.

1. Creating the mutex object:

   ```
   pthread_mutex_t my_mutex;

   pthread_mutexattr_t my_mutex_attr;
   ```

```
pthread_mutexattr_init(&my_mutex_attr);
```

2. Shared and Private mutexes:

   Shared mutexes can be used between processes, however they can create a lot more overhead.

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

3. Real-time priority inheritance:

   Priority inversion problems can be avoided by using priority inheritance.

   ```
   pthread_mutexattr_setprotocol(&my_mutex_attr, PTHREAD_PRIO_INHERIT);
   ```

4. Robust mutexes:

   Robust mutexes are released when the owner dies, however this can also come at a high overhead cost. **_NP** in this string indicates that this option is non-POSIX or not portable.

   ```
   pthread_mutexattr_setrobust_np(&my_mutex_attr, PTHREAD_MUTEX_ROBUST_NP);
   ```

5. Mutex initialization:

   Once the attributes are set, initialize a mutex using those properties.

   ```
   pthread_mutex_init(&my_mutex, &my_mutex_attr);
   ```

6. Cleaning up the attributes object:

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

   ```
   pthread_mutexattr_destroy(&my_mutex_attr);
   ```

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

Related Manual Pages

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

- futex(7)
- pthread_mutex_destroy(P)

  For information on **pthread_mutex_t** and **pthread_mutex_init**
- pthread_mutexattr_setprotocol(3p)

  For information on **pthread_mutexattr_setprotocol** and **pthread_mutexattr_getprotocol**
- pthread_mutexattr_setprioceiling(3p)

For information on **pthread_mutexattr_setprioceiling** and **pthread_mutexattr_getprioceiling**

4.4. TCP_NODELAY AND SMALL BUFFER WRITES

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

Procedure 4.3. Using TCP_NODELAY and TCP_CORK to Improve Network Latency

1. Applications that require lower latency on every packet sent must be run on sockets with **TCP_NODELAY** enabled. It can be enabled through the **setsockopt** command with the sockets API:

   ```
   # int one = 1;

   # setsockopt(descriptor, SOL_TCP, TCP_NODELAY, &one, sizeof(one));
   ```

2. For this to be used effectively, applications must avoid doing small, logically related buffer writes. Because **TCP_NODELAY** is enabled, these small writes will make TCP send these multiple buffers as individual packets, which can result in poor overall performance.

   If applications have several buffers that are logically related, and are to be sent as one packet, it is possible to build a contiguous packet in memory and then send the logical packet to TCP on a socket configured with **TCP_NODELAY**.

   Alternatively, create an I/O vector and pass it to the kernel using **writev** on a socket configured with **TCP_NODELAY**.

3. Another option is to use **TCP_CORK**, which tells TCP to wait for the application to remove the cork before sending any packets. This command will cause the buffers it receives to be appended to the existing buffers. This allows applications to build a packet in kernel space, which can be required when using different libraries that provides abstractions for layers. To enable **TCP_CORK**, set it to a value of ***1*** using the **setsockopt** sockets API (this is known as "corking the socket"):

   ```
   # int one = 1;

   # setsockopt(descriptor, SOL_TCP, TCP_CORK, &one, sizeof(one));
   ```

4. When the logical packet has been built in the kernel by the various components in the application, tell TCP to remove the cork. TCP will send the accumulated logical packet right away, without waiting for any further packets from the application.

   ```
   # int zero = 0;

   # setsockopt(descriptor, SOL_TCP, TCP_CORK, &zero, sizeof(zero));
   ```

Related Manual Pages

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

- tcp(7)

- setsockopt(3p)
- setsockopt(2)

4.5. SETTING REAL-TIME SCHEDULER PRIORITIES

Using **systemd** to set scheduler priorities is described in Procedure 3.1, "Using **systemd** to Set Priorities". In the example given in that procedure, some kernel threads could have been given a very high priority. This is to have the default priorities integrate well with the requirements of the Real Time Specification for Java (RTSJ). RTSJ requires a range of priorities from 10 to 89.

For deployments where RTSJ is not in use, there is a wide range of scheduling priorities below 90 which are at the disposal of applications. It is usually dangerous for user level applications to run at priority 50 and above - despite the fact that the capability exists. Preventing essential system services from running can result in unpredictable behavior, including blocked network traffic, blocked virtual memory paging and data corruption due to blocked filesystem journaling.

Use extreme caution when scheduling any application thread above priority 49. If any application threads are scheduled above priority 89, ensure that the threads only run a very short code path. Failure to do so would undermine the low latency capabilities of the Red Hat Enterprise Linux for Real Time kernel.

Setting Real-time Priority for Non-privileged Users

Generally, only root users are able to change priority and scheduling information. If you require non-privileged users to be able to adjust these settings, the best method is to add the user to the **realtime** group.

IMPORTANT

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

4.6. LOADING DYNAMIC LIBRARIES

When developing your real-time application, consider resolving symbols at startup. Although it can slow down program initialization, it is one way to avoid non-deterministic latencies during program execution.

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

The following is an example shell script. This script exports the **LD_BIND_NOW** variable with a value of ***1***, then runs a program with a scheduler policy of FIFO and a priority of ***1***.

```
#!/bin/sh

LD_BIND_NOW=1
export LD_BIND_NOW

chrt --fifo 1 /opt/myapp/myapp-server &
```

Related Manual Pages

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

- ld.so(8)

4.7. USING _COARSE POSIX CLOCKS FOR APPLICATION TIMESTAMPING

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

To illustrate that concept, imagine using a clock, inside a drawer, to time events being observed. If every time one has to open the drawer, get the clock and only then read the time, the cost of reading the clock is too high and can lead to missing events or incorrectly timestamping them.

Conversely, a clock on the wall would be faster to read, and timestamping would produce less interference to the observed events. Standing right in front of that wall clock would make it even faster to obtain time readings.

Likewise, this performance gain (in reducing the cost of reading the clock) can be obtained by selecting a hardware clock that has a faster reading mechanism. In Red Hat Enterprise Linux for Real Time, a further performance gain can be acquired by using POSIX clocks with the **clock_gettime()** function to produce clock readings with the lowest cost possible.

POSIX Clocks

POSIX clocks is a standard for implementing and representing time sources. The POSIX clocks can be selected by each application, without affecting other applications in the system. This is in contrast to the hardware clocks as described in Section 2.6, "Using Hardware Clocks for System Timestamping" , which is selected by the kernel and implemented across the system.

The function used to read a given POSIX clock is **clock_gettime()**, which is defined at **<time.h>**. **clock_gettime()** has a counterpart in the kernel, in the form of a system call. When the user process calls **clock_gettime()**, the corresponding C library (**glibc**) calls the **sys_clock_gettime()** system call which performs the requested operation and then returns the result to the user program.

However, this context switch from the user application to the kernel has a cost. Even though this cost is very low, if the operation is repeated thousands of times, the accumulated cost can have an impact on the overall performance of the application. To avoid that context switch to the kernel, thus making it faster to read the clock, support for the **CLOCK_MONOTONIC_COARSE** and **CLOCK_REALTIME_COARSE** POSIX clocks was created in the form of a VDSO library function.

Time readings performed by **clock_gettime()**, using one of the _***COARSE*** clock variants, do not require kernel intervention and are executed entirely in user space, which yields a significant performance gain. Time readings for _***COARSE*** clocks have a millisecond (ms) resolution, meaning that time intervals smaller than 1ms will not be recorded. The _***COARSE*** variants of the POSIX clocks are suitable for any application that can accommodate millisecond clock resolution, and the benefits are more evident on systems which use hardware clocks with high reading costs.

NOTE

To compare the cost and resolution of reading POSIX clocks with and without the _***COARSE*** prefix, see the Red Hat Enterprise Linux for Real Time Reference guide for Red Hat Enterprise Linux for Real Time.

Example 4.1. Using the *_COARSE* Clock Variant in clock_gettime

```
#include <time.h>

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

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

You can improve upon the example above, for example by using more strings to verify the return code of **clock_gettime()**, to verify the value of the ***rc*** variable, or to ensure the content of the ***ts*** structure is to be trusted. The **clock_gettime()** manpage provides more information to help you write more reliable applications.

IMPORTANT

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

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

Related Manual Pages

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

- clock_gettime()
- *Linux System Programming* by Robert Love
- *Understanding The Linux Kernel* by Daniel P. Bovet and Marco Cesati

4.8. ABOUT PERF

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

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

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

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

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

Example 4.2. Example of perf Options

```
]# perf

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

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

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

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

Example 4.3. Perf Record

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

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

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

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

Example 4.4. Example of the Perf Report and Archive Features

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

Run the archive command to create an archive of results.

```
~]# perf archive
```

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

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

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

```
~]# perf report
```

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

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

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

Example 4.5. Example of the Perf List and Stat Features

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

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

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

            15,619 context-switches

        5.002060064 seconds time elapsed
```

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

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

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

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

                 5 ext4:ext4_request_inode

       5.002253620 seconds time elapsed
```

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

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

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

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

IMPORTANT

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

CHAPTER 5. MORE INFORMATION

5.1. REPORTING BUGS

Diagnosing a Bug

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

1. Check that you have the latest version of the Red Hat Enterprise Linux 7 kernel, then boot into it from the **GRUB** menu. Try reproducing the problem with the standard kernel. If the problem still occurs, report a bug against Red Hat Enterprise Linux 7.

2. If the problem does not occur when using the standard kernel, then the bug is probably the result of changes introduced in the Red Hat Enterprise Linux for Real Time specific enhancements Red Hat has applied on top of the baseline (3.10.0) kernel.

Reporting a Bug

If you have determined that the bug is specific to Red Hat Enterprise Linux for Real Time follow these instructions to enter a bug report:

1. Create a Bugzilla account if you do not have it yet..

2. Click on Enter A New Bug Report . Log in if necessary.

3. Select the **Red Hat** classification.

4. Select the **Red Hat Enterprise Linux 7** product.

5. If it is a kernel issue, enter **kernel-rt** as the component. Otherwise, enter the name of the affected user-space component, such as **trace-cmd**.

6. Continue to enter the bug information by giving a detailed problem description. When entering the problem description be sure to include details of whether you were able to reproduce the problem on the standard Red Hat Enterprise Linux 7 kernel.

APPENDIX A. EVENT TRACING

See Event Tracing by Theodore Ts'o .

APPENDIX B. DETAILED DESCRIPTION OF FTRACE

```
ftrace - Linux kernel internal tracer

Introduction
------------

Ftrace is an internal tracer for the Linux kernel. It is designed to
follow the processing of what happens within the kernel as that is
normally a black box. It allows the user to trace kernel functions
that are called in real time, as well as to see various events like
tasks scheduling, interrupts, disk activity and other services that
the kernel provides.

Ftrace was intorduced to Linux in the 2.6.27 kernel, and has increased
in functionality ever since. It is not meant to trace what is happening
inside user applications, but can be used to trace within system calls
that user applications make.


The Debug File System
---------------------

The user interface for ftrace is a series of files within the debug
file system that is usually mounted at /sys/kernel/debug. The ftrace
files are in the tracing directory that can be accessed at
/sys/kernel/debug/tracing.

Note, there is also a user interface tool called trace-cmd. See later
in this document for more information about that tool.

In order to mount the debug filesystem, perform the following:

 mount -t debugfs nodev /sys/kernel/debug

Then you can change directory into the ftrace tracing location:

 cd /sys/kernel/debug/tracing

Note, all these files can only be modified by root user, as enabling
tracing can have an impact on the performance of the system.


Ftrace files
------------

The main files within this directory are:

 trace - the file that shows the output of a ftrace trace. This is
       really a snapshot of the trace in time, as it stops tracing as
       this file is read, and it does not consume the events read.
       That is, if the user disabled tracing and read this file, it
       will always report the same thing every time its read.
```

```
        Also, to clear the trace buffer, simply write into this file.

          >#  echo > trace

        This will erase the entire contents of the trace buffer.

  trace_pipe - like "trace" but is used to read the trace live. It is
        a producer / consumer trace, where each read will consume the
        event that is read. But this can be used to see 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 a 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.
        Note, 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.

There are several other files, but we will get to them as they come
up with functionalities of the tracers.


Tracers and Events
------------------

Tracers have specific functionality within the kernel, where as events
are just some kind of data that is recorded into the ftrace buffer.
To understand this more, we need to take a look at the tracers themselves
and the events as well.

nop
---

The default tracer is called "nop". It is just a nop tracer, and does not
provide any tracing facility itself. But, as events may interleave into
any tracer, the "nop" tracer is what is used if you are only interested
in tracing events.

When the "nop" tracer is active and the trace buffer is empty, the "trace"
file shows the following:

>#  cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 0/0   #P:8
#
#                              _-------=> irqs-off
#                             /  _------=> need-resched
#                             |/  _-----=> need-resched_lazy
#                              ||/  _----=> hardirq/softirq
#                              |||/  _---=> preempt-depth
#                              ||||/  _--=> preempt-lazy-depth
#                              ||||| / _-=> migrate-disable
#                              |||||| /     delay
#           TASK-PID   CPU#  |||||||    TIMESTAMP  FUNCTION
#              | |       |   |||||||       |         |


It starts with what tracer is active and then gives a default header.

Now to enable an event, you must write an ASCII '1' into the "enable"
file for the particular event.

>># echo 1 > events/sched/sched_switch/enable
>># cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 463/463   #P:8
#
#                                _-------=> irqs-off
#                              /  _------=> need-resched
#                              |/  _-----=> need-resched_lazy
#                              ||/  _----=> hardirq/softirq
#                              |||/  _---=> preempt-depth
#                              ||||/  _--=> preempt-lazy-depth
#                              ||||| / _-=> migrate-disable
#                              |||||| /     delay
#           TASK-PID   CPU#  |||||||    TIMESTAMP  FUNCTION
#              | |       |   |||||||       |         |
            bash-1367  [007] d...... 11927.750484: sched_switch: prev_comm=bash prev_pid=1367
prev_prio=120 prev_state=S ==> next_comm=kworker/7:1 next_pid=121 next_prio=120
     kworker/7:1-121   [007] d...... 11927.750514: sched_switch: prev_comm=kworker/7:1
prev_pid=121 prev_prio=120 prev_state=S ==> next_comm=swapper/7 next_pid=0 next_prio=120
          <idle>-0     [000] d...... 11927.750531: sched_switch: prev_comm=swapper/0 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=sshd next_pid=1365 next_prio=120
          <idle>-0     [007] d...... 11927.750555: sched_switch: prev_comm=swapper/7 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=kworker/7:1 next_pid=121 next_prio=120
     kworker/7:1-121   [007] d...... 11927.750575: sched_switch: prev_comm=kworker/7:1
prev_pid=121 prev_prio=120 prev_state=S ==> next_comm=swapper/7 next_pid=0 next_prio=120
            sshd-1365  [000] d...... 11927.750673: sched_switch: prev_comm=sshd prev_pid=1365
prev_prio=120 prev_state=S ==> next_comm=swapper/0 next_pid=0 next_prio=120
          <idle>-0     [001] d...... 11927.752568: sched_switch: prev_comm=swapper/1 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=kworker/1:1 next_pid=57 next_prio=120
          <idle>-0     [002] d...... 11927.752589: sched_switch: prev_comm=swapper/2 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=rcu_sched next_pid=10 next_prio=120
     kworker/1:1-57    [001] d...... 11927.752590: sched_switch: prev_comm=kworker/1:1 prev_pid=57
prev_prio=120 prev_state=S ==> next_comm=swapper/1 next_pid=0 next_prio=120
       rcu_sched-10    [002] d...... 11927.752610: sched_switch: prev_comm=rcu_sched prev_pid=10
prev_prio=120 prev_state=S ==> next_comm=swapper/2 next_pid=0 next_prio=120
          <idle>-0     [007] d...... 11927.753548: sched_switch: prev_comm=swapper/7 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=rcu_sched next_pid=10 next_prio=120
       rcu_sched-10    [007] d...... 11927.753568: sched_switch: prev_comm=rcu_sched prev_pid=10
prev_prio=120 prev_state=S ==> next_comm=swapper/7 next_pid=0 next_prio=120
          <idle>-0     [007] d...... 11927.755538: sched_switch: prev_comm=swapper/7 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=kworker/7:1 next_pid=121 next_prio=120
```

```
As you can see there is quite a lot of information that is displayed
by simply enabling the sched_switch event.

Events
------

The events are broken up into "systems". Each system of events has its
own directory under the "events" directory located in the ftrace "tracing"
directory in the debug file system.

># ls -F events
block/      header_event  lock/     printk/      skb/      vsyscall/
compaction/  header_page   mce/      random/      sock/     workqueue/
drm/        i915/        migrate/  raw_syscalls/  sunrpc/    writeback/
enable      irq/         module/   rcu/         syscalls/
ext4/       jbd2/        napi/     rpm/         task/
ftrace/     kmem/        net/      sched/       timer/
hda/        kvm/         oom/      scsi/        udp/
hda_intel/   kvmmmu/      power/    signal/      vmscan/

Each of these directories represent a system or group of events. Notice that
there's three files in this directory:

enable
header_event
header_page

The only one you should be concerned about is the "enable" file, as that
will enable all events when an ASCII '1' is written into it and disable
all events when an ASCII '0' is written into it.

The header_event and header_page provides information necessary for
the trace-cmd tool.

Each of these directories shows the events that are within that system:

># ls -F events/sched
enable                 sched_process_exit/  sched_stat_sleep/
filter                 sched_process_fork/  sched_stat_wait/
sched_kthread_stop/      sched_process_free/  sched_switch/
sched_kthread_stop_ret/  sched_process_wait/  sched_wait_task/
sched_migrate_task/      sched_stat_blocked/  sched_wakeup/
sched_pi_setprio/        sched_stat_iowait/   sched_wakeup_new/
sched_process_exec/      sched_stat_runtime/


Each directory here represents a single event. Notice that there's two
files in the system directory:

enable
filter

The "enable" file here can enable or disable all events within the system
when an ASCII '1' or '0', respectively, is written to this file.
```

The "filter" file will be described shortly.

Within the individual event directories exist control files:

```
>># ls -F events/sched/sched_wakeup/
enable  filter  format  id
```

We already used the "enable" file. Now to explain the other files.

The "format" file shows the fields that are written when the event is enabled, as well as the fields that can be used for the filter.

The "id" file is used by the perf tool and is not something that needs to be delt with here.

```
>># cat events/sched/sched_wakeup/format
name: sched_wakeup
ID: 249
format:
        field:unsigned short common_type;       offset:0;       size:2; signed:0;
        field:unsigned char common_flags;       offset:2;       size:1; signed:0;
        field:unsigned char common_preempt_count;       offset:3;       size:1; signed:0;
        field:int common_pid;   offset:4;       size:4; signed:1;
        field:unsigned short common_migrate_disable;    offset:8;       size:2; signed:0;
        field:unsigned short common_padding;    offset:10;      size:2; signed:0;

        field:char comm[16];    offset:16;      size:16;        signed:1;
        fieldid_t pid;  offset:32;      size:4; signed:1;
        field:int prio; offset:36;      size:4; signed:1;
        field:int success;      offset:40;      size:4; signed:1;
        field:int target_cpu;   offset:44;      size:4; signed:1;

print fmt: "comm=%s pid=%d prio=%d success=%d target_cpu=%03d", REC->comm, REC->pid,
REC->prio, REC->success, REC->target_cpu
```

This file is also used by perf and trace-cmd to tell how to read the raw binary output from the tracing buffers for the event. But what you need to know is the field names, as they are used by the filtering.

The first set of fields before the blank line are the common fields that exist for all events. The specific fields for the event come after the blank line and here it starts with "comm".

Filtering events

There are times when you may not want to trace all events, but only events where one of the event's fields contains a certain value.
The "filter" file allows for this.

```
The filter provides the following predicates:

For numerical fields:

 ==, !=, <, <=, >, >=


For string fields:

 ==, !=, ~


Logical && and || as well as parenthesis are also acceptable.

The syntax is

 <filter> = FIELD <pred-num> | FIELD <pred-string> |
    '(' <filter> ')' | <filter> '&&' <filter> | <filter> '||' <filter>

 <pred-num> = <num-op> <number>

 <pred-string> = <string-op> <string>

 <num-op> = '==' | '!=' | '<' | '<=' | '>' | '>='

 <string-op> = '==' | '!=' | '~'

 <number> = <digits> | '0x'<hex-number>

 <digits> = [0-9] | <digits><digits>

 <hex-number> = [0-9] | [a-f] | [A-F] | <hex-number><hex-number>

 <string> = '"' VALUE '"'

The glob expression '~' is a very simple glob. it can only be:

 <glob> = VALUE | '*' VALUE | VALUE '*' | '*' VALUE '*'

That is, anything more complex will not be valid. Such as:

  VALUE '*' VALUE


What the glob does is to match a string with wild cards at the beginning
or end or both, of a value:

  comm ~ "kwork*"


Example:

To trace all schedule switches to a real time task:

>=# echo 'next_prio < 100' > events/sched/sched_switch/filter
>=# cat events/sched/sched_switch/filter
```

```
next_prio < 100
># cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 11/11   #P:8
#
#                              _-------=> irqs-off
#                            /  _------=> need-resched
#                            |/  _-----=> need-resched_lazy
#                            ||/  _----=> hardirq/softirq
#                            |||/  _---=> preempt-depth
#                            ||||/  _--=> preempt-lazy-depth
#                            ||||| / _-=> migrate-disable
#                            |||||| /     delay
#           TASK-PID   CPU#  |||||||    TIMESTAMP  FUNCTION
#              | |       |   |||||||       |         |
          <idle>-0     [001] d...... 14331.192687: sched_switch: prev_comm=swapper/1 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=rtkit-daemon next_pid=992 next_prio=0
          <idle>-0     [001] d...... 14333.737030: sched_switch: prev_comm=swapper/1 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=watchdog/1 next_pid=12 next_prio=0
          <idle>-0     [000] d...... 14333.738023: sched_switch: prev_comm=swapper/0 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=watchdog/0 next_pid=11 next_prio=0
          <idle>-0     [002] d...... 14333.751985: sched_switch: prev_comm=swapper/2 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=watchdog/2 next_pid=17 next_prio=0
          <idle>-0     [003] d...... 14333.765947: sched_switch: prev_comm=swapper/3 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=watchdog/3 next_pid=22 next_prio=0
          <idle>-0     [004] d...... 14333.779933: sched_switch: prev_comm=swapper/4 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=watchdog/4 next_pid=27 next_prio=0
          <idle>-0     [005] d...... 14333.794114: sched_switch: prev_comm=swapper/5 prev_pid=0
prev_prio=120 prev_state=R ==> next_comm=watchdog/5 next_pid=32 next_prio=0



Task priorities
---------------

This is a good time to explain task priorities, as the tracer reports them
differently than the way user processes see priorities. A task has priority
policies that are SCHED_OTHER, SCHED_FIFO and SCHED_RR. By default
tasks are assigned SCHED_OTHER which runs under the kernels Completely
Fail Scheduler (CFS), where as SCHED_FIFO and SCHED_RR runs under
the real-time scheduler. The real-time scheduler has 99 different priorities
ranging from 1 - 99, where 99 is the highest priority and 1 is the lowest.
This is set by sched_setscheduler(2).

If you noticed above, to show real time tasks, the filter used
"next_prio < 100". Ftrace reports the internal kernel version of priorities
for tasks and not the priority that a task sees. This can be a little
confusing. For user real-time priorities of 1 through 99 are mapped
internally as 98 to 0, where 0 is the highest priority and 98 is the lowest
of the real time priorities. All non real-time tasks show a priority of 120,
as CFS does not use the priority to determine which tasks to run, although
it does use a nice value, but that's not represented by the prio field
reported in the traces.
```

```
Tracers
-------

Depending on how the kernel was configured, not all tracers may be available
for a given kernel.For the Red Hat Enterprise Linux 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.


To see what tracers are available for the kernel, cat out the contents
of "available_tracers":

># cat available_tracers
function_graph wakeup_rt wakeup preemptirqsoff preemptoff irqsoff function nop


The "nop" tracer has already been discussed and is available in all
kernels.


The "function" tracer
---------------------

The most popular tracer aside from the "nop" tracer is the "function"
tracer. This tracer traces the function calls within the kernel.
Depending on how many functions are tracer or which specific functions,
it can cause a very noticeable overhead when tracing is active.

Note, due to a clever trick with code modification, the function tracer
induces very little overhead when not active. This is because the
hooks in the function calls to be traced are converted into nops on
boot, and are only converted back to hooks into the tracer when activated.

># echo function > current_tracer
># cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 319338/253106705   #P:8
#
#                                _-------=> irqs-off
#                               /  _------=> need-resched
#                               |/  _-----=> need-resched_lazy
#                               ||/  _----=> hardirq/softirq
#                               |||/  _---=> preempt-depth
#                               ||||/  _--=> preempt-lazy-depth
#                               ||||| / _-=> migrate-disable
#                               |||||| /     delay
#           TASK-PID   CPU#  |||||||    TIMESTAMP  FUNCTION
#              | |       |   |||||||       |         |
     kworker/5:1-58    [005] ....... 32462.200700: smp_call_function_single <-
cpufreq_get_measured_perf
     kworker/5:1-58    [005] d...... 32462.200700: read_measured_perf_ctrs <-smp_call_function_single
```

```
    kworker/5:1-58    [005] ....... 32462.200701: cpufreq_cpu_put <-__cpufreq_driver_getavg
    kworker/5:1-58    [005] ....... 32462.200702: module_put <-cpufreq_cpu_put
    kworker/5:1-58    [005] ....... 32462.200702: od_check_cpu <-dbs_check_cpu
    kworker/5:1-58    [005] ....... 32462.200702: usecs_to_jiffies <-od_dbs_timer
    kworker/5:1-58    [005] ....... 32462.200703: schedule_delayed_work_on <-od_dbs_timer
    kworker/5:1-58    [005] ....... 32462.200703: queue_delayed_work_on <-
schedule_delayed_work_on
    kworker/5:1-58    [005] d...... 32462.200704: __queue_delayed_work <-queue_delayed_work_on
    kworker/5:1-58    [005] d...... 32462.200704: get_work_gcwq <-__queue_delayed_work
    kworker/5:1-58    [005] d...... 32462.200704: get_cwq <-__queue_delayed_work
    kworker/5:1-58    [005] d...... 32462.200705: add_timer_on <-__queue_delayed_work
    kworker/5:1-58    [005] d...... 32462.200705: _raw_spin_lock_irqsave <-add_timer_on
    kworker/5:1-58    [005] d...... 32462.200705: internal_add_timer <-add_timer_on


Filtering on functions
----------------------

As tracing all functions can be induce a substantial overhead, as well
as adding a lot of noise to the trace (you may not be interested in every
function call), ftrace provides a way to limit what functions can be
traced. There are two files for this purpose:

 set_ftrace_filter

 set_ftrace_notrace


For a list of functions that can be traced, as well as added to these files:

 available_filter_functions


By writing a name of a function into the "set_ftrace_filter" file, the
function tracer will only trace that function.

># echo schedule_delayed_work > set_ftrace_filter
># cat set_ftrace_filter
schedule_delayed_work
># cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 8/8   #P:8
#
#                              _-------=> irqs-off
#                             / _------=> need-resched
#                             |/ _-----=> need-resched_lazy
#                             ||/ _----=> hardirq/softirq
#                             |||/ _---=> preempt-depth
#                             ||||/ _--=> preempt-lazy-depth
#                             ||||| / _-=> migrate-disable
#                             |||||| /     delay
#           TASK-PID   CPU#  |||||||    TIMESTAMP  FUNCTION
#              | |       |   |||||||       |         |
    kworker/0:2-1586  [000] ....... 32820.361913: schedule_delayed_work <-vmstat_update
    kworker/2:1-62    [002] ....... 32820.370891: schedule_delayed_work <-vmstat_update
    kworker/3:2-5004  [003] ....... 32820.373881: schedule_delayed_work <-vmstat_update
    kworker/0:2-1586  [000] ....... 32820.448658: schedule_delayed_work <-do_cache_clean
    kworker/4:1-61    [004] ....... 32820.537541: schedule_delayed_work <-vmstat_update
    kworker/4:1-61    [004] ....... 32820.537546: schedule_delayed_work <-sync_cmos_clock
    kworker/7:1-121   [007] ....... 32820.897372: schedule_delayed_work <-vmstat_update
    kworker/1:1-57    [001] ....... 32820.898361: schedule_delayed_work <-vmstat_update


Note, modifications to these files follows shell concatenation rules:

>#  cat set_ftrace_filter
schedule_delayed_work
>#  echo do_IRQ > set_ftrace_filter
>#  cat set_ftrace_filter
do_IRQ

Notice that writing with '>' into set_ftrace_filter cleared what was
currently in the file and replaced it with the new contents. Just
writing into the file will clear it:

>#  cat set_ftrace_filter
do_IRQ
>#  echo > set_ftrace_filter
>#  cat set_ftrace_filter
#### all functions enabled ####

To append to the list, use the shell append operation '>>':

>#  cat set_ftrace_filter
do_IRQ
>#  echo schedule_delayed_work >> set_ftrace_filter
>#  cat set_ftrace_filter
schedule_delayed_work
do_IRQ


Note, the order of functions displayed has nothing to do with how they
were added. Their order is dependent upon how the functions are layed
out in the kernel internal function list table.


Globs
-----

Functions can be added to these files with the same type of glob
expressions described in the event filtering section. The format is
identical:

 <glob> = VALUE | '*' VALUE | VALUE '*' | '*' VALUE '*'


If you want to trace all functions that start with "sched":

>#  echo 'sched*' > set_ftrace_filter
>#  cat set_ftrace_filter
schedule_delayed_work_on
```

```
schedule_delayed_work
schedule_work_on
schedule_work
schedule_on_each_cpu
sched_feat_open
sched_feat_show
[...]
># echo function > current_tracer
># cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 1270/1270   #P:8
#
#                                _-------=> irqs-off
#                               / _------=> need-resched
#                               |/ _-----=> need-resched_lazy
#                               ||/ _----=> hardirq/softirq
#                               |||/ _---=> preempt-depth
#                               ||||/ _--=> preempt-lazy-depth
#                               ||||| / _-=> migrate-disable
#                               |||||| /     delay
#            TASK-PID   CPU#  |||||||    TIMESTAMP  FUNCTION
#               | |       |   |||||||       |         |
            bash-1367  [001] ....... 34240.654888: schedule_work <-tty_flip_buffer_push
            bash-1367  [001] .N..... 34240.654902: schedule <-sysret_careful
     kworker/1:1-57    [001] ....... 34240.654921: schedule <-worker_thread
          <idle>-0     [000] .N..... 34240.654949: schedule <-cpu_idle
            bash-1367  [001] ....... 34240.655069: schedule_work <-tty_flip_buffer_push
            bash-1367  [001] .N..... 34240.655079: schedule <-sysret_careful
            sshd-1365  [000] ....... 34240.655087: schedule_timeout <-wait_for_common
            sshd-1365  [000] ....... 34240.655088: schedule <-schedule_timeout



set_ftrace_notrace
------------------

There are cases were you may want to trace everything except for various
functions that you don't care about. Perhaps there's functions that cause
too much noise in the trace, for example, perhaps locks are showing
up in the trace and you don't care about them:

># echo '*lock*' > set_ftrace_notrace
># cat set_ftrace_notrace
update_persistent_clock
read_persistent_clock
set_task_blockstep
user_enable_block_step
read_hv_clock
__acpi_acquire_global_lock
__acpi_release_global_lock
cpu_hotplug_driver_lock
cpu_hotplug_driver_unlock
[...]

But notice that you also included functions that have "clock" and "block"
```

```
in their names. To remove them but still keep the "lock" functions, use
the '!' symbol:


>#  echo '!*clock*' >> set_ftrace_notrace
>#  echo '!*block*' >> set_ftrace_notrace
>#  cat set_ftrace_notrace
__acpi_acquire_global_lock
__acpi_release_global_lock
cpu_hotplug_driver_lock
cpu_hotplug_driver_unlock
lock_vector_lock
unlock_vector_lock
console_lock
console_trylock
console_unlock
is_console_locked
kmsg_dump_get_line_nolock
[...]

But remember to use '>>' instead of '>', as that will clear out all
functions in the file.



Latency tracers
---------------

As stated, the difference between events and tracers, is that events
just enable recording some specific information within the kernel.
Traces have a bit more impact. Function tracing, in essence, also
just records information, but it requires a bit more work than enabling
a static tracepoint (event). Also, to limit what function tracing can
trace, requires writing into control files for the function tracer.

Another type of tracer is the latency tracers. These record a snapshot
of the trace when the latency is greater than the previously recorded
latency. There are two types of latency tracers, one kind records the
length of time when activities within the kernel are disabled, and the
other records the time it takes from when a task is woken from sleep
to the time it gets scheduled.

tracing_max_latency
-------------------

A latency tracer will just keep track of a snapshot of a trace when a new
max latency is hit. To see the current max latency time, cat the contents
of the file "tracing_max_latency". This file can also be used to set
the max time. Either to reset it back to zero or some lesser number to
trigger new snapshots of latencies, or to set it to a greater number to
not record anything unless a latency has exceeded some given time.

The unit of time that "tracing_max_latency" uses (as well as all other
tracing files, unless otherwise specified) is microseconds.
```

```
irqsoff tracer
--------------

A common use of the tracing facility is to see how long interrupts have
been disabled for. When interrupts are disabled, the system cannot
respond to external events, which can include a packet coming in on the
network card, or perhaps a task on another CPU woke up a task on the current
CPU and sent an interprocessor interrupt (IPI) to tell the current CPU
to run the new task. With interrupts disabled, the current CPU will
ignore all external events, which is a source of latencies. This is why
monitorying how long interrupts are disabled can show why the system
did not react in a proper time that was expected.

The irqsoff tracer traces the time interrupts are disabled to the time
they are enabled again. If the time interrupts were disabled is larger
than the time specified by "tracing_max_latency" has, then it will
save the current trace off to a "snapshot" buffer, reset the current
buffer and continue tracing looking for the next time interrupts
are off for a long time.

Here's an example of how to use irqsoff tracer:

>\# echo 0 > tracing_max_latency
>\# echo irqsoff > current_tracer
>\# sleep 10
>\# cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.8.13-test-mrg-rt9+
# --------------------------------------------------------------------
# latency: 523 us, #1301/1301, CPU#2 | (Mreempt VP:0, KP:0, SP:0 HP:0 #P:8)
#    -----------------
#    | task: swapper/2-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: cpu_idle
#  => ended at:   cpu_idle
#
#
#                   _--------=> CPU#
#                  / _-------=> irqs-off
#                 | / _------=> need-resched
#                 || / _-----=> need-resched_lazy
#                 ||| / _----=> hardirq/softirq
#                 |||| / _---=> preempt-depth
#                 ||||| / _--=> preempt-lazy-depth
#                 |||||| / _-=> migrate-disable
#                 ||||||| /     delay
#  cmd     pid    |||||||| time  |   caller
#     \   /       ||||||||  \   |   /
  <idle>-0       2dN..1..    0us : tick_nohz_idle_exit <-cpu_idle
  <idle>-0       2dN..1..    1us : menu_hrtimer_cancel <-tick_nohz_idle_exit
  <idle>-0       2dN..1..    1us : ktime_get <-tick_nohz_idle_exit
  <idle>-0       2dN..1..    1us : tick_do_update_jiffies64 <-tick_nohz_idle_exit
  <idle>-0       2dN..1..    2us : update_cpu_load_nohz <-tick_nohz_idle_exit
  <idle>-0       2dN..1..    2us : _raw_spin_lock <-update_cpu_load_nohz
  <idle>-0       2dN..1..    3us : add_preempt_count <-_raw_spin_lock
  <idle>-0       2dN..2..    3us : __update_cpu_load <-update_cpu_load_nohz
  <idle>-0       2dN..2..    4us : sub_preempt_count <-update_cpu_load_nohz
  <idle>-0       2dN..1..    4us : calc_load_exit_idle <-tick_nohz_idle_exit
  <idle>-0       2dN..1..    5us : touch_softlockup_watchdog <-tick_nohz_idle_exit
  <idle>-0       2dN..1..    5us : hrtimer_cancel <-tick_nohz_idle_exit

[...]

  <idle>-0       2dN..1..  521us : account_idle_time <-irqtime_account_process_tick.isra.2
  <idle>-0       2dN..1..  521us : irqtime_account_process_tick.isra.2 <-account_idle_ticks
  <idle>-0       2dN..1..  521us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       2dN..1..  522us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       2dN..1..  522us : account_idle_time <-irqtime_account_process_tick.isra.2
  <idle>-0       2dN..1..  522us : irqtime_account_process_tick.isra.2 <-account_idle_ticks
  <idle>-0       2dN..1..  522us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       2dN..1..  523us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       2dN..1..  523us : account_idle_time <-irqtime_account_process_tick.isra.2
  <idle>-0       2dN..1..  523us : tick_nohz_idle_exit <-cpu_idle
  <idle>-0       2dN..1..  524us+: trace_hardirqs_on <-cpu_idle
  <idle>-0       2dN..1..  537us : <stack trace>
 => tick_nohz_idle_exit
 => cpu_idle
 => start_secondary


By default, the irqsoff tracer enables function tracing to show what functions
are being called while interrupts were disabled. But as you can see, it
can produce a lot of output (the total line count of the above trace
was 1,327 lines. Most of that was cut to not waste space in this document).
The problem with the function tracer is that it incurs a substantial overhead
and exagerates the actual latency.

The reported latency above is 523 microseconds. The trace ends at 537
microseconds, but that's because it took 14 microseconds to produce the
stack trace.

The end of the trace does a stack dump to show where the latency occurred.
The above happened in tick_nohz_idle_exit(), and even though we can blame
the function tracer for exagerating the latency, this trace shows
that using NO HZ idle can have issues with a real time system. When a
system with NO HZ set is idle, the timer tick is stopped. When the system
resumes from idle, the timer must catch up to the current time and executes
all the ticks it missed in the loop. This is done with interrupts disabled.

Looking at the latency field "2dN..1.." you can see that this loop
ran on CPU 2, had interrupts disabled "d". The scheduler needed to run
"N" (for NEED_RESCHED). Preemption was disabled, as the preempt_count
counter was set to "1".

Ideally, when coming out of NO HZ, the accounting could be done in a single
step, but as that is tricky to get right, the current method is to just
run the current code in a loop as if the timer went off each time.


No function tracing
-------------------
```

As function tracing can exaggerate the latency, you can either limit what functions are traced via the "set_ftrace_filter" and "set_ftrace_notrace" files as described above in the function tracing section. But you can also disable tracing totally via the tracing option function-trace.

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

This disables function tracing by all the ftrace tracers. Including the function tracer, which would make it rather pointless because the function tracer would act just like the "nop" tracer.

```
>## echo 0 > options/function-trace
>## echo 0 > tracing_max_latency
>## echo irqsoff > current_tracer
>## sleep 10
>## cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.8.13-test-mrg-rt9+
# -----------------------------------------------------------------
# latency: 80 us, #4/4, CPU#6 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:8)
#    -----------------
#    | task: swapper/6-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: cpu_idle
#  => ended at:   cpu_idle
#
#
#                  _--------=> CPU#
#                 / _-------=> irqs-off
#                | / _------=> need-resched
#                || / _-----=> need-resched_lazy
#                ||| / _----=> hardirq/softirq
#                |||| / _---=> preempt-depth
#                ||||| / _--=> preempt-lazy-depth
#                |||||| / _-=> migrate-disable
#                ||||||| /     delay
#  cmd     pid   |||||||| time  |   caller
#     \   /      ||||||||  \   |   /
  <idle>-0       6dN..1..    0us+: tick_nohz_idle_exit <-cpu_idle
  <idle>-0       6dN..1..   81us : tick_nohz_idle_exit <-cpu_idle
  <idle>-0       6dN..1..   81us+: trace_hardirqs_on <-cpu_idle
  <idle>-0       6dN..1..   87us : <stack trace>
 => tick_nohz_idle_exit
 => cpu_idle
 => start_secondary
```

This time the latency is much more compact and accurate (80 microseconds is still a lot, but much lower than 523). Here the backtrace is much more important as its now the only real information to know where the latency occurred.

```
preemptoff tracer
-----------------

There are points in the kernel that disables preemption but not interrupts.
That is, an interrupt can still interrupt the current process but that
process cannot be scheduled out for a higher priority process.

This tracer records the time that preemption is disabed via the
kernel internal "preempt_disable()" function.

># echo 0 > /sys/kernel/debug/tracing/options/function-trace
># echo 0 > tracing_max_latency
># echo preemptoff > current_tracer
># sleep 10
># cat trace
# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.8.13-test-mrg-rt9+
# --------------------------------------------------------------------
# latency: 65 us, #4/4, CPU#6 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:8)
#    -----------------
#    | task: swapper/6-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: cpuidle_enter
#  => ended at:   start_secondary
#
#
#                   _--------=> CPU#
#                  / _-------=> irqs-off
#                 | / _------=> need-resched
#                 || / _-----=> need-resched_lazy
#                 ||| / _----=> hardirq/softirq
#                 |||| / _---=> preempt-depth
#                 ||||| / _--=> preempt-lazy-depth
#                 |||||| / _-=> migrate-disable
#                 ||||||| /     delay
#  cmd     pid    |||||||| time  |   caller
#     \   /      ||||||||  \   |   /
  <idle>-0       6d...1..    1us+: intel_idle <-cpuidle_enter
  <idle>-0       6.N..1..   65us : cpu_idle <-start_secondary
  <idle>-0       6.N..1..   66us+: trace_preempt_on <-start_secondary
  <idle>-0       6.N..1..   71us : <stack trace>
 => sub_preempt_count
 => cpu_idle
 => start_secondary


There's not much interesting in this trace except that preemption was
disabled for 65 microseconds.


preemptirqsoff tracer
---------------------
```

```
Knowing when interrupts are disabled or how long preemption is disabled
via the preempt_disable() kernel interface is not as interesting as
knowing how long true preemption is disabled. That is, if we have the
following scenario:

A) preempt_disable()

[...]

B) irqs_disable()

[...]

C) preempt_enable();

[...]

D) irqs_enable();


"irqsoff" tracer will give you the time from B to D
"preemptoff" tracer will give you the time from A to C.

But the current task cannot be preempted from A to D which is what we
really care about. When a task cannot be preempted, a new task can
no execute when it is woken up if it is to run on the same CPU as the
task that has true preemption disabled (either interrupts disabled or
preemption disabled). The "preemptirqsoff" tracer will handle this.

"preemptirqsoff" tracer will give you the time from A to D


>># echo 1 > /sys/kernel/debug/tracing/options/function-trace
>># echo 0 > tracing_max_latency
>># echo preemptirqsoff > current_tracer
>># sleep 10
>># cat trace
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.8.13-test-mrg-rt9+
# --------------------------------------------------------------------
# latency: 377 us, #1289/1289, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:8)
#    -----------------
#    | task: swapper/1-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: cpuidle_enter
#  => ended at:   start_secondary
#
#
#                  _--------=> CPU#
#                 / _-------=> irqs-off
#                | / _------=> need-resched
#                || / _-----=> need-resched_lazy
#                ||| / _----=> hardirq/softirq
#                |||| / _---=> preempt-depth
#                ||||| / _--=> preempt-lazy-depth
#                 |||||| / _-=> migrate-disable
#                 ||||||| /     delay
#  cmd     pid   |||||||| time  |   caller
#     \   /      ||||||||  \   |   /
  <idle>-0       1d...1..    0us : intel_idle <-cpuidle_enter
  <idle>-0       1d...1..    1us : ktime_get <-cpuidle_wrap_enter
  <idle>-0       1d...1..    2us : smp_reschedule_interrupt <-reschedule_interrupt
  <idle>-0       1d...1..    3us : scheduler_ipi <-smp_reschedule_interrupt
  <idle>-0       1d...1..    3us : irq_enter <-scheduler_ipi
  <idle>-0       1d...1..    4us : rcu_irq_enter <-irq_enter
  <idle>-0       1d...1..    4us : rcu_eqs_exit_common.isra.45 <-rcu_irq_enter
  <idle>-0       1d...1..    5us : tick_check_idle <-irq_enter
  <idle>-0       1d...1..    5us : tick_check_oneshot_broadcast <-tick_check_idle
  <idle>-0       1d...1..    5us : ktime_get <-tick_check_idle
  <idle>-0       1d...1..    6us : tick_nohz_stop_idle <-tick_check_idle
  <idle>-0       1d...1..    6us : update_ts_time_stats <-tick_nohz_stop_idle
  <idle>-0       1d...1..    7us : nr_iowait_cpu <-update_ts_time_stats
  <idle>-0       1d...1..    7us : touch_softlockup_watchdog <-sched_clock_idle_wakeup_event
  <idle>-0       1d...1..    7us : tick_do_update_jiffies64 <-tick_check_idle
  <idle>-0       1d...1..    8us : touch_softlockup_watchdog <-tick_check_idle
  <idle>-0       1d...1..    8us : irqtime_account_irq <-irq_enter
  <idle>-0       1d...1..    9us : in_serving_softirq <-irqtime_account_irq
  <idle>-0       1d...1..    9us : add_preempt_count <-irq_enter
  <idle>-0       1d..h1..    9us : sched_ttwu_pending <-scheduler_ipi
  <idle>-0       1d..h1..   10us : _raw_spin_lock <-sched_ttwu_pending
  <idle>-0       1d..h1..   10us : add_preempt_count <-_raw_spin_lock
  <idle>-0       1d..h2..   11us : sub_preempt_count <-sched_ttwu_pending
  <idle>-0       1d..h1..   11us : raise_softirq_irqoff <-scheduler_ipi
  <idle>-0       1d..h1..   12us : do_raise_softirq_irqoff <-raise_softirq_irqoff
  <idle>-0       1d..h1..   12us : irq_exit <-scheduler_ipi
  <idle>-0       1d..h1..   12us : irqtime_account_irq <-irq_exit
  <idle>-0       1d..h1..   13us : sub_preempt_count <-irq_exit
  <idle>-0       1d...2..   13us : wakeup_softirqd <-irq_exit
  <idle>-0       1d...2..   14us : wake_up_process <-wakeup_softirqd
  <idle>-0       1d...2..   14us : try_to_wake_up <-wake_up_process

[...]

  <idle>-0       1d...4..   18us : dequeue_rt_stack <-enqueue_task_rt
  <idle>-0       1d...4..   19us : cpupri_set <-enqueue_task_rt
  <idle>-0       1d...4..   20us : update_rt_migration <-enqueue_task_rt
  <idle>-0       1d...4..   20us : ttwu_do_wakeup <-ttwu_do_activate.constprop.90
  <idle>-0       1d...4..   20us : check_preempt_curr <-ttwu_do_wakeup
  <idle>-0       1d...4..   21us : resched_task <-check_preempt_curr
  <idle>-0       1dN..4..   21us : task_woken_rt <-ttwu_do_wakeup
  <idle>-0       1dN..4..   22us : sub_preempt_count <-try_to_wake_up
  <idle>-0       1dN..3..   22us : ttwu_stat <-try_to_wake_up
  <idle>-0       1dN..3..   23us : _raw_spin_unlock_irqrestore <-try_to_wake_up
  <idle>-0       1dN..3..   23us : sub_preempt_count <-_raw_spin_unlock_irqrestore

[...]

  <idle>-0       1dN..1..  376us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       1dN..1..  376us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       1dN..1..  376us : account_idle_time <-irqtime_account_process_tick.isra.2
  <idle>-0       1dN..1..  377us : irqtime_account_process_tick.isra.2 <-account_idle_ticks
  <idle>-0       1dN..1..  377us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       1dN..1..  377us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       1dN..1..  377us : account_idle_time <-irqtime_account_process_tick.isra.2
  <idle>-0       1.N..1..  378us : cpu_idle <-start_secondary
  <idle>-0       1.N..1..  378us+: trace_preempt_on <-start_secondary
  <idle>-0       1.N..1..  391us : <stack trace>
 => sub_preempt_count
 => cpu_idle
 => start_secondary
```

The above is a much more interesting trace. Although we enabled function tracing again, it allows us to see more of what is happening during the trace.

The trace starts out at intel_idle() which on the box the trace was run on is the idle function. Idle function usually disable preemption and sometimes interrupts when the system is put to sleep, although an interrupt will wake up the processor, the interrupt will not be serviced until the processor re-enables interrupts again.

As interrupts and preemption is disabled across a full idle, the tracer must account for this, as it is pretty useless to trace how long the CPU has been idle. Thus, immediately exiting the idle state, the latency tracers are re-enabled. This is where the start of the trace occurred.

Then we can see that an interrupt is triggered after interrupts were enabled (schedule_ipi). An interprocessor interrupt happened to wake up a process that is on the current CPU.

Next the irq_enter() is called. This tells the system (including the tracing system) that the kernel is now int interrupt mode. Notice that 'h' is not set until after "add_preempt_count" is called. That's because the irq accounting is shared with the preempt_count code. A lot has happened before that got set, as NO HZ and RCU must perform activities immediately when coming out of idle via an interrupt.

A softirq was raised while in the interrupt and as the Red Hat Enterprise Linux for Real Time kernel runs soft interrupts as threads, the corresponding softirq was woken up on exiting the interrupt (irq_exit).

This wakeup also triggered the NEED_RESCHED flag "N" to be set, to let the system know that the kernel needs to call schedule as soon as preemption is re-enabled.

Finally the NO HZ accounting ran again with interrupts and preemption disabled. Finally, interrupts were enabled and so was the preemption.

wakeup tracer

The previous tracers ("irqsoff", "preemptoff", and "preemptirqsoff") were single CPU tracers. That is, they only reported the activities on a single CPU, as interrupts only occurred there.

Both "wakeup" and "wakeup_rt" tracers are full CPU tracers. That is, they report the activities of what happens across all CPUs. This is because a task may be woken from one CPU but get scheduled on another CPU.

The "wakeup" tracer is not that interresting from a real-time perspective, as it records the time it takes to wake up the highest priority task in the system even if that task does not happen to be a real time task. Non real-time tasks may be delayed due scheduling balacing, and not immediately scheduled for throughput reasons. Real-time tasks are scheduled immediately after they are woken. Recording the max time it takes to wake up a non real-time task will hide the times it takes to wake up a real-time task. Because of this, we will focus on the "wakeup_rt" tracer instead.

wakeup_rt tracer

The "wakeup" tracer records the time it takes from the current highest priority task to wake up to the time it is scheduled. Because non real-time tasks may take much longer to wake up than a real-time task, and that the latency tracers only record the longest time, "wakeup" tracer is not that suitable for seeing how long a real-time task takes to be scheduled from the time it is woken. For that, we use the "wakeup_rt" tracer.

The "wakeup_rt" tracer only records the time for real-time tasks and ignores the time for non real-time tasks.

```
>># echo 0 > tracing_max_latency
>># echo preemptirqsoff > current_tracer
>># sleep 10
>># cat trace
# tracer: wakeup_rt
#
# wakeup_rt latency trace v1.1.5 on 3.8.13-test-mrg-rt9+
# --------------------------------------------------------------------
# latency: 385 us, #1339/1339, CPU#7 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:8)
#    -----------------
#    | task: ksoftirqd/7-51 (uid:0 nice:0 policy:1 rt_prio:1)
#    -----------------
#
#                  _--------=> CPU#
#                 / _-------=> irqs-off
#                | / _------=> need-resched
#                || / _-----=> need-resched_lazy
#                ||| / _----=> hardirq/softirq
#                |||| / _---=> preempt-depth
#                ||||| / _--=> preempt-lazy-depth
#                |||||| / _-=> migrate-disable
#                ||||||| /     delay
#  cmd     pid   |||||||| time  |   caller
#     \   /      ||||||||  \   |   /
  <idle>-0       7d...5..    0us :      0:120:R   + [007]    51: 98:R ksoftirqd/7
  <idle>-0       7d...5..    2us : ttwu_do_activate.constprop.90 <-try_to_wake_up
  <idle>-0       7d...4..    2us : check_preempt_curr <-ttwu_do_wakeup
  <idle>-0       7d...4..    3us : resched_task <-check_preempt_curr
  <idle>-0       7dN..4..    3us : task_woken_rt <-ttwu_do_wakeup
  <idle>-0       7dN..4..    4us : sub_preempt_count <-try_to_wake_up
  <idle>-0       7dN..3..    4us : ttwu_stat <-try_to_wake_up
  <idle>-0       7dN..3..    4us : _raw_spin_unlock_irqrestore <-try_to_wake_up
  <idle>-0       7dN..3..    5us : sub_preempt_count <-_raw_spin_unlock_irqrestore
  <idle>-0       7dN..2..    5us : idle_cpu <-irq_exit
  <idle>-0       7dN..2..    5us : rcu_irq_exit <-irq_exit
  <idle>-0       7dN..2..    6us : rcu_eqs_enter_common.isra.47 <-rcu_irq_exit

[...]

  <idle>-0       7dN..1..   53us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   53us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   54us : account_idle_time <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   54us : irqtime_account_process_tick.isra.2 <-account_idle_ticks
  <idle>-0       7dN..1..   54us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   54us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   55us : account_idle_time <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   55us : irqtime_account_process_tick.isra.2 <-account_idle_ticks
  <idle>-0       7dN..1..   55us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   55us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   56us : account_idle_time <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   56us : irqtime_account_process_tick.isra.2 <-account_idle_ticks
  <idle>-0       7dN..1..   56us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   56us : nsecs_to_jiffies64 <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   57us : account_idle_time <-irqtime_account_process_tick.isra.2
  <idle>-0       7dN..1..   57us : irqtime_account_process_tick.isra.2 <-account_idle_ticks

[...]

  <idle>-0       7dN.h1..  377us : tick_program_event <-hrtimer_interrupt
  <idle>-0       7dN.h1..  378us : clockevents_program_event <-tick_program_event
  <idle>-0       7dN.h1..  378us : ktime_get <-clockevents_program_event
  <idle>-0       7dN.h1..  378us : lapic_next_deadline <-clockevents_program_event
  <idle>-0       7dN.h1..  379us : irq_exit <-smp_apic_timer_interrupt
  <idle>-0       7dN.h1..  379us : irqtime_account_irq <-irq_exit
  <idle>-0       7dN.h1..  379us : sub_preempt_count <-irq_exit
  <idle>-0       7dN..2..  379us : wakeup_softirqd <-irq_exit
  <idle>-0       7dN..2..  380us : idle_cpu <-irq_exit
  <idle>-0       7dN..2..  380us : rcu_irq_exit <-irq_exit
  <idle>-0       7dN..2..  380us : sub_preempt_count <-irq_exit
  <idle>-0       7.N..1..  381us : sub_preempt_count <-cpu_idle
  <idle>-0       7.N.....  381us : __schedule <-preempt_schedule
  <idle>-0       7.N.....  382us : add_preempt_count <-__schedule
  <idle>-0       7.N..1..  382us : rcu_note_context_switch <-__schedule
  <idle>-0       7.N..1..  382us : _raw_spin_lock_irq <-__schedule
  <idle>-0       7dN..1..  382us : add_preempt_count <-_raw_spin_lock_irq
  <idle>-0       7dN..2..  383us : update_rq_clock <-__schedule
  <idle>-0       7dN..2..  383us : put_prev_task_idle <-__schedule
  <idle>-0       7dN..2..  383us : pick_next_task_stop <-__schedule
  <idle>-0       7dN..2..  384us : pick_next_task_rt <-__schedule
  <idle>-0       7dN..2..  384us : dequeue_pushable_task <-pick_next_task_rt
  <idle>-0       7d...3..  385us : __schedule <-preempt_schedule
  <idle>-0       7d...3..  385us :      0:120:R ==> [007]    51: 98:R ksoftirqd/7
```

And once again we can see that NO HZ affects the wake up time of a real time task (this case it was ksoftirqd).

Notice the first traced item:

```
       0:120:R   + [007]    51: 98:R ksoftirqd/7
```

This is in the format of:

```
  <pid>:<prio>:<process-state>    + [<CPU#>]   <pid>:<prio>:<process-state>
```

The first pid, prio and process-state is for the task performing the wake up. Again, the prio is the internal kernel prio, where 120 is for SCHED_OTHER. The "+" represents a wake up is happening. The CPU# the CPU waking task in currently assigned to (and being woken up on).
The second set of pid, prio and process-state is for the task being woken up. The prio of 98 is internal to the kernel, and to get the real real-time priority for the task you must subtract it from 99.
(99 - 98 = real-time priority of 1 - low priority)

The process-state should be always in the "R" (running) state, and can be ignored. The original location to record the trace when waking up was before the task was actually woken. Due to changes in the wake up code, the trace hook had to be moved to after the wake up, which means the task being woken up will have already been set to running and the trace will reflect that.

The last line of the trace:

```
      0:120:R ==> [007]    51: 98:R ksoftirqd/7
```

Represents the scheduling of a task.

```
  <pid>:<prio>:<process-state> ==> [CPU#] <pid>:<prio><process-state>
```

The first set of pid, prio and process-state belongs to the task that is being scheduled out. The second set is for the task that is being scheduled in. The "==>" represents a task scheduling switch, and the CPU# should always match the current CPU that is on (7 in this case).

The first process-state here is of more importance than that of the wake up trace. If the previous task is in the running state (as it is in this case), that means it has been preempted (still wants to run but must yield for the new task).

Using events in tracers

With the "wakeup_rt" tracer, as with all tracers, function tracing can exaggerate the latency times. But disabling the function tracing for "wakeup_rt" is not very useful.

```
>># echo 0 > /sys/kernel/debug/tracing/options/function-trace
>># echo 0 > tracing_max_latency
>># echo wakeup_rt > current_tracer
>># sleep 10
>># cat trace
# tracer: wakeup_rt
#
# wakeup_rt latency trace v1.1.5 on 3.8.13-test-mrg-rt9+
# --------------------------------------------------------------------
# latency: 64 us, #18446744073709512109/18446744073709512109, CPU#5 | (M:preempt VP:0,
KP:0, SP:0 HP:0 #P:8)
#    -----------------
#    | task: irq/43-em1-878 (uid:0 nice:0 policy:1 rt_prio:50)
#    -----------------
#
#                  _--------=> CPU#
#                 / _-------=> irqs-off
#                | / _------=> need-resched
#                || / _-----=> need-resched_lazy
#                ||| / _----=> hardirq/softirq
#                |||| / _---=> preempt-depth
#                ||||| / _--=> preempt-lazy-depth
#                |||||| / _-=> migrate-disable
#                ||||||| /     delay
#  cmd     pid   |||||||| time  |   caller
#     \   /      ||||||||  \   |   /
  <idle>-0       0d..h4..    0us :      0:120:R   + [005]   878: 49:R irq/43-em1
  <idle>-0       0d..h4..    2us+: ttwu_do_activate.constprop.90 <-try_to_wake_up
  <idle>-0       5d...3..   63us : __schedule <-preempt_schedule
  <idle>-0       5d...3..   64us :      0:120:R ==> [005]   878: 49:R irq/43-em1

The irq thread was woken up by a task on CPU 0, and it scheduled on
CPU 5.


As function tracing causes a large overhead, with the wakeup tracers, you
can still get information by using events, and events are sparse enough
to not cause much overhead even when enabled.

>># echo 0 > /sys/kernel/debug/tracing/options/function-trace
>># echo 1 > events/enable
>># echo 0 > tracing_max_latency
>># echo wakeup_rt > current_tracer
>># sleep 10
>># cat trace
# tracer: wakeup_rt
#
# wakeup_rt latency trace v1.1.5 on 3.8.13-test-mrg-rt9+
# --------------------------------------------------------------------
# latency: 67 us, #15/15, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:8)
#    -----------------
#    | task: irq/43-em1-878 (uid:0 nice:0 policy:1 rt_prio:50)
#    -----------------
#
#                  _--------=> CPU#
#                  / _-------=> irqs-off
#                 | / _------=> need-resched
#                 || / _-----=> need-resched_lazy
#                 ||| / _----=> hardirq/softirq
#                 |||| / _---=> preempt-depth
#                 ||||| / _--=> preempt-lazy-depth
#                 |||||| / _-=> migrate-disable
#                 ||||||| /     delay
#  cmd     pid    |||||||| time  |   caller
#     \   /      ||||||||  \   |   /
  <idle>-0       0d..h4..    0us :      0:120:R   + [001]   878: 49:R irq/43-em1
  <idle>-0       0d..h4..    1us : ttwu_do_activate.constprop.90 <-try_to_wake_up
  <idle>-0       0d..h4..    1us+: sched_wakeup: comm=irq/43-em1 pid=878 prio=49 success=1
target_cpu=001
  <idle>-0       0....2..    5us : power_end: cpu_id=0
  <idle>-0       0....2..    6us+: cpu_idle: state=4294967295 cpu_id=0
  <idle>-0       0d...2..    9us : power_start: type=1 state=3 cpu_id=0
  <idle>-0       0d...2..   10us+: cpu_idle: state=3 cpu_id=0
  <idle>-0       1.N..2..   25us+: power_end: cpu_id=1
  <idle>-0       1.N..2..   27us+: cpu_idle: state=4294967295 cpu_id=1
  <idle>-0       1dN..3..   30us : hrtimer_cancel: hrtimer=ffff88011ea4cf40
  <idle>-0       1dN..3..   31us+: hrtimer_start: hrtimer=ffff88011ea4cf40 function=tick_sched_timer
expires=9670689000000 softexpires=9670689000000
  <idle>-0       1.N..2..   64us : rcu_utilization: Start context switch
  <idle>-0       1.N..2..   65us+: rcu_utilization: End context switch
  <idle>-0       1d...3..   66us : __schedule <-preempt_schedule
  <idle>-0       1d...3..   67us :      0:120:R ==> [001]   878: 49:R irq/43-em1
```

The above trace is much more accurate to a real latency, but this time
we get a lot more information. The task being woken up in on CPU 1, and
the first time we see CPU 1 is at the 25 microsecond time. The "power_end"
trace point shows that the CPU is coming out of a deep power state, which
explains why the time took so long. The high resolution timer has been
reinitialized, and we can assume from our other traces that the NO HZ
code is running again to catch up on the tick, although no trace points
currently represent that. This process took 33 microseconds, where we
see RCU handling a context switch, and eventually the schedule takes place.

function_graph

The "function" tracer is extremely informative, albeit invasive, but
it is a bit difficult for a human to read.

```
          <idle>-0     [000] ....1.. 10698.878897: sub_preempt_count <-__schedule
            less-3062  [006] ....... 10698.878897: add_preempt_count <-migrate_disable
             cat-3061  [007] d...... 10698.878897: add_preempt_count <-_raw_spin_lock
          <idle>-0     [000] ....... 10698.878897: add_preempt_count <-cpu_idle
            less-3062  [006] ....11. 10698.878897: pin_current_cpu <-migrate_disable
          <idle>-0     [000] ....1.. 10698.878898: tick_nohz_idle_enter <-cpu_idle
             cat-3061  [007] d...1.. 10698.878898: sub_preempt_count <-__raw_spin_unlock
            less-3062  [006] ....111 10698.878898: sub_preempt_count <-migrate_disable
          <idle>-0     [000] ....1.. 10698.878898: set_cpu_sd_state_idle <-tick_nohz_idle_enter
```

```
             cat-3061  [007] ....... 10698.878898: free_delayed <-__slab_alloc.isra.60
            less-3062  [006] .....11 10698.878898: migrate_disable <-get_page_from_freelist
            less-3062  [006] .....11 10698.878898: add_preempt_count <-migrate_disable
          <idle>-0     [000] d...1.. 10698.878898: __tick_nohz_idle_enter <-tick_nohz_idle_enter
            less-3062  [006] ....112 10698.878898: sub_preempt_count <-migrate_disable
          <idle>-0     [000] d...1.. 10698.878898: ktime_get <-__tick_nohz_idle_enter
             cat-3061  [007] ....... 10698.878898: __rt_mutex_init <-tracing_open


The "function_graph" tracer is a bit more easy on the eyes, and lets
the developer follow the code in much more detail.

># echo function_graph > current_tracer
># cat trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 5)   0.125 us    |              source_load();
 5)   0.137 us    |              idle_cpu();
 5)   0.105 us    |              source_load();
 5)   0.110 us    |              idle_cpu();
 5)   0.132 us    |              source_load();
 5)   0.134 us    |              idle_cpu();
 5)   0.127 us    |              source_load();
 5)   0.144 us    |              idle_cpu();
 5)   0.132 us    |              source_load();
 5)   0.112 us    |              idle_cpu();
 5)   0.120 us    |              source_load();
 5)   0.130 us    |              idle_cpu();
 5) + 20.812 us   |            } /* find_busiest_group */
 5) + 21.905 us   |          } /* load_balance */
 5)   0.099 us    |        msecs_to_jiffies();
 5)   0.120 us    |        __rcu_read_unlock();
 5)               |        _raw_spin_lock() {
 5)   0.115 us    |          add_preempt_count();
 5)   1.115 us    |        }
 5) + 46.645 us   |      } /* idle_balance */
 5)               |      put_prev_task_rt() {
 5)               |        update_curr_rt() {
 5)               |          cpuacct_charge() {
 5)   0.110 us    |            __rcu_read_lock();
 5)   0.110 us    |            __rcu_read_unlock();
 5)   2.111 us    |          }
 5)   0.100 us    |          sched_avg_update();
 5)               |          _raw_spin_lock() {
 5)   0.116 us    |            add_preempt_count();
 5)   1.151 us    |          }
 5)   0.122 us    |          balance_runtime();
 5)   0.110 us    |          sub_preempt_count();
 5)   8.165 us    |        }
 5)   9.152 us    |      }
 5)   0.148 us    |      pick_next_task_fair();
 5)   0.112 us    |      pick_next_task_stop();
 5)   0.117 us    |      pick_next_task_rt();
 5)   0.123 us    |      pick_next_task_fair();
 5)   0.138 us    |      pick_next_task_idle();
 ------------------------------------------
 5)   ksoftir-39   =>    <idle>-0
 ------------------------------------------

 5)               |      finish_task_switch() {
 5)               |        _raw_spin_unlock_irq() {
 5)   0.260 us    |          sub_preempt_count();
 5)   1.289 us    |        }
 5)   2.309 us    |      }
 5)   0.132 us    |      sub_preempt_count();
 5) ! 151.784 us  |    } /* __schedule */
 5)   0.272 us    |  } /* sub_preempt_count */
```

The "function" tracer only traces the start of the function where as the "function_graph" tracer also traces the exit of the function, allowing to show a flow of function calls in the kernel. As one function calls the next function, it is indented in the trace and C code curly brackets are placed around them. When there's a leaf function (a function that does not call any other function, or any function that happens to be traced), it is simply finished with a ";".

This tracer has a different format than the other tracers, to help ease the reading of the trace. The first number "5)" represents the CPU that the trace happened on. The second number is the time the function took to execute. Note, this time also include the overhead of the "function_graph" tracer itself, so for functions that have several other functions traced within it, its time will be rather exaggerated. For leaf functions, the time is rather accurate.

When a schedule switch is detected (does not require the sched_switch event enabled, as all traces record the pid), it shows up as separately displayed.

```
 ------------------------------------------
 5)   ksoftir-39   =>    <idle>-0
 ------------------------------------------
```

The name is cropped to 7 characters (from "ksoftirqd" to "ksoftir").

Follow a function

Because the "function_graph" tracer records both the start and exit of a function, several more features are possible. One of these features is to graph only a specific function. That is, to see what a specific function calls and ignore all other functions.

For example, if you are interested in what the sys_read() function calls, you can use the "set_graph_function" file in the tracing debug file system.

```
># echo sys_read > set_graph_function
># echo function_graph > current_tracer
># sleep 10
># cat trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0)               |  sys_read() {
 0)   0.126 us    |    fget_light();
 0)               |    vfs_read() {
 0)               |      rw_verify_area() {
 0)               |        security_file_permission() {
 0)   0.077 us    |          cap_file_permission();
 0)   0.076 us    |          __fsnotify_parent();
 0)   0.100 us    |          fsnotify();
 0)   2.001 us    |        }
 0)   2.608 us    |      }
 0)               |      tty_read() {
 0)   0.070 us    |        tty_paranoia_check();
 0)               |        tty_ldisc_ref_wait() {
 0)               |          tty_ldisc_try() {
 0)               |            _raw_spin_lock_irqsave() {
 0)   0.130 us    |              add_preempt_count();
 0)   0.759 us    |            }
 0)               |            _raw_spin_unlock_irqrestore() {
 0)   0.132 us    |              sub_preempt_count();
 0)   0.774 us    |            }
 0)   2.576 us    |          }
 0)   3.161 us    |        }
 0)               |        n_tty_read() {
 0)               |          _mutex_lock_interruptible() {
 0)   0.087 us    |            rt_mutex_lock_interruptible();
 0)   0.694 us    |          }
 0)               |          add_wait_queue() {
 0)               |            migrate_disable() {
 0)   0.100 us    |              add_preempt_count();
 0)   0.073 us    |              pin_current_cpu();
 0)   0.085 us    |              sub_preempt_count();
 0)   1.829 us    |            }
 0)   0.060 us    |            rt_spin_lock();
 0)   0.065 us    |            rt_spin_unlock();
 0)               |            migrate_enable() {
 0)   0.077 us    |              add_preempt_count();
 0)   0.070 us    |              unpin_current_cpu();
 0)   0.077 us    |              sub_preempt_count();
 0)   1.847 us    |            }
 0)   5.899 us    |          }
```

The above shows the flow of functions called by sys_read().

To reset the "set_graph_function" simply write into that file like the "set_ftrace_filter" file is done.

```
># echo > set_graph_function
```

```
Time a function
---------------

As the "function_graph" tracer is associated to the "function" tracer
it is also affected by the "set_ftrace_filter", "set_ftrace_notrace"
as well as the sysctl feature "kernel.ftrace_enabled".

As mentioned previously, only the leaf functions contain the most accurate
times of execution. By filtering on a specific function, you can see
the time it takes to execute a single function.


>#  echo do_IRQ > set_ftrace_filter
>#  echo function_graph > current_tracer
>#  sleep 10
>#  cat trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 4)   ==========> |
 4)   6.486 us    |  do_IRQ();
 0)   ==========> |
 0)   3.801 us    |  do_IRQ();
 4)   ==========> |
 4)   3.221 us    |  do_IRQ();
 0)   ==========> |
 0) + 11.153 us   |  do_IRQ();
 0)   ==========> |
 0) + 10.968 us   |  do_IRQ();
 6)   ==========> |
 6)   9.280 us    |  do_IRQ();
 0)   ==========> |
 0)   9.467 us    |  do_IRQ();
 0)   ==========> |
 0) + 11.238 us   |  do_IRQ();


The "==========>" show when an interrupt entered. The "<==========" is
missing because it is associated with the exit part of the trace.
As "do_IRQ" is a leaf function here, the exit arrow was folded into
the function and does not appear in the trace.


Events in function graph tracer
-------------------------------

As explained previously, events can be enabled with all tracers.
But with the "function_graph" tracer, they are displayed a little
differently.

>#  echo 1 > events/irq/enable
>#  echo do_IRQ > set_ftrace_filter
>#  echo function_graph > current_tracer
>#  sleep 10
>#  cat trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 5)   ==========> |
 5)               |  do_IRQ() {
 5)               |  /* irq_handler_entry: irq=43 name=em1 */
 5)               |  /* irq_handler_exit: irq=43 ret=handled */
 5) + 15.721 us   |  }
 5)   <========== |
 3)               |  /* softirq_raise: vec=3 [action=NET_RX] */
 3)               |  /* softirq_entry: vec=3 [action=NET_RX] */
 3)               |  /* softirq_exit: vec=3 [action=NET_RX] */
 0)   ==========> |
 0)               |  do_IRQ() {
 0)               |  /* irq_handler_entry: irq=43 name=em1 */
 0)               |  /* irq_handler_exit: irq=43 ret=handled */
 0)   8.915 us    |  }
 0)   <========== |
 3)               |  /* softirq_raise: vec=3 [action=NET_RX] */
 3)               |  /* softirq_entry: vec=3 [action=NET_RX] */
 3)               |  /* softirq_exit: vec=3 [action=NET_RX] */
 0)               |  /* softirq_raise: vec=1 [action=TIMER] */
 0)               |  /* softirq_raise: vec=9 [action=RCU] */
 ------------------------------------------
 0)    <idle>-0    =>   ksoftir-3
 ------------------------------------------

 0)               |  /* softirq_entry: vec=1 [action=TIMER] */
 0)               |  /* softirq_exit: vec=1 [action=TIMER] */
 0)               |  /* softirq_entry: vec=9 [action=RCU] */
 0)               |  /* softirq_exit: vec=9 [action=RCU] */
 ------------------------------------------
 0)   ksoftir-3    =>    <idle>-0
 ------------------------------------------

Keeping with the C formatting, events in the "function_graph" tracer
appear as comments. Recording the interrupt events gives more detail
to what interrupts are occurring when "do_IRQ()" is called. As the
"do_IRQ()" exit trace is not folded, the "<==========" appears to
display that the interrupt is over.


Annotations
-----------

In the traces, including the "function_graph" tracer, you may see
annotations around the times. "+" and "!". A "+" appears when the
time between events is greater than 10 microseconds, and a "!" appears
when that time is greater than 100 microseconds. You can see this in the
above tracers:

  <idle>-0       0d..h4..    2us+: ttwu_do_activate.constprop.90 <-try_to_wake_up
  <idle>-0       5d...3..   63us : __schedule <-preempt_schedule
 5) + 20.812 us   |          } /* find_busiest_group */
 5) + 21.905 us   |        } /* load_balance */

 5) ! 151.784 us  |    } /* __schedule */



Buffer size
-----------

When tracing functions, you will almost always use events. This is because
the amount of functions being traced will quickly fill the ring buffer
faster than anything can read from it. The amount lost can be minimized
with filtering the trace as well as increasing the size of the buffer.

The size of the buffer is controlled by the "buffer_size_kb" file.
As the name suggests, the size is in kilobytes. When you first boot up,
as tracing is used by only a small minority of users, the trace buffer
is compressed. The first time you use any of the tracing features,
the tracing buffer will automatically increase to a decent size.

>
# cat buffer_size_kb
7 (expanded: 1408)

Note, for efficiency reasons, the buffer is split into multiple buffers
per CPU. The size displayed by "buffer_size_kb" is the size of each
CPU buffer. To see the total size of all buffers look at
"buffer_total_size_kb"

>
# cat buffer_total_size_kb
56 (expanded: 11264)

After running any trace, the buffer will expand to the size that is
denoted by the "expanded" value.

>
# echo 1 > events/enable
>
# cat buffer_size_kb
1408

To change the size of the buffer, simply echo in a number.

>
# echo 10000 > buffer_size_kb
>
# cat buffer_size_kb
10000

Note, if you change the size before using any tracer, the buffers
will go to that size, and the expanded value will then be ignored.


Buffer size per CPU
-------------------

If there's a case you care about activity on one CPU more than another
CPU, and you need to save memory, you can change the sizes of the
ring buffers per CPU. These files exist in a "per_cpu/cpuX/" directory.
```

```
>># cat per_cpu/cpu1/buffer_size_kb
10000

>># echo 100 > per_cpu/cpu1/buffer_size_kb
>># cat per_cpu/cpu1/buffer_size_kb
100

When the per CPU buffers differ in size, the top level buffer_size_kb
will display an "X".

>># cat buffer_size_kb
X

But the total size will still display the amount allocated.

>># cat buffer_total_size_kb
70100


Trace Marker
------------

It is sometimes useful to synchronize actions in userspace with events
within the kernel. The "trace_marker" allows userspace to write into
the ftrace buffer.

>># echo hello world > trace_marker
>># cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 1/1   #P:8
#
#                              _-------=> irqs-off
#                             /  _------=> need-resched
#                             |/  _-----=> need-resched_lazy
#                             ||/  _----=> hardirq/softirq
#                             |||/  _---=> preempt-depth
#                             ||||/  _--=> preempt-lazy-depth
#                             ||||| / _-=> migrate-disable
#                             |||||| /     delay
#           TASK-PID   CPU#  |||||||    TIMESTAMP  FUNCTION
#              | |       |   |||||||       |         |
            bash-1086  [001] .....11 21351.346541: tracing_mark_write: hello world

Writing into the kernel is very light weight. User programs can take
advantage of this with the following C code:

        static int trace_fd = -1;

        void trace_write(const char *fmt, ...)
        {
                va_list ap;
                char buf[256];
                int n;
                if (trace_fd < 0)
                        return;

                va_start(ap, fmt);
                n = vsnprintf(buf, 256, fmt, ap);
                va_end(ap);

                write(trace_fd, buf, n);
        }


        [...]

        trace_fd = open("trace_marker", WR_ONLY);
```

and later use the "trace_write()" function to record into the ftrace
buffer.

```
        trace_write("record this event\n");
```

tracer options

There are several options that can affect the formating of the trace
output as well as how the tracers behave. Some trace options only exist
for a given tracer and their control file appears only when the tracer
is activated.

The trace option control files exist in the "options" directory.

```
># ls options
annotate        graph-time      print-parent  sym-userobj
bin             hex             raw           test_nop_accept
block           irq-info        record-cmd    test_nop_refuse
branch          latency-format  sleep-time    trace_printk
context-info    markers         stacktrace    userstacktrace
disable_on_free overwrite       sym-addr      verbose
ftrace_preempt  printk-msg-only sym-offset
```

The "function_graph" tracer adds several of its own.

```
># echo function_graph > current_tracer
># ls options
annotate          funcgraph-cpu      irq-info        sleep-time
bin               funcgraph-duration latency-format  stacktrace
block             funcgraph-irqs     markers         sym-addr
branch            funcgraph-overhead overwrite       sym-offset
context-info      funcgraph-overrun  printk-msg-only sym-userobj
disable_on_free   funcgraph-proc     print-parent    trace_printk
ftrace_preempt    graph-time         raw             userstacktrace
funcgraph-abstime hex                record-cmd      verbose
```

```
  annotate - It is sometimes confusing when the CPU buffers are full
             and one CPU buffer had a lot of events recently, thus
           a shorter time frame, were another CPU may have only had
           a few events, which lets it have older events. When
           the trace is reported, it shows the oldest events first,
           and it may look like only one CPU ran (the one with the
           oldest events). When the annotate option is set, it will
           display when a new CPU buffer started:

          <idle>-0     [005] d...1..   910.328077: cpuidle_wrap_enter <-cpuidle_enter_tk
          <idle>-0     [005] d...1..   910.328077: ktime_get <-cpuidle_wrap_enter
          <idle>-0     [005] d...1..   910.328078: intel_idle <-cpuidle_enter
          <idle>-0     [005] d...1..   910.328078: leave_mm <-intel_idle
##### CPU 7 buffer started ####
          <idle>-0     [007] d...1..   910.360866: tick_do_update_jiffies64 <-tick_check_idle
          <idle>-0     [007] d...1..   910.360866: _raw_spin_lock <-tick_do_update_jiffies64
          <idle>-0     [007] d...1..   910.360866: add_preempt_count <-_raw_spin_lock


  bin - This will print out the formats in raw binary.

  block - When set, reading trace_pipe will not block when polled.

  context-info - Show only the event data. Hides the comm, PID,
                 timestamp, CPU, and other useful data.

  disable_on_free - When the free_buffer is closed, tracing will
                    stop (tracing_on set to 0).

  ftrace_preempt - Normally the function tracer disables interrupts as
                   the recursion protection will hide interrupts from being
                 traced if the interrupt happened while another function
                 was being traced. If this option is enabled, then it
                 will not disable interrupts but will only disable
                 preemption. But note, if an interrupt were to arrive
                 when another function is being traced, all functions
                 within that interrupt will not be traced, as function
                 tracing is temporarily disablde for recursion protection.

  graph-time - When running function graph tracer, to include the
               time to call nested functions. When this is not set,
             the time reported for the function will only include
             the time the function itself executed for, not the time
             for functions that it called.

  hex - Similar to raw, but the numbers will be in a hexadecimal
        format.

  irq-info - Shows the interrupt, preempt count, need resched data.
             When disabled, the trace looks like:

# tracer: function
#
# entries-in-buffer/entries-written: 319494/4972382   #P:8
#
#           TASK-PID   CPU#      TIMESTAMP  FUNCTION
#              | |       |          |         |
          <idle>-0     [004]    983.062800: lock_hrtimer_base.isra.25 <-__hrtimer_start_range_ns
          <idle>-0     [004]    983.062801: _raw_spin_lock_irqsave <-lock_hrtimer_base.isra.25
          <idle>-0     [004]    983.062801: add_preempt_count <-_raw_spin_lock_irqsave
          <idle>-0     [004]    983.062801: __remove_hrtimer <-__hrtimer_start_range_ns
          <idle>-0     [004]    983.062801: hrtimer_force_reprogram <-__remove_hrtimer


  latency-format - This option changes the trace. When
                 it is enabled, the trace displays
                 additional information about the
                 latencies, as described in "Latency
                 trace format".

  markers - When set, the trace_marker is writable (only by root).
            When disabled, the trace_marker will error with EINVAL
          on write.

  overwrite - This controls what happens when the trace buffer is
            full. If "1" (default), the oldest events are
            discarded and overwritten. If "0", then the newest
            events are discarded.
              (see per_cpu/cpu0/stats for overrun and dropped)

  printk-msg-only - When set, trace_printk()s will only show the format
                    and not their parameters (if trace_bprintk() or
                  trace_bputs() was used to save the trace_printk()).

  print-parent - On function traces, display the calling (parent)
               function as well as the function being traced.

    print-parent:
       bash-1423  [006]   1755.774709: msecs_to_jiffies <-idle_balance

    noprint-parent:
       bash-1423  [006]   1755.774709: msecs_to_jiffies


  raw - This will display raw numbers. This option is best for
        use with user applications that can translate the raw
        numbers better than having it done in the kernel.

  record-cmd - When any event or tracer is enabled, a hook is enabled
               in the sched_switch trace point to fill comm cache
             with mapped pids and comms. But this may cause some
             overhead, and if you only care about pids, and not the
             name of the task, disabling this option can lower the
             impact of tracing.

  sleep-time - When running function graph tracer, to include
               the time a task schedules out in its function.
             When enabled, it will account time the task has been
             scheduled out as part of the function call.

  stacktrace - This is one of the options that changes the trace
             itself. When a trace is recorded, so is the stack
             of functions. This allows for back traces of
             trace sites.

  sym-addr - this will also display the function address as well
           as the function name.

  sym-offset - Display not only the function name, but also the
             offset in the function. For example, instead of
             seeing just "ktime_get", you will see
             "ktime_get+0xb/0x20".

   sym-offset:
    bash-1423  [006]   1755.774709: msecs_to_jiffies+0x0/0x20

   sym-addr:
    bash-1423  [006]   1755.774709: msecs_to_jiffies <ffffffff8106b5f0>


  sym-userobj - when user stacktrace are enabled, look up which
              object the address belongs to, and print a
              relative address. This is especially useful when
              ASLR is on, otherwise you don't get a chance to
              resolve the address to object/file/line after
              the app is no longer running

              The lookup is performed when you read
              trace,trace_pipe. Example:

        a.out-1623  [000] 40874.465068: /root/a.out[+0x480] <-/root/a.out[+0x494] <- /root/a.out[+0x4a8]
<- /lib/libc-2.7.so[+0x1e1a6]

  trace_printk - Can disable trace_printk() from writing into the buffer.

  userstacktrace - This option changes the trace. It records a
                 stacktrace of the current userspace thread at each event.

  verbose - This deals with the trace file when the
          latency-format option is enabled.

   bash  4000 1 0 00000000 00010a95 [58127d26] 1720.415ms \
   (+0.000ms): simple_strtoul (strict_strtoul)




This has been quite an in depth look at how to use ftrace via the
debug file system. But it can be quite daunting to handle all these
different files. Luckily, there's a tool that can do most of this
work for you. It's called "trace-cmd".
```

```
Using trace-cmd
---------------

trace-cmd is a tool that interacts with the ftrace tracing facility.
It reads and writes to the same files that are described above as
well as reading the files that can transfer the binary data of
```

```
the kernel tracing buffers in an efficient manner to be read later.
The tool is very simple and easy to use.

There are several man pages for trace-cmd. First look at

  man trace-cmd

to find out more information on the other commands. All of trace-cmd's
commands also have their own man pages in the format of:

  man trace-cmd-<command>

For example, the "record" command's man page is under trace-cmd-record.

This document will describe all the options for each command, but
instead will briefly discuss how to use trace-cmd and describe most of
its commands.



trace-cmd record and report
--------------------------

To use ftrace tracers and events you must first have to start tracing
by either echoing a name of a tracer into the "current_tracer" file
or by echoing "1" into one of the event "enable" files.

For trace-cmd, the record option starts the tracing and will also save
the traced data into a file. Let's start with an example:


>\# cd ~
>\# trace-cmd record -p function
  plugin 'function'
Hit Ctrl^C to stop recording
(^C)
Kernel buffer statistics:
  Note: "entries" are the entries left in the kernel ring buffer and are not
        recorded in the trace data. They should all be zero.

CPU: 0
entries: 0
overrun: 38650181
commit overrun: 0
bytes: 3060
oldest event ts: 15634.891771
now ts: 15634.953219
dropped events: 0

CPU: 1
entries: 0
overrun: 38523960
commit overrun: 0
bytes: 1368
oldest event ts: 15634.891771
now ts: 15634.953938
```

```
dropped events: 0

CPU: 2
entries: 0
overrun: 41461508
commit overrun: 0
bytes: 1872
oldest event ts: 15634.891773
now ts: 15634.954630
dropped events: 0

CPU: 3
entries: 0
overrun: 38246206
commit overrun: 0
bytes: 36
oldest event ts: 15634.891785
now ts: 15634.955263
dropped events: 0

CPU: 4
entries: 0
overrun: 32730902
commit overrun: 0
bytes: 432
oldest event ts: 15634.891716
now ts: 15634.955952
dropped events: 0

CPU: 5
entries: 0
overrun: 33264601
commit overrun: 0
bytes: 2952
oldest event ts: 15634.891769
now ts: 15634.956630
dropped events: 0

CPU: 6
entries: 0
overrun: 30974204
commit overrun: 0
bytes: 2484
oldest event ts: 15634.891772
now ts: 15634.957249
dropped events: 0

CPU: 7
entries: 0
overrun: 32374274
commit overrun: 0
bytes: 3564
oldest event ts: 15634.891652
now ts: 15634.957938
dropped events: 0
```

```
CPU0 data recorded at offset=0x302000
    146325504 bytes in size
CPU1 data recorded at offset=0x8e8e000
    148217856 bytes in size
CPU2 data recorded at offset=0x11be8000
    148066304 bytes in size
CPU3 data recorded at offset=0x1a91d000
    146219008 bytes in size
CPU4 data recorded at offset=0x2348f000
    145940480 bytes in size
CPU5 data recorded at offset=0x2bfbd000
    145403904 bytes in size
CPU6 data recorded at offset=0x34a68000
    141570048 bytes in size
CPU7 data recorded at offset=0x3d16b000
    147513344 bytes in size


The "-p" is for ftrace tracers (use to be known as 'plugins' and the name
is kept for historical reasons). In this case we started the
"function" tracer. Since we did not add a command to execute, by
default, trace-cmd will just start the tracing and record the data
and wait for the user to hit Ctrl^C to stop.

When the trace stops, it prints out status of each of the kernel's
per cpu trace buffers. The are:

  entries:  - Which is the number of entries still in the kernel buffer.
              Ideally this should be zero, as trace-cmd would consume them
            all and put them into the data file.

  overrun:  - As tracing can be much faster than the saving of data,
              events can be lost due to overwriting of the old events
            that were not consumed yet when the buffer filled up.
            This is the number of events that were lost.

            The "function" tracer can fill up the buffer extremely fast
            it is not uncommon to lose millions of events when
            tracing functions for any length of time.

  commit overrun: - This should always be zero, and if it is not, then
              the buffer size is way too small or something went wrong
            with the tracer.

  bytes: - The number of bytes consumed (not read as pages). This is
             more a status for developers of the tracing utitily.

  oldest event ts: - The timestamp for the oldest event still in the ring
             buffer. Unless it gets overwritten, it will be the timestamp
           of the next event read.

  now ts: The current timestamp used by the tracing facility.

  dropped events: - If the buffer has overwrite mode disabled (from the
            trace options), then this will show the number of events that
          were lost due to not being able to write to the buffer because
```

```
        it was full. This is similar to the overrun field except that
        those are events that made it into the buffer but were overwritten.
```

By default, the file used to record the trace is called "trace.dat".
You can override the output file with the -o option.

To read the trace.dat file, simply run the trace-cmd report command:

```
>># trace-cmd report
version = 6
cpus=8
     trace-cmd-3735  [003] 15618.722889: function:             __hrtimer_start_range_ns
     trace-cmd-3734  [002] 15618.722889: function:             _mutex_unlock
          <idle>-0   [000] 15618.722889: function:         cpuidle_wrap_enter
     trace-cmd-3735  [003] 15618.722890: function:               lock_hrtimer_base.isra.25
     trace-cmd-3734  [002] 15618.722890: function:               rt_mutex_unlock
          <idle>-0   [000] 15618.722890: function:           ktime_get
     trace-cmd-3735  [003] 15618.722890: function:                 _raw_spin_lock_irqsave
     trace-cmd-3735  [003] 15618.722891: function:                   add_preempt_count
     trace-cmd-3734  [002] 15618.722891: function:             __fsnotify_parent
          <idle>-0   [000] 15618.722891: function:         intel_idle
     trace-cmd-3735  [003] 15618.722891: function:               idle_cpu
     trace-cmd-3734  [002] 15618.722891: function:             fsnotify
          <idle>-0   [000] 15618.722891: function:           leave_mm
     trace-cmd-3735  [003] 15618.722891: function:               ktime_get
     trace-cmd-3734  [002] 15618.722891: function:               __srcu_read_lock
          <idle>-0   [000] 15618.722891: function:             __phys_addr
     trace-cmd-3734  [002] 15618.722891: function:                 add_preempt_count
     trace-cmd-3735  [003] 15618.722891: function:               enqueue_hrtimer
     trace-cmd-3735  [003] 15618.722892: function:               _raw_spin_unlock_irqrestore
     trace-cmd-3734  [002] 15618.722892: function:                 sub_preempt_count
     trace-cmd-3735  [003] 15618.722892: function:                 sub_preempt_count
     trace-cmd-3734  [002] 15618.722892: function:               __srcu_read_unlock
     trace-cmd-3735  [003] 15618.722892: function:             schedule
     trace-cmd-3734  [002] 15618.722892: function:                 add_preempt_count
     trace-cmd-3735  [003] 15618.722893: function:               __schedule
     trace-cmd-3734  [002] 15618.722893: function:                 sub_preempt_count
     trace-cmd-3735  [003] 15618.722893: function:                 add_preempt_count
     trace-cmd-3735  [003] 15618.722893: function:                 rcu_note_context_switch
     trace-cmd-3734  [002] 15618.722893: function:             __audit_syscall_exit
     trace-cmd-3735  [003] 15618.722893: function:                 _raw_spin_lock_irq
     trace-cmd-3735  [003] 15618.722894: function:                   add_preempt_count
     trace-cmd-3734  [002] 15618.722894: function:               path_put
     trace-cmd-3735  [003] 15618.722894: function:                 deactivate_task
     trace-cmd-3734  [002] 15618.722894: function:                 dput
     trace-cmd-3735  [003] 15618.722894: function:                   dequeue_task
     trace-cmd-3734  [002] 15618.722894: function:                 mntput
     trace-cmd-3735  [003] 15618.722894: function:                     update_rq_clock
     trace-cmd-3734  [002] 15618.722894: function:               unroll_tree_refs
```

To filter out a CPU, use the --cpu option.

```
>># trace-cmd report --cpu 1
```

```
version = 6
cpus=8
       <idle>-0     [001] 15618.723287: function:             ktime_get
       <idle>-0     [001] 15618.723288: function:             smp_apic_timer_interrupt
       <idle>-0     [001] 15618.723289: function:                irq_enter
       <idle>-0     [001] 15618.723289: function:                   rcu_irq_enter
       <idle>-0     [001] 15618.723289: function:                      rcu_eqs_exit_common.isra.45
       <idle>-0     [001] 15618.723289: function:                   tick_check_idle
       <idle>-0     [001] 15618.723290: function:                      tick_check_oneshot_broadcast
       <idle>-0     [001] 15618.723290: function:                      ktime_get
       <idle>-0     [001] 15618.723290: function:                      tick_nohz_stop_idle
       <idle>-0     [001] 15618.723290: function:                         update_ts_time_stats
       <idle>-0     [001] 15618.723290: function:                            nr_iowait_cpu
       <idle>-0     [001] 15618.723291: function:             touch_softlockup_watchdog
       <idle>-0     [001] 15618.723291: function:                      tick_do_update_jiffies64
       <idle>-0     [001] 15618.723291: function:                      touch_softlockup_watchdog
       <idle>-0     [001] 15618.723291: function:                   irqtime_account_irq
       <idle>-0     [001] 15618.723292: function:                      in_serving_softirq
       <idle>-0     [001] 15618.723292: function:                   add_preempt_count
       <idle>-0     [001] 15618.723292: function:             exit_idle
       <idle>-0     [001] 15618.723292: function:                atomic_notifier_call_chain
       <idle>-0     [001] 15618.723293: function:                   __atomic_notifier_call_chain
       <idle>-0     [001] 15618.723293: function:                      __rcu_read_lock
```

Notice how the functions are indented similar to the function_graph tracer. This is because trace-cmd can post process the trace data with more complex algorithms than are acceptable to implement in the kernel. It uses the parent function to follow which function is called by other functions and be able to deduce a call graph.

To disable the indentation, use the -O report option.

```
>#  trace-cmd report --cpu 1 -O indent=0
version = 6
cpus=8
       <idle>-0     [001] 15618.723287: function:             ktime_get
       <idle>-0     [001] 15618.723288: function:             smp_apic_timer_interrupt
       <idle>-0     [001] 15618.723289: function:             irq_enter
       <idle>-0     [001] 15618.723289: function:             rcu_irq_enter
       <idle>-0     [001] 15618.723289: function:             rcu_eqs_exit_common.isra.45
       <idle>-0     [001] 15618.723289: function:             tick_check_idle
       <idle>-0     [001] 15618.723290: function:             tick_check_oneshot_broadcast
       <idle>-0     [001] 15618.723290: function:             ktime_get
       <idle>-0     [001] 15618.723290: function:             tick_nohz_stop_idle
       <idle>-0     [001] 15618.723290: function:             update_ts_time_stats
       <idle>-0     [001] 15618.723290: function:             nr_iowait_cpu
       <idle>-0     [001] 15618.723291: function:             touch_softlockup_watchdog
       <idle>-0     [001] 15618.723291: function:             tick_do_update_jiffies64
       <idle>-0     [001] 15618.723291: function:             touch_softlockup_watchdog
```

To add back the parent:

```
>#  trace-cmd report --cpu 1 -O indent=0 -O parent=1
version = 6
```

```
cpus=8
        <idle>-0     [001] 15618.723287: function:             ktime_get <-- cpuidle_wrap_enter
        <idle>-0     [001] 15618.723288: function:             smp_apic_timer_interrupt <--
apic_timer_interrupt
        <idle>-0     [001] 15618.723289: function:             irq_enter <-- smp_apic_timer_interrupt
        <idle>-0     [001] 15618.723289: function:             rcu_irq_enter <-- irq_enter
        <idle>-0     [001] 15618.723289: function:             rcu_eqs_exit_common.isra.45 <--
rcu_irq_enter
        <idle>-0     [001] 15618.723289: function:             tick_check_idle <-- irq_enter
        <idle>-0     [001] 15618.723290: function:             tick_check_oneshot_broadcast <--
tick_check_idle
        <idle>-0     [001] 15618.723290: function:             ktime_get <-- tick_check_idle
        <idle>-0     [001] 15618.723290: function:             tick_nohz_stop_idle <-- tick_check_idle
        <idle>-0     [001] 15618.723290: function:             update_ts_time_stats <-- tick_nohz_stop_idle
        <idle>-0     [001] 15618.723290: function:             nr_iowait_cpu <-- update_ts_time_stats
        <idle>-0     [001] 15618.723291: function:             touch_softlockup_watchdog <--
sched_clock_idle_wakeup_event
        <idle>-0     [001] 15618.723291: function:             tick_do_update_jiffies64 <-- tick_check_idle
        <idle>-0     [001] 15618.723291: function:             touch_softlockup_watchdog <--
tick_check_idle
        <idle>-0     [001] 15618.723291: function:             irqtime_account_irq <-- irq_enter
        <idle>-0     [001] 15618.723292: function:             in_serving_softirq <-- irqtime_account_irq
        <idle>-0     [001] 15618.723292: function:             add_preempt_count <-- irq_enter
        <idle>-0     [001] 15618.723292: function:             exit_idle <-- smp_apic_timer_interrupt
        <idle>-0     [001] 15618.723292: function:             atomic_notifier_call_chain <-- exit_idle
        <idle>-0     [001] 15618.723293: function:             __atomic_notifier_call_chain <--
atomic_notifier_call_chain
```

Now the trace looks similar to the debug file system output.

Use the "-e" option to record events:

```
># trace-cmd record -e sched_switch
/sys/kernel/debug/tracing/events/sched_switch/filter
/sys/kernel/debug/tracing/events/*/sched_switch/filter
Hit Ctrl^C to stop recording
(^C)
[...]

># trace-cmd report
version = 6
cpus=8
        <idle>-0     [006] 21642.751755: sched_switch:         swapper/6:0 [120] R ==> trace-cmd:4876
[120]
        <idle>-0     [002] 21642.751776: sched_switch:         swapper/2:0 [120] R ==> sshd:1208 [120]
     trace-cmd-4875  [005] 21642.751782: sched_switch:         trace-cmd:4875 [120] D ==>
swapper/5:0 [120]
     trace-cmd-4869  [001] 21642.751792: sched_switch:         trace-cmd:4869 [120] S ==>
swapper/1:0 [120]
     trace-cmd-4873  [003] 21642.751819: sched_switch:         trace-cmd:4873 [120] S ==>
swapper/3:0 [120]
        <idle>-0     [005] 21642.751835: sched_switch:         swapper/5:0 [120] R ==> trace-cmd:4875
[120]
```

```
       trace-cmd-4877  [007] 21642.751847: sched_switch:         trace-cmd:4877 [120] D ==>
swapper/7:0 [120]
            sshd-1208  [002] 21642.751875: sched_switch:         sshd:1208 [120] S ==> swapper/2:0 [120]
          <idle>-0     [007] 21642.751880: sched_switch:         swapper/7:0 [120] R ==> trace-cmd:4877
[120]
       trace-cmd-4874  [004] 21642.751885: sched_switch:         trace-cmd:4874 [120] S ==>
swapper/4:0 [120]
          <idle>-0     [001] 21642.751902: sched_switch:         swapper/1:0 [120] R ==> irq/43-em1:865
[49]
       trace-cmd-4876  [006] 21642.751903: sched_switch:         trace-cmd:4876 [120] D ==>
swapper/6:0 [120]
          <idle>-0     [006] 21642.751926: sched_switch:         swapper/6:0 [120] R ==> trace-cmd:4876
[120]
      irq/43-em1-865   [001] 21642.751927: sched_switch:         irq/43-em1:865 [49] S ==> swapper/1:0
[120]
       trace-cmd-4875  [005] 21642.752029: sched_switch:         trace-cmd:4875 [120] S ==>
swapper/5:0 [120]


Notice that only the "sched_switch" name was used. trace-cmd will
search for a match of "-e"'s option for trace event systems, or single
trace events themselves. To trace all interrupt events:

># trace-cmd record -e irq sleep 10
/sys/kernel/debug/tracing/events/irq/filter
/sys/kernel/debug/tracing/events/*/irq/filter
[...]

Notice that when a command is passed to trace-cmd, it will just run that
command and exit the trace when complete.

># trace-cmd report
version = 6
cpus=8
          <idle>-0     [002] 21767.342089: softirq_raise:        vec=9 [action=RCU]
           sleep-4917  [007] 21767.342089: softirq_raise:        vec=9 [action=RCU]
          <idle>-0     [006] 21767.342089: softirq_raise:        vec=9 [action=RCU]
     ksoftirqd/0-3     [000] 21767.342096: softirq_entry:        vec=1 [action=TIMER]
     ksoftirqd/4-33    [004] 21767.342096: softirq_entry:        vec=1 [action=TIMER]
     ksoftirqd/3-27    [003] 21767.342097: softirq_entry:        vec=1 [action=TIMER]
     ksoftirqd/7-51    [007] 21767.342097: softirq_entry:        vec=1 [action=TIMER]
     ksoftirqd/4-33    [004] 21767.342097: softirq_exit:         vec=1 [action=TIMER]



To get the status information of events similar to what the debug
file system provides, add the "-l" (think "latency") option to the report.


># trace-cmd report -l
version = 6
cpus=8
  <idle>-0       3d.h20 21767.341545: softirq_raise:        vec=8 [action=HRTIMER]
ksoftirq-27      3...11 21767.341552: softirq_entry:        vec=8 [action=HRTIMER]
ksoftirq-27      3...11 21767.341554: softirq_exit:         vec=8 [action=HRTIMER]
  <idle>-0       4d.h20 21767.342085: softirq_raise:        vec=7 [action=SCHED]
  <idle>-0     0d.h20 21767.342086: softirq_raise:       vec=7 [action=SCHED]
  <idle>-0     3d.h20 21767.342086: softirq_raise:       vec=7 [action=SCHED]
   sleep-4917    7d.h10 21767.342086: softirq_raise:       vec=7 [action=SCHED]
  <idle>-0     6d.h20 21767.342087: softirq_raise:       vec=7 [action=SCHED]
  <idle>-0     2d.h20 21767.342087: softirq_raise:       vec=1 [action=TIMER]
  <idle>-0     1d.h20 21767.342087: softirq_raise:       vec=1 [action=TIMER]



Tracing all events
------------------

As mentioned above, the "-e" option to trace-cmd record is to choose
what event should be traced. You can specify either an individual event,
or a trace system:

>:# trace-cmd record -e irq

The above enables all tracepoints within the "irq" system.

>:# trace-cmd record -e irq_handler_enter
>:# trace-cmd record -e irq:irq_handler_enter

The commands above are equivalent and will enable the tracepoint
event "irq_handler_enter".

But then there is the case where you want to trace all events.
To do this, use the keyword "all".

>:# trace-cmd record -e all

This will enable all events.


Tracing tracers and events
--------------------------

As events can be enabled within any tracer, it makes sense that trace-cmd
would allow this as well. This is indeed the case. You may use both
the "-p" and the "-e" options at the same time.

>:# trace-cmd record -p function_graph -e all
[...]
>:# trace-cmd report
version = 6
cpus=8
       trace-cmd-1698  [002]  2724.485397: funcgraph_entry:                   |
kmem_cache_alloc() {
       trace-cmd-1699  [007]  2724.485397: funcgraph_entry:        0.073 us   |      find_vma();
       trace-cmd-1696  [000]  2724.485397: funcgraph_entry:                   |        lg_local_lock() {
       trace-cmd-1698  [002]  2724.485397: funcgraph_entry:        0.033 us   |
add_preempt_count();
       trace-cmd-1696  [000]  2724.485397: funcgraph_entry:                   |          migrate_disable() {
       trace-cmd-1699  [007]  2724.485398: funcgraph_entry:                   |      handle_mm_fault() {
       trace-cmd-1696  [000]  2724.485398: funcgraph_entry:        0.027 us   |
add_preempt_count();
```

```
      trace-cmd-1698  [002]  2724.485398: funcgraph_entry:       0.034 us   |
sub_preempt_count();
      trace-cmd-1699  [007]  2724.485398: funcgraph_entry:                  |
__mem_cgroup_count_vm_event() {
      trace-cmd-1696  [000]  2724.485398: funcgraph_entry:       0.031 us   |
pin_current_cpu();
      trace-cmd-1699  [007]  2724.485398: funcgraph_entry:       0.029 us   |         __rcu_read_lock();
      trace-cmd-1698  [002]  2724.485398: kmem_cache_alloc:     (return_to_handler+0x0)
call_site=ffffffff81662345 ptr=0xffff880114e260f0 bytes_req=240 bytes_alloc=240 gfp_flags=G
FP_KERNEL
      trace-cmd-1696  [000]  2724.485398: funcgraph_entry:       0.034 us   |
sub_preempt_count();
      trace-cmd-1699  [007]  2724.485398: funcgraph_entry:       0.028 us   |
__rcu_read_unlock();
      trace-cmd-1698  [002]  2724.485398: funcgraph_exit:        0.758 us   |                 }
      trace-cmd-1698  [002]  2724.485398: funcgraph_entry:       0.029 us   |
__rt_mutex_init();
      trace-cmd-1696  [000]  2724.485398: funcgraph_exit:        0.727 us   |           }
      trace-cmd-1699  [007]  2724.485398: funcgraph_exit:        0.466 us   |       }

Notice here that trace-cmd report does not disply the function graph
tracer any different than any other trace, like the "trace" file does.


Function filtering
------------------

The "set_ftrace_filter" and "set_ftrace_notrace" is very useful in
filtering out functions that you do not care about. These can be done
with trace-cmd as well.

The "-l" and "-n" are used the same as "set_ftrace_filter" and
"set_ftrace_notrace" respectively. Think of "limit functions" for
"-l" as the "-f" is used for event filtering.

To add more than one function to the list, either used the glob expressions
described previously, or use multiple "-l" or "-n" options.

># trace-cmd record -p function -l "sched*" -n "*stat*"

The above traces all functions that start with "sched" except those that
have "stat" in their names.



Event filtering
---------------

To filter events the same way as writing to the "filter" file inside
the "events" directory (see "Filtering events" above), use the "-f"
option. This option must follow the event that it will filter.

># trace-cmd record -e sched_switch -f "prev_prio < 100" \
   -e sched_wakeup -f 'comm == "bash"'
```

```
Graph a function
----------------

To perform a graph of a specific function using "function_graph" tracer,
trace-cmd provides the "-g" option.

>#  trace-cmd record -p function_graph -g sys_read ls /
[...]
>#  trace-cmd report
version = 6
CPU 3 is empty
CPU 4 is empty
CPU 5 is empty
cpus=8
      trace-cmd-2183  [006]  4689.643252: funcgraph_entry:                   |  sys_read() {
      trace-cmd-2183  [006]  4689.643253: funcgraph_entry:        0.147 us   |    fget_light();
      trace-cmd-2183  [006]  4689.643254: funcgraph_entry:                   |    vfs_read() {
      trace-cmd-2183  [006]  4689.643254: funcgraph_entry:                   |      rw_verify_area() {
      trace-cmd-2183  [006]  4689.643255: funcgraph_entry:                   |
security_file_permission() {
      trace-cmd-2183  [006]  4689.643255: funcgraph_entry:        0.068 us   |
cap_file_permission();
      trace-cmd-2183  [006]  4689.643256: funcgraph_entry:        0.064 us   |          __fsnotify_parent();
      trace-cmd-2183  [006]  4689.643256: funcgraph_entry:        0.095 us   |          fsnotify();
      trace-cmd-2183  [006]  4689.643257: funcgraph_exit:        1.792 us   |        }
      trace-cmd-2183  [006]  4689.643257: funcgraph_exit:        2.328 us   |      }
      trace-cmd-2183  [006]  4689.643257: funcgraph_entry:                   |      seq_read() {
      trace-cmd-2183  [006]  4689.643257: funcgraph_entry:                   |        _mutex_lock() {
      trace-cmd-2183  [006]  4689.643258: funcgraph_entry:        0.062 us   |          rt_mutex_lock();
      trace-cmd-2183  [006]  4689.643258: funcgraph_exit:        0.584 us   |        }
      trace-cmd-2183  [006]  4689.643259: funcgraph_entry:                   |        m_start() {
      trace-cmd-2183  [006]  4689.643259: funcgraph_entry:                   |          rt_down_read() {
      trace-cmd-2183  [006]  4689.643259: funcgraph_entry:                   |            rt_mutex_lock() {


Modify trace buffer size via trace-cmd
--------------------------------------

The trace-cmd record "-b" option lets you change the size of the
ftrace buffer before recording the trace. Note, currently trace-cmd
does not support per-cpu resize. The size is what is entered into
"buffer_size_kb" at the top level.


>#  trace-cmd record -b 10000 -p function


trace-cmd start, stop and extract
---------------------------------

The trace-cmd start command takes almost all the options as the trace-cmd
record command does. The difference between the two is that "start"
will only enable ftrace, it will not do any recording. It is equivalent
to enabling ftrace via the debug file system.

>#  trace-cmd start -p function -e all
```

```
>># cat /sys/kernel/debug/tracing/trace
# tracer: function
#
# entries-in-buffer/entries-written: 1544167/2039168   #P:8
#
#                              _-------=> irqs-off
#                             /  _------=> need-resched
#                             |/  _-----=> need-resched_lazy
#                             ||/  _----=> hardirq/softirq
#                             |||/  _---=> preempt-depth
#                             ||||/  _--=> preempt-lazy-depth
#                             ||||| / _-=> migrate-disable
#                             |||||| /     delay
#            TASK-PID   CPU#  |||||||    TIMESTAMP  FUNCTION
#               | |       |   |||||||       |         |
       trace-cmd-2390  [003] .......  5946.816132: _mutex_unlock <-rb_simple_write
       trace-cmd-2390  [003] .......  5946.816133: rt_mutex_unlock <-_mutex_unlock
       trace-cmd-2390  [003] .......  5946.816134: __fsnotify_parent <-vfs_write
       trace-cmd-2390  [003] .......  5946.816134: fsnotify <-vfs_write
       trace-cmd-2390  [003] .......  5946.816135: __srcu_read_lock <-fsnotify
       trace-cmd-2390  [003] .......  5946.816135: add_preempt_count <-__srcu_read_lock
       trace-cmd-2390  [003] ....1..  5946.816135: sub_preempt_count <-__srcu_read_lock
       trace-cmd-2390  [003] .......  5946.816135: __srcu_read_unlock <-fsnotify
       trace-cmd-2390  [003] .......  5946.816136: add_preempt_count <-__srcu_read_unlock
       trace-cmd-2390  [003] ....1..  5946.816136: sub_preempt_count <-__srcu_read_unlock
       trace-cmd-2390  [003] .......  5946.816137: syscall_trace_leave <-int_check_syscall_exit_work
       trace-cmd-2390  [003] .......  5946.816137: __audit_syscall_exit <-syscall_trace_leave
       trace-cmd-2390  [003] .......  5946.816137: path_put <-__audit_syscall_exit
       trace-cmd-2390  [003] .......  5946.816137: dput <-path_put
       trace-cmd-2390  [003] .......  5946.816138: mntput <-path_put
       trace-cmd-2390  [003] .......  5946.816138: unroll_tree_refs <-__audit_syscall_exit
       trace-cmd-2390  [003] .......  5946.816138: kfree <-__audit_syscall_exit
       trace-cmd-2390  [003] ....1..  5946.816139: kfree: call_site=ffffffff810eaff0 ptr=          (null)
       trace-cmd-2390  [003] ....1..  5946.816139: sys_exit: NR 1 = 1
       trace-cmd-2390  [003] d......  5946.816140: sys_write -> 0x1
       trace-cmd-2390  [003] d......  5946.816151: do_page_fault <-page_fault
       trace-cmd-2390  [003] d......  5946.816151: __do_page_fault <-do_page_fault
       trace-cmd-2390  [003] .......  5946.816152: rt_down_read_trylock <-__do_page_fault
       trace-cmd-2390  [003] .......  5946.816152: rt_mutex_trylock <-rt_down_read_trylock


Running trace-cmd stop is exactly the same as echoing "0" into the
"tracing_on" file in the debug file system. This only stops writing to
the trace buffers, it does not stop all the tracing mechanisms inside
the kernel and still adds some overhead to the system.

>># cat /sys/kernel/debug/tracing/tracing_on
1
>># trace-cmd stop
>># cat /sys/kernel/debug/tracing/tracing_on
0


Finally, if you want to create a "trace.dat" file from the ftrace
kernel buffers you use the "extract" command. The tracing could
have started with the "start" command or by manually modifying the
```

ftrace debug file system files. This is useful if you found a trace and want to save it off where you can send it to other people, and also have the full features of the trace-cmd "report" command.

```
>#  trace-cmd extract
>#  trace-cmd report
version = 6
cpus=8
CPU:6 [2544372 EVENTS DROPPED]
    ksoftirqd/6-45    [006]  6192.717580: function:             rcu_note_context_switch
    ksoftirqd/6-45    [006]  6192.717580: rcu_utilization:      ffffffff819e743b
    ksoftirqd/6-45    [006]  6192.717580: rcu_utilization:      ffffffff819e7450
    ksoftirqd/6-45    [006]  6192.717581: function:             add_preempt_count
    ksoftirqd/6-45    [006]  6192.717581: function:             kthread_should_stop
    ksoftirqd/6-45    [006]  6192.717581: function:             kthread_should_park
    ksoftirqd/6-45    [006]  6192.717581: function:             ksoftirqd_should_run
    ksoftirqd/6-45    [006]  6192.717582: function:             sub_preempt_count
    ksoftirqd/6-45    [006]  6192.717582: function:             schedule
    ksoftirqd/6-45    [006]  6192.717582: function:               __schedule
    ksoftirqd/6-45    [006]  6192.717582: function:                 add_preempt_count
    ksoftirqd/6-45    [006]  6192.717582: function:                 rcu_note_context_switch
    ksoftirqd/6-45    [006]  6192.717583: rcu_utilization:      ffffffff819e743b
    ksoftirqd/6-45    [006]  6192.717583: rcu_utilization:      ffffffff819e7450
    ksoftirqd/6-45    [006]  6192.717583: function:                 _raw_spin_lock_irq
    ksoftirqd/6-45    [006]  6192.717583: function:                   add_preempt_count
    ksoftirqd/6-45    [006]  6192.717584: function:                 deactivate_task
    ksoftirqd/6-45    [006]  6192.717584: function:                   dequeue_task
    ksoftirqd/6-45    [006]  6192.717584: function:                     update_rq_clock
```

The "extract" command takes a "-o" option to save the trace in a different name like the "record" command does. By default it just saves it into a file called "trace.dat".

Resetting the trace

As mentioned, the "stop" command does not lower the overhead of ftrace. It simply disables writing to the ftrace buffer. There's two ways of resetting ftrace with trace-cmd.

The first way is with the "reset" command.

```
>#  trace-cmd reset
```

This disables practically everything in ftrace. It also sets the "tracing_on" file to "0". It also erases everything inside the buffers, so make sure to do your "extract" before running the "reset" command.

The "reset" command also takes a "-b" option that lets you resize the buffer as well. This is useful to free the allocated buffers when you are finished tracing.

```
>#  trace-cmd reset -b 0
>#  cat /sys/kernel/debug/tracing/buffer_total_size_kb
```

```
8


The problem with the "reset" command is that it may make it hard to
use the debug file system tracing files directly. It may disable various
parts of tracing that may give unexpected results when trying to use
the files directly. If you plan to use ftrace's files directly after
using trace-cmd, the trick is to start the "nop" tracer.

>#  trace-cmd start -p nop

This sets up ftrace to run the "nop" tracer, which does no tracing and
has no overhead when enabled, and disables all events, and clears out
the "trace" file. After running this command, the system should be
set up to use the ftrace files directly as they are expected.



Using trace-cmd over the network
--------------------------------

If the target system to trace is limited on disk space, or perhaps
the disk usage is what is being traced, it can be prudent to record
the trace via another median than to the hard drive. The "listen"
command sets up a way for trace-cmd to record over the network.

[Server]
>$ mkdir traces
>$ cd traces
>$ trace-cmd listen -p 55577


Notice that the prompt above is "$". This denotes that the listen command
does not need to be root if the listening port is not a privileged port.

[Target]
>#  trace-cmd record -e all -N Server:55577 ls /


[Server]
connected!
Connected with Target:50671
cpus=8
pagesize=4096
version = 6
CPU0 data recorded at offset=0x3a7000
    0 bytes in size
CPU1 data recorded at offset=0x3a7000
    8192 bytes in size
CPU2 data recorded at offset=0x3a9000
    8192 bytes in size
CPU3 data recorded at offset=0x3ab000
    8192 bytes in size
CPU4 data recorded at offset=0x3ad000
    8192 bytes in size
CPU5 data recorded at offset=0x3af000
```

```
   8192 bytes in size
CPU6 data recorded at offset=0x3b1000
   4096 bytes in size
CPU7 data recorded at offset=0x3b2000
   8192 bytes in size
connected!
(^C)

>$ ls
trace.Target:50671.dat
>$ trace-cmd report trace.Target:50671.dat
version = 6
CPU 0 is empty
cpus=8
        <...>-2976  [007]  8865.266143: mm_page_alloc:        page=0xffffea00007e8740 pfn=8292160
order=0 migratetype=0 gfp_flags=GFP_KERNEL|GFP_REPEAT|GFP_ZERO|GFP_NOTRACK
        <...>-2976  [007]  8865.266145: kmalloc:              (pte_lock_init+0x2c) call_site=ffffffff8116d78c
ptr=0xffff880111e40d00 bytes_req=48 bytes_alloc=64 gfp_flags=GFP_KERNEL
        <...>-2976  [007]  8865.266152: mm_page_alloc:        page=0xffffea00034a50c0
pfn=55201984 order=0 migratetype=0
gfp_flags=GFP_KERNEL|GFP_REPEAT|GFP_ZERO|GFP_NOTRACK
        <...>-2976  [007]  8865.266153: kmalloc:              (pte_lock_init+0x2c) call_site=ffffffff8116d78c
ptr=0xffff880111e40e40 bytes_req=48 bytes_alloc=64 gfp_flags=GFP_KERNEL
        <...>-2976  [007]  8865.266155: mm_page_alloc:        page=0xffffea000307d380
pfn=50844544 order=0 migratetype=2 gfp_flags=GFP_HIGHUSER_MOVABLE
        <...>-2976  [007]  8865.266167: mm_page_alloc:        page=0xffffea000323f900 pfn=52689152
order=0 migratetype=2 gfp_flags=GFP_HIGHUSER_MOVABLE
        <...>-2976  [007]  8865.266171: mm_page_alloc:        page=0xffffea00032cda80
pfn=53271168 order=0 migratetype=2 gfp_flags=GFP_HIGHUSER_MOVABLE
        <...>-2976  [007]  8865.266192: hrtimer_cancel:       hrtimer=0xffff88011ebccf40
       <idle>-0     [006]  8865.266193: hrtimer_cancel:       hrtimer=0xffff88011eb8cf40
        <...>-2976  [007]  8865.266193: hrtimer_expire_entry: hrtimer=0xffff88011ebccf40
now=8905356001470 function=tick_sched_timer/0x0
       <idle>-0     [006]  8865.266194: hrtimer_expire_entry: hrtimer=0xffff88011eb8cf40
now=8905356002620 function=tick_sched_timer/0x0
        <...>-2976  [007]  8865.266196: sched_stat_runtime:   comm=trace-cmd pid=2976
runtime=228684 [ns] vruntime=2941412131 [ns]
       <idle>-0     [006]  8865.266197: softirq_raise:        vec=1 [action=TIMER]
       <idle>-0     [006]  8865.266197: rcu_utilization:      ffffffff819e740d
        <...>-2976  [007]  8865.266198: softirq_raise:        vec=1 [action=TIMER]
       <idle>-0     [006]  8865.266198: softirq_raise:        vec=9 [action=RCU]
        <...>-2976  [007]  8865.266199: rcu_utilization:      ffffffff819e740d
```

By default, the data is transfered via UDP. This is very efficient but it is possible to lose data and not know it. If you are worried about a full connection, then use the TCP protocol. The "-t" option on the "record" command forces trace-cmd to send the data over a TCP connection instead of a UDP one.

Summary

This document just highlighted the most common features of ftrace and trace-cmd. For more in depth look at what trace-cmd can do, read

> the man pages:
>
> ```
> trace-cmd
> trace-cmd-record
> trace-cmd-report
> trace-cmd-start
> trace-cmd-stop
> trace-cmd-extract
> trace-cmd-reset
> trace-cmd-listen
> trace-cmd-split
> trace-cmd-restore
> trace-cmd-list
> trace-cmd-stack
> ```

APPENDIX C. REVISION HISTORY

Revision 1-6	**Tue Aug 6 2019**	**Jaroslav Klech**
Preparing document for 7.7 GA publication.		
Revision 1-5	**Fri Oct 19 2018**	**Jaroslav Klech**
Preparing document for 7.6 GA publication.		
Revision 1-4	**Mon Mar 26 2018**	**Marie Doleželová**
Preparing document for 7.5 GA publication.		
Revision 1-3	**Tue Jul 25 2017**	**Jana Heves**
Version for 7.4 GA publication.		
Revision 1-2	**Mon Nov 3 2016**	**Maxim Svistunov**
Version for 7.3 GA publication.		
Revision 1-1	**Fri Nov 06 2015**	**Tomáš Čapek**
Version for 7.2 GA publication.		
Revision 1-0	**Fri Feb 13 2015**	**Radek Bíba**
Version for 7.1 GA publication.		