Finding Origins of Latencies Using Ftrace

Steven Rostedt
Red Hat, Inc.
1801 Varsity Drive,Raleigh, North Carolina 27606,USA
srostedt@redhat.com

Abstract

One of the difficult tasks analyzing Real-Time systems is finding a source/cause of an unexpected latency. Is the latency caused by the application or the kernel? Is it a wake up scheduling latency or a latency caused by interrupts being disabled, or is it a latency caused by preemption being disabled, or a combination of disabled interrupts and preemption.

Ftrace has its origins from the -rt patch [1] latency tracer, and still carries the capabilities to track down latencies. It can catch the maximum wake up latency for the highest priority task. This wake up latency can also be tuned to only trace real-time processes. There is a latency tracer to find the latency of how long interrupts and/or preemption are disabled. The maximum latency is captured and you can even see the functions that were called in the mean time. Ftrace also has a rich array of tracing features that can help determine if latencies are caused by the kernel, or simply are a bi-product of an application.

1 Introduction

Ftrace has its control files in the debugfs system. This is usually mounted in `/sys/kernel/debug`. If it is not already mounted, then you can mount it yourself with:

```
# mount -t debugfs nodev /sys/kernel/debug
# cd /sys/kernel/debug/tracing
# ls
available_events            set_ftrace_notrace
available_filter_functions  set_ftrace_pid
available_tracers           set_graph_function
buffer_size_kb              stack_max_size
current_tracer              stack_trace
dyn_ftrace_total_info       sysprof_sample_period
events                      trace
failures                    trace_marker
function_profile_enabled    trace_options
options                     trace_pipe
per_cpu                     trace_stat
printk_formats              tracing_cpumask
README                      tracing_enabled
saved_cmdlines              tracing_max_latency
set_event                   tracing_on
set_ftrace_filter           tracing_thresh
```

As you can tell, there are a lot of files in this directory. We will only be concerning ourselves with those that will help us trace latencies in the system. Those are:

1. available_tracers
2. current_tracer
3. events
4. trace
5. trace_marker
6. trace_max_latency
7. tracing_on

The version of the kernel that I am using for this paper is 2.6.31-rc6-rt4.

2 Enabling Plugin Tracers

Looking in the file `available_tracers`, you will see the available tracers that have been configured.

We are most interested in `irqsoff`, `preemptoff`, `preemptirqsoff`, `wakeup_rt`, and `wakeup`. To enable a plugin, you simply echo the name into the `current_tracer` file:

```
# echo preemptoff > current_tracer
```

All examples in this document will assume that you have mounted the debugfs directory and changed directory into the debugfs tracing directory.

You can see the tracer that is activated by cat'ing the `current_tracer` file.

```
# cat current_tracer
preemptoff
```

To disable the plugin, echo the special `nop` tracer into the `current_tracer` file. The `nop` tracer is special in that it is not a plugin tracer, but lets the user disable all plugins.

```
# echo nop > current_tracer
# cat current_tracer
nop
```

3 Things That Might Cause Latencies

There are various events that can trigger a latency. First, lets define what a latency is. A latency is the time between an event is suppose to occur and when it actually does. The term **latency tracer** is really a misnomer, because the tracing tools do not actually trace latency, but instead it traces events that may cause a latency.

Within the kernel, there are basically four different events that can cause a latency.

1. Interrupts disabled - keeping interrupts from calling their handlers when a device triggers an interrupt to the CPU.
2. Preemption disabled - preventing a process that just woke up from running.
3. Scheduling latency - the time it takes a process to schedule in.
4. Interrupt inversion - the time that an interrupt handler is performing a task that is lower in priority than the task that it preempted.

Ftrace latency tracers can record the first three. But the interrupt inversion is not covered by the latency tracer but can be seen with other tracers.

4 Measuring Interrupts Disabled

Ftrace piggy backs on top of `lockdep` [2] to measure interrupts disabled. `Lockdep` is a tool made by Ingo Molnar that can detect possible deadlock scenarios within the kernel. It keeps track of locks that are taken and can check the order of locks to ensure that two locks are always taken in the proper order. `Lockdep` also makes sure that a `spinlock` that is used within an interrupt is not taken without disabling interrupts. If you do not understand `lockdep`, do not worry, it is beyond the scope of this paper. What is important, is that `lockdep` keeps track of all locations that interrupts are disabled as well as when they are enabled. Ftrace uses this implementation to record when interrupts are disabled and enabled.

Note: ftrace hooks into the `lockdep` infrastructure, but you do not need to enable `lockdep` to use the interrupt tracer. By enabling `lockdep` you will add even more overhead. If you are concerned about measuring latency and not debugging the locking of the kernel, then it is recommended to keep `lockdep` disabled (`CONFIG_PROVE_LOCKING` and `CONFIG_LOCKDEP`).

Measuring the time that interrupts are disabled in the system is key to for analyzing causes of latency. If interrupts are disabled when an event occurs, then that event must wait till interrupts are enabled to continue. The time it must wait is added latency on top of the overhead to get to the event. When interrupts are disabled, a device that sends an interrupt to the CPU will not be noticed until interrupts are re-enabled.

4.1 Irqsoff Latency Tracer

The plugin `irqsoff` is a way to measure times in the kernel that interrupts are disabled. Listings 1, 2, 3, and 4 show the output of running irqsoff tracer for just a little while. As you can see by the fact that I needed to break this up over 4 pages, it can get a bit verbose. This is due to the function tracer.

The function tracer (enabled by `CONFIG_FUNCTION_TRACER`) is a way to trace almost all functions in the kernel. When function tracing is enabled, the kernel is compiled with the gcc option `-pg`. This is a profiler that will make all functions call a special function named `mcount`. One would realize that this could cause a very large overhead, but if the kernel is also configured with dynamic function tracing (`CONFIG_DYNAMIC_FTRACE`) then these calls, when not in use, are converted at run time to `nops`. This allows the function tracer to have zero overhead when not in use. If you do not understand this part, don't worry, you do not need to understand the implementation to use it. Just realize that enabling the dynamic function tracer gives you great power with no overhead.

4.2 The Heisenberg Principle

Any computer scientist (or any scientist for that matter) should be aware of the **Heisenberg Principle** [3]. Basically this means that the act of measuring something can and will modify the result. This is especially true with the interrupt tracer and even more so when the function tracer is enabled. The idea is to trace the time interrupts are disabled, but by adding a tracer to these core functions, it adds a little overhead. By running with the function tracer, it adds even more overhead to the time interrupts are disabled, because we are tracing every function that is called within the critical section.

You do not need to unconfigure the function tracer to keep it from running while tracing interrupt latency. There exists a proc file that lets you disable the function tracer from running at run time.

```
# echo 0 > /proc/sys/kernel/ftrace_enabled
```

This will allow you to find something a bit closer to the actual latency[1]. Listing 5 shows the result of a latency trace with the function tracer disabled.

To get a good idea of the overhead, the benchmark test `hackbench` [4] can show the results well. Running `hackbench` with the function tracer enabled yields a test run time of 47.686 seconds and a max latency of 171 microseconds (way above the max that we allow for the real-time kernel). Running `hackbench` with the function tracing disabled, yields a test run of 34.361 seconds and a max latency of 30 microseconds[2]. Note: running `hackbench` with both tracers disabled only took a running time of 9.774 seconds. I do not know the latency because it was not being traced.

Note: when enabling or disabling the function tracing for the latency tracers, it is best to reset the tracer or it may take effect. That is, echo in `nop` into the `current_tracer` file and `irqsoff` again.

5 Reading the Trace

Before we continue to the other tracers, a description of how to read the output is in order. The lines in the Listings of 1, 2, 3 and 4 are numbered. We will go through some of the lines and explain their meanings.

Lines 001 through 018 is the latency tracer header, and is annotated with a '#' at the beginning of the line. Line 001 states the name of the current plugin tracer. Line 003 has the kernel version that is executing (ignore the trace version, that has not changed in a long time). Line 005 has a bit of information. Here we see that the latency trace recorded a 70 microsecond time that interrupts were disabled. This may be different than the last trace entry, but not by much, due to the tracer writing entries after it took the finishing time stamp. The #170/170 means that there was 170 entries printed out of 170 that were recorded. Since the latency trace ftrace plugins are usually small[3] the two numbers should always match. But for other tracers, it is quite possible to have the first number smaller than the second due to the trace ring buffer overwriting older data.

The CPU#0 shows that this latency happened on CPU 0. Inside the parenthesis, the VP, KP, SP and HP will always be zero since they are not yet implemented. The M element shows what type of preemption the kernel was configured at. Here it is "preempt" but really should be "preempt-rt". Since the latency tracer has been replaced with the upstream ftrace, this field has not been updated. The other selections of preempt type are "desktop" for CONFIG_PREEMPT_VOLUNTARY (kernel preempts only at preemption points) or "server" for CONFIG_PREEMPT_NONE (no preemption inside the kernel). The #P:2 shows that there were 2 online CPUS active.

Line 007 shows information about the task that was executing when the latency was recorded. The task here was "sirqtimer/0" with process id 5. The policy shows that it was running under SCHED_FIFO (1) where as 0 would be a non real-time running the SCHED_NORMAL policy. SCHED_RR is represented with 2, SCHED_BATCH is 3, and SCHED_IDLE is 5. Because this is running under a real-time policy, the nice value can be ignored. The `rt_prio` field is the real-time prio as

[1] Note: you must have a space between the 0 and the > otherwise the shell will interpret it as a redirection of standard I/O.
[2] hackbench did not even get on the radar in this run
[3] 170 is small compared to thousands that the function tracer can do.

maintained in the kernel. This can be a little confusing. Real time priorities for users range from 1 to 99, but these are represented in the kernel as 98 to 0, where the lower the number, the higher the priority. The trace shows the priority to be 49, but that is the kernel's representation. To convert the `rt_prio` to the user priority, subtract it from 99. The user priority of this task is actually 50.

The `trace_hardirqs_off_thunk` is a helper function called from assembly to trace when interrupts are disabled there, usually by entering of an interrupt. When an interrupt occurs, interrupts are disabled. Looking at the first function called on line 020 we can see the APIC timer interrupt went off.

Line 020 starts off with the `cmd` (the kernel name of the task) and the process id. The task at this recording is `bash` and its process id is 2724. Even though the trace header shows the task was `sirq-timer` a schedule switch happened inside this disabling of interrupts and ending task was `sirq-timer`. The next five items are labeled in the header. The first is the CPU number. The second is whether interrupts were disabled. A **d** means that interrupts are disabled. In the irqs off trace, all lines should show that interrupts are disabled. When interrupts are enabled a period (.) will be displayed.

The third item is for `need-resched`. When the kernel determines that a schedule should take place because a higher priority task woke up or the current running task is at the end of its time slice, it sets a `need-resched` flag to signal that a schedule should take place. The trace will annotate this with a **N** in that field. Line 093 shows this being set when we wake up the `sirq-timer` task that is of higher priority than the `bash` task. When the `need-resched` flag is not set, a period (.) is displayed.

The forth item denotes if we are in a hard interrupt or soft interrupt. The soft interrupt is a little misnomer because it really only denotes soft interrupts are disabled. Soft interrupts are disabled when ever the kernel is running a soft interrupt, as one soft interrupt can not preempt another. A **h** means that the trace was recorded in an interrupt. A **s** denotes that soft interrupts are disabled or the trace was recording inside a soft interrupt. A **H** denotes that the trace was recorded in an interrupt and soft interrupts are also disabled. Since the Real-Time Linux kernel runs the soft interrupts as threads, the soft interrupt disabling is not applicable. When hard and soft interrupts are enabled, a period (.) is displayed.

The fifth item denotes the preempt disable depth. When a kernel disables preemption in critical sections[4], it uses a preempt counter. The preempt count is recorded in all traces, and this field shows the value when it is greater than zero. When preemption is enabled, this field will contain a period (.). Line 041 shows a preempt depth of 1 that was caused by the `_atomic_spin_lock` just before it. Line 046 shows a preempt depth of 2 caused by the `_atomic_spin_lock_irqsave` before it.

Lines 181 and 182 shows the schedule switch that took place between the tasks `bash` and `sirq-timer`. Only the first 8 characters of the task name are printed, as can be seen by the truncated name of the task `sirq-timer`.

6 Preemption Disabled Tracing

When interrupts are disabled, events from devices and timers and even inter-processor communication is disabled. But the kernel can keep interrupts enabled but disable preemption. This allows devices and timers to be able to notify the CPU that an event has happened, but if a task should wake up because of it, it must wait till the kernel comes to a place it can preempt before it will schedule. The `preemptoff` plugin tracer will trace the maximum time that preemption is disabled.

Measuring the time preemption is disabled may be something used for academics, but it has really no practical meaning by itself. Being able to trace the time that both interrupts are disabled and/or preemption is disabled is much more informative. This is the total time that a task can not be scheduled. If interrupts are disabled, no event can occur to cause a preemption. The scheduler will not be called if preemption is enabled but interrupts are not. The `preemptirqsoff` plugin tracer shows this information.

```
[root@mxf tracing]# echo preemptirqsoff > current_tracer
```

The output for this trace is not much different than the output of the `irqsoff` trace so I will omit it from this paper.

7 Using the Event Tracer

Since the function tracing can add a large overhead it is not always practical to use it. But without the

[4] In this case we can see spin locks disable preemption

function tracing enabled, the information may not be enough to see what is happening. Luckily, there is the **event tracer**. The event tracing is not a plugin. When events are enabled, they will be recorded in any plugin, including the special `nop` plugin.

There are two ways to enable events. One is with the `set_event` file and the other is with the `events` directory. The `set_event` file is a way to echo in events to enable them. The available events are:

```
[root@mxf tracing]# cat available_events
skb:kfree_skb
block:block_rq_abort
block:block_rq_insert
block:block_rq_issue
block:block_rq_requeue
block:block_rq_complete
block:block_bio_bounce
block:block_bio_complete
block:block_bio_backmerge
block:block_bio_frontmerge
block:block_bio_queue
block:block_getrq
block:block_sleeprq
block:block_plug
block:block_unplug_timer
block:block_unplug_io
block:block_split
block:block_remap
kmem:kmalloc
kmem:kmem_cache_alloc
kmem:kmalloc_node
kmem:kmem_cache_alloc_node
kmem:kfree
kmem:kmem_cache_free
lockdep:lock_acquire
lockdep:lock_release
workqueue:workqueue_insertion
workqueue:workqueue_execution
workqueue:workqueue_creation
workqueue:workqueue_destruction
irq:irq_handler_entry
irq:irq_handler_exit
irq:softirq_entry
irq:softirq_exit
sched:sched_kthread_stop
sched:sched_kthread_stop_ret
sched:sched_wait_task
sched:sched_wakeup
sched:sched_wakeup_new
sched:sched_switch
sched:sched_migrate_task
sched:sched_process_free
sched:sched_process_exit
sched:sched_task_setprio
sched:sched_process_wait
sched:sched_process_fork
sched:sched_signal_send
```

The name before the colon is the system that the event is under. The event name is after the colon[5]. By echoing in the system name you will enable all the events in that system.

```
[root@mxf tracing]# echo irq > set_event
[root@mxf tracing]# cat set_event
irq:irq_handler_entry
irq:irq_handler_exit
irq:softirq_entry
irq:softirq_exit
```

Echoing in just the event name will enable the event as well. But if there are two event names under two systems that are identical, then both will be enabled. Currently no two event names are identical.

Adding new names follows shell concatenation rules. Using a '>' will truncate the file and disable the events that were previously enabled. Using a '>>' will add new events without disabling the ones that are currently enabled.

```
[root@mxf tracing]# echo sched_switch >> set_event
[root@mxf tracing]# cat set_event
irq:irq_handler_entry
irq:irq_handler_exit
irq:softirq_entry
irq:softirq_exit
sched:sched_switch
```

The '!' character can be used to remove events. Note that this is also a bash command so it must be added in quotes.

```
[root@mxf tracing]# echo '!softirq_entry' >> set_event
[root@mxf tracing]# cat set_event
irq:irq_handler_entry
irq:irq_handler_exit
irq:softirq_exit
sched:sched_switch
```

The `events` directory is also useful. The directory structure is made of the event systems, and within each system directory is the events. Each level has an `enable` file.

```
[root@mxf tracing]# ls events/
block     ftrace  header_page  kmem     sched
workqueue enable  header_event irq      lockdep  skb
[root@mxf tracing]# ls events/irq
enable  irq_handler_entry  softirq_entry
filter  irq_handler_exit   softirq_exit
[root@mxf tracing]# ls events/irq/softirq_exit/
enable  filter  format  id
```

[5] The kernel I have has `lockdep` enabled where you can see from the `lockdep` events

To enable all events echo 1 into `events/enable`. To enable all events within a system, echo 1 into the system enable file (`events/irq/enable`). To enable just a single event, echo 1 into the enable file for that event (`events/irq/softirq_entry/enable`). Echoing in a 0 will disable the same events that a 1 would enable.

Just enabling the scheduling events yields a nice useful output, as seen in Listing 6.

8 Tracing Scheduling Latencies

The time a task is awoken to the time it is scheduled is considered the scheduling latency. Two plugin tracers exist to measure this latency. The **wakeup** plugin will consider all tasks and the **wakeup_rt** will only consider real-time tasks to trace. Both of these plugins only trace the current highest priority task of the system. The trace records the max latency, and tracing anything but the highest priority task would lose the trace for the highest task, because the highest task may cause a lower priority task to take a long time to be scheduled.

If you are concerned about the wake up times of all tasks, simply enable all the scheduling events and examine the trace with the nop plugin.

Listing 7 shows the output of the `wakeup` plugin. Notice that the time is quite exaggerated. This is because of the way the wakeup tracer works. The tracer picks the highest priority task that has started. If it wakes up another task of equal priority it does not switch the trace to that task. Although we see that our wake up latency was 150 microseconds, the true wake up was only 7 microseconds. What happened was after the highest priority task `hald-addon-stor` with pid 2103 was woken up, we see that the task `hald` with pid 1952 was woken up afterwards. But since the two tasks have the same priority, the tracer did not switch over to test the wake up time of the second task. The scheduler chose the second task (`hald`) first, and the trace included the entire time that the task `hald` ran. Luckily it only ran for 142 microseconds.

If we chose not to enable the scheduling events we would not have seen the `hald` task wake up and we would assume that the true scheduling latency was 150 microseconds.

The `wakeup_rt` plugin only records real-time tasks. The tracer only records the maximum trace which makes the `wakeup` plugin hide real-time tasks latencies. If it constantly records the long latency that is described above, then we will never see the latencies of real-time tasks that we care about. This is why there are two plugins to record scheduling latencies. Listing 8 shows the output of the `wakeup_rt` plugin.

The values in the scheduling events needs a little explanation. The first trace item (which is from the `wakeup_rt` tracer) shows that task with pid 6808 at priority 120[6]. The **R** means that it is in the running state[7]. The + denotes that it is waking up the task that follows. The task that is being woken up is the `migration/0` task with the pid of 3. It's priority is zero which is the highest priority Linux supports (99 - 0 = 99 user level priority). The `migration/0` task is in the sleep state denoted by the **S**. The number inside the brackets (`[000]`) is the CPU that the task that is being woken up on is assigned to. The task may migrate before it wakes up.

The third trace item is a scheduling event (denoted with the `sched_switch:`). This even is the scheduling context switch between the current running task (`bash`) and the task that was just woken up (`migration/0`). This time the first number in the bracket (`[120]`) is the priority of the current task with the state of the task in parenthesis (**(R)**). The "==>" also denotes a scheduling switch is occurring, followed by the task that is scheduling, its pid and in brackets, its priority (`[0]`).

The third and last events are pretty much identical. The third event came from enabling the scheduling events, and the last event is part of the `wakeup_rt` plugin tracer.

9 Adding Placeholders into the Trace

When you discover that there exists an unexpected latency in your system and none of the latency plugin tracers showed anything, then you may need to confirm that the issue may be with something in userspace. Assuming that you have access to the source code of the application, you can make have the application write into the tracer ring buffer.

Just enabling the events (for now we'll enable the `sched` and `irq` events) and running with the `nop` tracer, you can watch what is happening with your application.

[6] This is the kernel internal priority. Anything over 100 is a nice value. Here it is 120 - 20 = nice value of zero

[7] This field holds the same enumerations that `top` uses.

Two particular files are of importance.

1. **tracing_on** - Enabling and disabling the tracer.

2. **trace_marker** - Writing into the trace buffer.

Since the tracing facility uses a ring buffer that overwrites older data with newer data, it is not important to enabled the trace, but instead let it constantly run. When you hit a point where you detect a latency is when you want to disable it. Having a file descriptor opened to both of the above files lets you see what is happening inside the application as well as stop tracing as soon as a latency is detected. Stopping the trace as soon as it happens is critical since you do not want to overwrite the trace that recorded the latency, as well as it will be easier to find the trouble area if it is relatively close to the end of the trace.

```
[root@mxf tracing]# echo Hello Dresden > trace_marker
[root@mxf tracing]# cat trace
# tracer: nop
#
#       TASK-PID    CPU#    TIMESTAMP  FUNCTION
#          | |       |          |         |
        bash-2702  [001]  8934.777334: 0: Hello Dresden
```

Inside the application, you can add comments to the trace at particular points and use them as markers to what is happening in the kernel.

```
write(trace_mark_fd, "hit this point\n", 15);
```

When you detect a latency inside the application you can stop the trace by writing the ASCII character '0' (zero) to the `tracing_on` file.

```
/* Detected latency, stop the trace */
write(tracing_on_fd, "0", 1);
```

Using this in combination with the event tracers (or even the full function tracer) will help tremendously with finding latency problems in your application.

There is another plugin that is useful with the above: The **syscall** plugin. This is similar to the `strace` tool but it traces all programs, not just one. The output ends up in the trace. Mixing the `syscall` plugin along with the event tracing will give lots of useful information to pin point trouble areas in you application.

10 Conclusions

During the early development of the -rt patch, some of our first testers were from audio users. People using the `jack` [5] utility to record music. They found the -rt patch gave the minimum latencies to record without defects. But every so often, they would come across something that would exceed the minimum latency, and would complain to us. Using the early `latency_tracer` we were able to find bugs in the kernel that caused their latencies and fixed them. But there were times that the `latency_tracer` proved that the latency was not in the kernel, and with further investigation, bugs in the `jack` utility were being discovered.

When your application fails to meet a deadline, it can happen due to several issues: hardware, latency in the kernel, or a bug in the application itself. When you discover something has gone wrong, the next step is to find what and where the problem arises. Having a good set of tracing utilities at your disposal will facilitate solving these issues.

References

[1] *RT Patch*,`http://www.kernel.org/pub/linux/kernel/projects/rt/`, `http://rt.wiki.kernel.org/index.php/Main_Page`

[2] *lockdep*, Ingo Molnar, See `Documentation/lockdep-design.txt` in the Linux kernel source.

[3] *Heisenberg Principle*, `http://en.wikipedia.org/wiki/Uncertainty_principle`

[4] *hackbench*, Rusty Russell, `http://devresources.linux-foundation.org/craiger/hackbench/src/hackbench.c`

[5] *JackAudio*, `http://jackaudio.org/`

```
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 2.6.31-rc6-rt4
# --------------------------------------------------------------------
# latency: 70 us, #170/170, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#    -----------------
#    | task: sirq-timer/0-5 (uid:0 nice:-5 policy:1 rt_prio:49)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /
#                |||||     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||   \   |   /
    bash-2724    0d...    1us : trace_hardirqs_off_thunk <-save_args
    bash-2724    0d...    1us : smp_apic_timer_interrupt <-apic_timer_interrupt
    bash-2724    0d...    2us : apic_write <-smp_apic_timer_interrupt
    bash-2724    0d...    2us : native_apic_mem_write <-apic_write
    bash-2724    0d...    2us : exit_idle <-smp_apic_timer_interrupt
    bash-2724    0d...    3us : irq_enter <-smp_apic_timer_interrupt
    bash-2724    0d...    3us : rcu_irq_enter <-irq_enter
    bash-2724    0d...    4us : idle_cpu <-irq_enter
    bash-2724    0d.h.    4us : hrtimer_interrupt <-smp_apic_timer_interrupt
    bash-2724    0d.h.    5us : ktime_get <-hrtimer_interrupt
    bash-2724    0d.h.    5us : clocksource_read <-ktime_get
    bash-2724    0d.h.    5us : _atomic_spin_lock <-hrtimer_interrupt
    bash-2724    0d.h1    6us : hrtimer_rt_defer <-hrtimer_interrupt
    bash-2724    0d.h1    6us : __run_hrtimer <-hrtimer_interrupt
    bash-2724    0d.h1    7us : __remove_hrtimer <-__run_hrtimer
    bash-2724    0d.h1    7us : timer_stats_account_hrtimer <-__run_hrtimer
    bash-2724    0d.h1    7us : _atomic_spin_unlock <-__run_hrtimer
    bash-2724    0d.h.    8us : tick_sched_timer <-__run_hrtimer
    bash-2724    0d.h.    8us : ktime_get <-tick_sched_timer
    bash-2724    0d.h.    8us : clocksource_read <-ktime_get
    bash-2724    0d.h.    9us : tick_do_update_jiffies64 <-tick_sched_timer
    bash-2724    0d.h.    9us : _atomic_spin_lock <-tick_do_update_jiffies64
    bash-2724    0d.h1   10us : do_timer <-tick_do_update_jiffies64
    bash-2724    0d.h1   10us : update_wall_time <-do_timer
    bash-2724    0d.h1   10us : clocksource_read <-update_wall_time
    bash-2724    0d.h1   11us : clocksource_get_next <-update_wall_time
    bash-2724    0d.h1   11us : _atomic_spin_lock_irqsave <-clocksource_get_next
    bash-2724    0d.h2   12us : _atomic_spin_unlock_irqrestore <-clocksource_get_next
    bash-2724    0d.h1   12us : update_vsyscall <-update_wall_time
    bash-2724    0d.h1   13us : _atomic_spin_lock <-update_vsyscall
    bash-2724    0d.h2   13us : _atomic_spin_unlock <-update_vsyscall
    bash-2724    0d.h1   14us : calc_global_load <-do_timer
    bash-2724    0d.h1   14us : _atomic_spin_unlock <-tick_do_update_jiffies64
    bash-2724    0d.h.   14us : update_process_times <-tick_sched_timer
    bash-2724    0d.h.   15us : account_process_tick <-update_process_times
    bash-2724    0d.h.   15us : account_system_time <-account_process_tick
    bash-2724    0d.h.   16us : cpuacct_update_stats <-account_system_time
    bash-2724    0d.h.   16us : __rcu_read_lock <-cpuacct_update_stats
    bash-2724    0d.h.   17us : __rcu_read_unlock <-cpuacct_update_stats
    bash-2724    0d.h.   17us : acct_update_integrals <-account_system_time
    bash-2724    0d.h.   18us : jiffies_to_timeval <-acct_update_integrals
    bash-2724    0d.h.   18us : run_local_timers <-update_process_times
    bash-2724    0d.h.   18us : hrtimer_run_queues <-run_local_timers
```

Listing 1: Interrupts Off Latency Trace (Part 1)

```
062:    bash-2724   0d.h.   19us : raise_softirq <-run_local_timers
063:    bash-2724   0d.h.   19us : raise_softirq_irqoff <-raise_softirq
064:    bash-2724   0d.h.   19us : wakeup_softirqd <-raise_softirq_irqoff
065:    bash-2724   0d.h.   20us : wake_up_process <-wakeup_softirqd
066:    bash-2724   0d.h.   20us : try_to_wake_up <-wake_up_process
067:    bash-2724   0d.h.   21us : task_rq_lock <-try_to_wake_up
068:    bash-2724   0d.h.   21us : _atomic_spin_lock <-task_rq_lock
069:    bash-2724   0d.h1   21us : update_rq_clock <-try_to_wake_up
070:    bash-2724   0d.h1   22us : select_task_rq_rt <-try_to_wake_up
071:    bash-2724   0d.h1   22us : activate_task <-try_to_wake_up
072:    bash-2724   0d.h1   23us : enqueue_task <-activate_task
073:    bash-2724   0d.h1   23us : enqueue_task_rt <-enqueue_task
074:    bash-2724   0d.h1   23us : enqueue_rt_entity <-enqueue_task_rt
075:    bash-2724   0d.h1   24us : __enqueue_rt_entity <-enqueue_rt_entity
076:    bash-2724   0d.h1   24us : cpupri_set <-__enqueue_rt_entity
077:    bash-2724   0d.h1   25us : _atomic_spin_lock_irqsave <-cpupri_set
078:    bash-2724   0d.h2   25us : _atomic_spin_unlock_irqrestore <-cpupri_set
079:    bash-2724   0d.h1   26us : _atomic_spin_lock_irqsave <-cpupri_set
080:    bash-2724   0d.h2   26us : _atomic_spin_unlock_irqrestore <-cpupri_set
081:    bash-2724   0d.h1   27us : update_rt_migration <-__enqueue_rt_entity
082:    bash-2724   0d.h1   27us : check_preempt_curr <-try_to_wake_up
083:    bash-2724   0d.h1   28us : check_preempt_wakeup <-check_preempt_curr
084:    bash-2724   0d.h1   28us : update_curr <-check_preempt_wakeup
085:    bash-2724   0d.h1   28us : calc_delta_fair <-update_curr
086:    bash-2724   0d.h1   29us : cpuacct_charge <-update_curr
087:    bash-2724   0d.h1   29us : __rcu_read_lock <-cpuacct_charge
088:    bash-2724   0d.h1   30us : __rcu_read_unlock <-cpuacct_charge
089:    bash-2724   0d.h1   30us : account_group_exec_runtime <-update_curr
090:    bash-2724   0d.h1   30us : resched_task <-check_preempt_wakeup
091:    bash-2724   0d.h1   31us : test_tsk_need_resched <-resched_task
092:    bash-2724   0d.h1   31us : test_ti_thread_flag <-test_tsk_need_resched
093:    bash-2724   0dNh1   32us : task_wake_up_rt <-try_to_wake_up
094:    bash-2724   0dNh1   32us : test_tsk_need_resched <-task_wake_up_rt
095:    bash-2724   0dNh1   32us : test_ti_thread_flag <-test_tsk_need_resched
096:    bash-2724   0dNh1   33us : task_rq_unlock <-try_to_wake_up
097:    bash-2724   0dNh1   33us : _atomic_spin_unlock_irqrestore <-task_rq_unlock
098:    bash-2724   0dNh.   33us : preempt_schedule <-_atomic_spin_unlock_irqrestore
099:    bash-2724   0dNh.   34us : softlockup_tick <-run_local_timers
100:    bash-2724   0dNh.   34us : __touch_softlockup_watchdog <-softlockup_tick
101:    bash-2724   0dNh.   35us : rcu_pending <-update_process_times
102:    bash-2724   0dNh.   35us : rcu_check_callbacks <-update_process_times
103:    bash-2724   0dNh.   36us : idle_cpu <-rcu_check_callbacks
104:    bash-2724   0dNh.   36us : rcu_try_flip <-rcu_check_callbacks
105:    bash-2724   0dNh.   36us : rcupreempt_trace_try_flip_1 <-rcu_try_flip
106:    bash-2724   0dNh.   37us : _atomic_spin_trylock <-rcu_try_flip
107:    bash-2724   0dNh1   37us : rcupreempt_trace_try_flip_a1 <-rcu_try_flip
108:    bash-2724   0dNh1   37us : cpumask_next <-rcu_try_flip
109:    bash-2724   0dNh1   38us : cpumask_next <-rcu_try_flip
110:    bash-2724   0dNh1   38us : cpumask_next <-rcu_try_flip
111:    bash-2724   0dNh1   39us : rcupreempt_trace_try_flip_a2 <-rcu_try_flip
112:    bash-2724   0dNh1   39us : _atomic_spin_unlock_irqrestore <-rcu_try_flip
```

Listing 2: Interrupts Off Latency Trace (Part 2)

```
113:  bash-2724  0dNh.   40us : preempt_schedule <-_atomic_spin_unlock_irqrestore
114:  bash-2724  0dNh.   40us : _atomic_spin_lock_irqsave <-rcu_check_callbacks
115:  bash-2724  0dNh1   40us : rcupreempt_trace_check_callbacks <-rcu_check_callbacks
116:  bash-2724  0dNh1   41us : __rcu_advance_callbacks <-rcu_check_callbacks
117:  bash-2724  0dNh1   41us : _atomic_spin_unlock_irqrestore <-rcu_check_callbacks
118:  bash-2724  0dNh.   42us : preempt_schedule <-_atomic_spin_unlock_irqrestore
119:  bash-2724  0dNh.   42us : scheduler_tick <-update_process_times
120:  bash-2724  0dNh.   42us : ktime_get <-sched_clock_tick
121:  bash-2724  0dNh.   43us : clocksource_read <-ktime_get
122:  bash-2724  0dNh.   43us : _atomic_spin_lock <-scheduler_tick
123:  bash-2724  0dNh1   44us : update_rq_clock <-scheduler_tick
124:  bash-2724  0dNh1   44us : task_tick_fair <-scheduler_tick
125:  bash-2724  0dNh1   44us : update_curr <-task_tick_fair
126:  bash-2724  0dNh1   45us : calc_delta_fair <-update_curr
127:  bash-2724  0dNh1   45us : cpuacct_charge <-update_curr
128:  bash-2724  0dNh1   45us : __rcu_read_lock <-cpuacct_charge
129:  bash-2724  0dNh1   46us : __rcu_read_unlock <-cpuacct_charge
130:  bash-2724  0dNh1   46us : account_group_exec_runtime <-update_curr
131:  bash-2724  0dNh1   47us : _atomic_spin_unlock <-scheduler_tick
132:  bash-2724  0dNh.   48us : preempt_schedule <-_atomic_spin_unlock
133:  bash-2724  0dNh.   48us : perf_counter_task_tick <-scheduler_tick
134:  bash-2724  0dNh.   48us : find_new_ilb <-scheduler_tick
135:  bash-2724  0dNh.   49us : cpumask_first <-find_new_ilb
136:  bash-2724  0dNh.   49us : resched_cpu <-scheduler_tick
137:  bash-2724  0dNh.   49us : _atomic_spin_trylock <-resched_cpu
138:  bash-2724  0dNh1   50us : resched_task <-resched_cpu
139:  bash-2724  0dNh1   50us : test_tsk_need_resched <-resched_task
140:  bash-2724  0dNh1   51us : test_ti_thread_flag <-test_tsk_need_resched
141:  bash-2724  0dNh1   51us : _atomic_spin_unlock_irqrestore <-resched_cpu
142:  bash-2724  0dNh.   52us : preempt_schedule <-_atomic_spin_unlock_irqrestore
143:  bash-2724  0dNh.   52us : run_posix_cpu_timers <-update_process_times
144:  bash-2724  0dNh.   52us : profile_tick <-tick_sched_timer
145:  bash-2724  0dNh.   53us : hrtimer_forward <-tick_sched_timer
146:  bash-2724  0dNh.   53us : _atomic_spin_lock <-__run_hrtimer
147:  bash-2724  0dNh1   54us : enqueue_hrtimer <-__run_hrtimer
148:  bash-2724  0dNh1   54us : _atomic_spin_unlock <-hrtimer_interrupt
149:  bash-2724  0dNh.   55us : preempt_schedule <-_atomic_spin_unlock
150:  bash-2724  0dNh.   55us : tick_program_event <-hrtimer_interrupt
151:  bash-2724  0dNh.   55us : tick_dev_program_event <-tick_program_event
152:  bash-2724  0dNh.   56us : ktime_get <-tick_dev_program_event
153:  bash-2724  0dNh.   56us : clocksource_read <-ktime_get
154:  bash-2724  0dNh.   56us : clockevents_program_event <-tick_dev_program_event
155:  bash-2724  0dNh.   57us : lapic_next_event <-clockevents_program_event
156:  bash-2724  0dNh.   57us : apic_write <-lapic_next_event
157:  bash-2724  0dNh.   57us : native_apic_mem_write <-apic_write
158:  bash-2724  0dNh.   58us : irq_exit <-smp_apic_timer_interrupt
159:  bash-2724  0dN.1   58us : do_softirq <-irq_exit
160:  bash-2724  0dN.1   59us : __do_softirq <-call_softirq
161:  bash-2724  0dN.1   59us : wakeup_softirqd <-__do_softirq
162:  bash-2724  0dN.1   59us : rcu_irq_exit <-irq_exit
163:  bash-2724  0dN.1   60us : idle_cpu <-irq_exit
164:  bash-2724  0dN..   60us : preempt_schedule_irq <-retint_kernel
```

Listing 3: Interrupts Off Latency Trace (Part 3)

```
       bash-2724    0dN..    61us : __schedule <-preempt_schedule_irq
       bash-2724    0dN..    61us : rcu_qsctr_inc <-__schedule
       bash-2724    0dN.1    62us : _atomic_spin_lock_irq <-__schedule
       bash-2724    0dN.2    62us : update_rq_clock <-__schedule
       bash-2724    0d..2    63us : put_prev_task_fair <-__schedule
       bash-2724    0d..2    63us : update_curr <-put_prev_task_fair
       bash-2724    0d..2    63us : calc_delta_fair <-update_curr
       bash-2724    0d..2    64us : cpuacct_charge <-update_curr
       bash-2724    0d..2    64us : __rcu_read_lock <-cpuacct_charge
       bash-2724    0d..2    65us : __rcu_read_unlock <-cpuacct_charge
       bash-2724    0d..2    65us : account_group_exec_runtime <-update_curr
       bash-2724    0d..2    65us : __enqueue_entity <-put_prev_task_fair
       bash-2724    0d..2    66us : pick_next_task <-__schedule
       bash-2724    0d..2    66us : pick_next_task_rt <-pick_next_task
       bash-2724    0d..2    66us : dequeue_pushable_task <-pick_next_task_rt
       bash-2724    0d..2    67us : perf_counter_task_sched_out <-__schedule
       bash-2724    0d..2    68us : __unlazy_fpu <-__switch_to
   sirq-tim-5       0d..2    68us : finish_task_switch <-thread_return
   sirq-tim-5       0d..2    68us : needs_post_schedule_rt <-finish_task_switch
   sirq-tim-5       0d..2    69us : perf_counter_task_sched_in <-finish_task_switch
   sirq-tim-5       0d..2    69us : _atomic_spin_unlock <-finish_task_switch
   sirq-tim-5       0d...    70us : trace_hardirqs_on <-schedule
   sirq-tim-5       0d...    70us : time_hardirqs_on <-schedule
```

Listing 4: Interrupts Off Latency Trace (Part 4)

```
[root@mxf tracing]# echo 0 > /proc/sys/kernel/ftrace_enabled
[root@mxf tracing]# echo irqsoff > current_tracer
[root@mxf tracing]# cat tracing_max_latency
19
[root@mxf tracing]# cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 2.6.31-rc6-rt4
# --------------------------------------------------------------------
# latency: 19 us, #3/3, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#    -----------------
#    | task: gnome-settings--2669 (uid:42 nice:0 policy:0 rt_prio:0)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /
#                |||||     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||   \   |   /
gnome-se-2669    0d...    2us+: trace_hardirqs_off_thunk <-save_args
gnome-se-2669    0dN..   19us : trace_hardirqs_on_thunk <-retint_check
gnome-se-2669    0dN..   20us : time_hardirqs_on <-retint_check
```

Listing 5: Interrupts Off Latency Trace (No Functions)

```
[root@mxf tracing]# echo 0 > /proc/sys/kernel/ftrace_enabled
[root@mxf tracing]# echo 1 > events/sched/enable
[root@mxf tracing]# echo preemptirqsoff > current_tracer
[root@mxf tracing]# cat trace
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 2.6.31-rc6-rt4
# --------------------------------------------------------------------
# latency: 12 us, #6/6, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#    -----------------
#    | task: bash-2653 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /
#                |||||     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||   \   |   /
    bash-2653    1d...    1us+: trace_hardirqs_off <-task_rq_lock
    bash-2653    1d..2    3us+: sched_wakeup: task sshd:2650 [120] success=1
    bash-2653    1dNh3    9us+: sched_wakeup: task irq/21-uhci_hcd:580 [49] success=1
    bash-2653    1dNh3   11us+: sched_wakeup: task irq/21-eth0:2333 [49] success=1
    bash-2653    1.N.1   13us : _atomic_spin_unlock_irqrestore <-task_rq_unlock
    bash-2653    1.N.1   13us : trace_preempt_on <-task_rq_unlock
```
Listing 6: Preemption and Interrupts Off Latency Trace

```
[root@mxf tracing]# echo 0 > /proc/sys/kernel/ftrace_enabled
[root@mxf tracing]# echo 1 > events/sched/enable
[root@mxf tracing]# echo wakeup > current_tracer
[root@mxf tracing]# cat trace
# tracer: wakeup
#
# wakeup latency trace v1.1.5 on 2.6.31-rc6-rt4
# --------------------------------------------------------------------
# latency: 150 us, #7/7, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#    -----------------
#    | task: hald-addon-stor-2103 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /
#                |||||     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||   \   |   /
  <idle>-0       1d.h3    0us :      0:140:R   + [001]  2103:120:S hald-addon-stor
  <idle>-0       1d.h3    1us+: wake_up_process <-hrtimer_wakeup
  <idle>-0       1dNh3    3us+: sched_wakeup: task hald:1952 [120] success=1
  <idle>-0       1d..3    7us!: sched_switch: task swapper:0 [140] (R) ==> hald:1952 [120]
    hald-1952    1d..3  149us : sched_switch: task hald:1952 [120] (D) ==> hald-addon-stor:2103 [
    hald-1952    1d..3  150us : __schedule <-schedule
    hald-1952    1d..3  151us :   1952:120:S ==> [001]  2103:120:R hald-addon-stor
```

Listing 7: Scheduling Latency Trace

```
[root@mxf tracing]# echo 0 > /proc/sys/kernel/ftrace_enabled
[root@mxf tracing]# echo 1 > events/sched/enable
[root@mxf tracing]# echo wakeup_rt > current_tracer
[root@mxf tracing]# cat trace
# tracer: wakeup_rt
#
# wakeup_rt latency trace v1.1.5 on 2.6.31-rc6
# --------------------------------------------------------------------
# latency: 5 us, #5/5, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#    -----------------
#    | task: migration/0-3 (uid:0 nice:-5 policy:1 rt_prio:99)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /
#                |||||     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||   \   |   /
    bash-6808    0d..2    0us :   6808:120:R   + [000]     3:   0:S migration/0
    bash-6808    0d..2    1us+: wake_up_process <-sched_exec
    bash-6808    0d..3    4us : sched_switch: task bash:6808 [120] (R) ==> migration/0:3 [0]
    bash-6808    0d..3    5us : schedule <-preempt_schedule
    bash-6808    0d..3    5us :   6808:120:R ==> [000]     3:   0:R migration/0
```

Listing 8: Scheduling Latency Trace