ftrace - Function Tracer

Author: Steven Rostedt <srostedt@redhat.com>

Original Reviewers: Elias Oltmanns, Randy Dunlap, Andrew Morton, John Kacur, and David Teigland.

- Written for: 2.6.28-rc2
- Updated for: 3.10
- Updated for: 4.13 - Copyright 2017 VMware Inc. Steven Rostedt
- Converted to rst format - Changbin Du <changbin.du@intel.com>

Introduction

Ftrace is an internal tracer designed to help out developers and designers of systems to find what is going on inside the kernel. It can be used for debugging or analyzing latencies and performance issues that take place outside of user-space.

Although ftrace is typically considered the function tracer, it is really a framework of several assorted tracing utilities. There's latency tracing to examine what occurs between interrupts disabled and enabled, as well as for preemption and from a time a task is woken to the task is actually scheduled in.

One of the most common uses of ftrace is the event tracing. Throughout the kernel is hundreds of static event points that can be enabled via the tracefs file system to see what is going on in certain parts of the kernel.

See events.rst for more information.

Implementation Details

See Documentation/trace/ftrace-design.rst for details for arch porters and such.

The File System

Ftrace uses the tracefs file system to hold the control files as well as the files to display output.

When tracefs is configured into the kernel (which selecting any ftrace option will do) the directory /sys/kernel/tracing will be created. To mount this directory, you can add to your /etc/fstab file:

```
tracefs       /sys/kernel/tracing       tracefs defaults        0       0
```

Or you can mount it at run time with:

```
mount -t tracefs nodev /sys/kernel/tracing
```

For quicker access to that directory you may want to make a soft link to it:

```
ln -s /sys/kernel/tracing /tracing
```

> **Attention!**
>
> Before 4.1, all ftrace tracing control files were within the debugfs file system, which is typically located at /sys/kernel/debug/tracing. For backward compatibility, when mounting the debugfs file system, the tracefs file system will be automatically mounted at:
>
> /sys/kernel/debug/tracing
>
> All files located in the tracefs file system will be located in that debugfs file system directory as well.

> **Attention!**
>
> Any selected ftrace option will also create the tracefs file system. The rest of the document will assume that you are in the ftrace directory (cd /sys/kernel/tracing) and will only concentrate on the files within that directory and not distract from the content with the extended "/sys/kernel/tracing" path name.

That's it! (assuming that you have ftrace configured into your kernel)

After mounting tracefs you will have access to the control and output files of ftrace. Here is a list of some of the key files:

Note: all time values are in microseconds.

current_tracer:

This is used to set or display the current tracer that is configured. Changing the current tracer clears the ring buffer content as well as the "snapshot" buffer.

available_tracers:

This holds the different types of tracers that have been compiled into the kernel. The tracers listed here can be configured by echoing their name into current_tracer.

tracing_on:

This sets or displays whether writing to the trace ring buffer is enabled. Echo 0 into this file to disable the tracer or 1 to enable it. Note, this only disables writing to the ring buffer, the tracing overhead may still be occurring.

The kernel function tracing_off() can be used within the kernel to disable writing to the ring buffer, which will set this file to "0". User space can re-enable tracing by echoing "1" into the file.

Note, the function and event trigger "traceoff" will also set this file to zero and stop tracing. Which can also be re-enabled by user space using this file.

trace:

This file holds the output of the trace in a human readable format (described below). Opening this file for writing with the O_TRUNC flag clears the ring buffer content. Note, this file is not a consumer. If tracing is off (no tracer running, or tracing_on is zero), it will produce the same output each time it is read. When tracing is on, it may produce inconsistent results as it tries to read the entire buffer without consuming it.

trace_pipe:

The output is the same as the "trace" file but this file is meant to be streamed with live tracing. Reads from this file will block until new data is retrieved. Unlike the "trace" file, this file is a consumer. This means reading from this file causes sequential reads to display more current data. Once data is read from this file, it is consumed, and will not be read again with a sequential read. The "trace" file is static, and if the tracer is not adding more data, it will display the same information every time it is read.

trace_options:

This file lets the user control the amount of data that is displayed in one of the above output files. Options also exist to modify how a tracer or events work (stack traces, timestamps, etc).

options:

This is a directory that has a file for every available trace option (also in trace_options). Options may also be set or cleared by writing a "1" or "0" respectively into the corresponding file with the option name.

tracing_max_latency:

Some of the tracers record the max latency. For example, the maximum time that interrupts are disabled. The maximum time is saved in this file. The max trace will also be stored, and displayed by "trace". A new max trace will only be recorded if the latency is greater than the value in this file (in microseconds).

By echoing in a time into this file, no latency will be recorded unless it is greater than the time in this file.

tracing_thresh:

Some latency tracers will record a trace whenever the latency is greater than the number in this file. Only active when the file contains a number greater than 0. (in microseconds)

buffer_size_kb:

This sets or displays the number of kilobytes each CPU buffer holds. By default, the trace buffers are the same size for each CPU. The displayed number is the size of the CPU buffer and not total size of all buffers. The trace buffers are allocated in pages (blocks of memory that the kernel uses for allocation, usually 4 KB in size). A few extra pages may be allocated to accommodate buffer management meta-data. If the last page allocated has room for more bytes than requested, the rest of the page will be used, making the actual allocation bigger than requested or shown. (Note, the size may not be a multiple of the page size due to buffer management meta-data.)

Buffer sizes for individual CPUs may vary (see "per_cpu/cpu0/buffer_size_kb" below), and if they do this file will show "X".

buffer_total_size_kb:

This displays the total combined size of all the trace buffers.

free_buffer:

If a process is performing tracing, and the ring buffer should be shrunk "freed" when the process is finished, even if it were to be killed by a signal, this file can be used for that purpose. On close of this file, the ring buffer will be resized to its minimum size. Having a process that is tracing also open this file, when the process exits its file descriptor for this file will be closed, and in doing so, the ring buffer will be "freed".

It may also stop tracing if disable_on_free option is set.

tracing_cpumask:

This is a mask that lets the user only trace on specified CPUs. The format is a hex string representing the CPUs.

set_ftrace_filter:

When dynamic ftrace is configured in (see the section below "dynamic ftrace"), the code is dynamically modified (code text rewrite) to disable calling of the function profiler (mcount). This lets tracing be configured in with practically no overhead in performance. This also has a side effect of enabling or disabling specific functions to be traced. Echoing names of functions into this file will limit the trace to only those functions. This influences the tracers "function" and "function_graph" and thus also function profiling (see "function_profile_enabled").

The functions listed in "available_filter_functions" are what can be written into this file.

This interface also allows for commands to be used. See the "Filter commands" section for more details.

As a speed up, since processing strings can be quite expensive and requires a check of all functions registered to tracing, instead an index can be written into this file. A number (starting with "1") written will instead select the same corresponding at the line position of the "available_filter_functions" file.

set_ftrace_notrace:

This has an effect opposite to that of set_ftrace_filter. Any function that is added here will not be traced. If a function exists in both set_ftrace_filter and set_ftrace_notrace, the function will _not_ be traced.

set_ftrace_pid:

Have the function tracer only trace the threads whose PID are listed in this file.

If the "function-fork" option is set, then when a task whose PID is listed in this file forks, the child's PID will automatically be added to this file, and the child will be traced by the function tracer as well. This option will also cause PIDs of tasks that exit to be removed from the file.

set_ftrace_notrace_pid:

Have the function tracer ignore threads whose PID are listed in this file.

If the "function-fork" option is set, then when a task whose PID is listed in this file forks, the child's PID will automatically be added to this file, and the child will not be traced by the function tracer as well. This option will also cause PIDs of tasks that exit to be removed from the file.

If a PID is in both this file and "set_ftrace_pid", then this file takes precedence, and the thread will not be traced.

set_event_pid:

Have the events only trace a task with a PID listed in this file. Note, sched_switch and sched_wake_up will also trace events listed in this file.

To have the PIDs of children of tasks with their PID in this file added on fork, enable the "event-fork" option. That option will also cause the PIDs of tasks to be removed from this file when the task exits.

set_event_notrace_pid:

Have the events not trace a task with a PID listed in this file. Note, sched_switch and sched_wakeup will trace threads not listed in this file, even if a thread's PID is in the file if the sched_switch or sched_wakeup events also trace a thread that should be traced.

To have the PIDs of children of tasks with their PID in this file added on fork, enable the "event-fork" option. That option will also cause the PIDs of tasks to be removed from this file when the task exits.

set_graph_function:

Functions listed in this file will cause the function graph tracer to only trace these functions and the functions that they call. (See the section "dynamic ftrace" for more details). Note, set_ftrace_filter and set_ftrace_notrace still affects what functions are being traced.

set_graph_notrace:

Similar to set_graph_function, but will disable function graph tracing when the function is hit until it exits the function. This makes it possible to ignore tracing functions that are called by a specific function.

available_filter_functions:

This lists the functions that ftrace has processed and can trace. These are the function names that you can pass to "set_ftrace_filter", "set_ftrace_notrace", "set_graph_function", or "set_graph_notrace". (See the section "dynamic ftrace" below for more details.)

dyn_ftrace_total_info:

This file is for debugging purposes. The number of functions that have been converted to nops and are available to be traced.

enabled_functions:

This file is more for debugging ftrace, but can also be useful in seeing if any function has a callback attached to it. Not only does the trace infrastructure use ftrace function trace utility, but other subsystems might too. This file displays all functions that have a callback attached to them as well as the number of callbacks that have been attached. Note, a callback may also call multiple functions which will not be listed in this count.

If the callback registered to be traced by a function with the "save regs" attribute (thus even more overhead), a 'R' will be displayed on the same line as the function that is returning registers.

If the callback registered to be traced by a function with the "ip modify" attribute (thus the regs->ip can be changed), an 'I' will be displayed on the same line as the function that can be overridden.

If the architecture supports it, it will also show what callback is being directly called by the function. If the count is greater than 1 it most likely will be ftrace_ops_list_func().

If the callback of a function jumps to a trampoline that is specific to the callback and which is not the standard trampoline, its address will be printed as well as the function that the trampoline calls.

function_profile_enabled:

When set it will enable all functions with either the function tracer, or if configured, the function graph tracer. It will keep a histogram of the number of functions that were called and if the function graph tracer was configured, it will also keep track of the time spent in those functions. The histogram content can be displayed in the files:

trace_stat/function<cpu> (function0, function1, etc).

trace_stat:

A directory that holds different tracing stats.

kprobe_events:

Enable dynamic trace points. See kprobetrace.rst.

kprobe_profile:

Dynamic trace points stats. See kprobetrace.rst.

max_graph_depth:

Used with the function graph tracer. This is the max depth it will trace into a function. Setting this to a value of one will show only the first kernel function that is called from user space.

printk_formats:

This is for tools that read the raw format files. If an event in the ring buffer references a string, only a pointer to the string is recorded into the buffer and not the string itself. This prevents tools from knowing what that string was. This file displays the string and address for the string allowing tools to map the pointers to what the strings were.

saved_cmdlines:

Only the pid of the task is recorded in a trace event unless the event specifically saves the task comm as well. Ftrace makes a cache of pid mappings to comms to try to display comms for events. If a pid for a comm is not listed, then "<...>" is displayed in the output.

If the option "record-cmd" is set to "0", then comms of tasks will not be saved during recording. By default, it is enabled.

saved_cmdlines_size:

By default, 128 comms are saved (see "saved_cmdlines" above). To increase or decrease the amount of comms that are cached, echo the number of comms to cache into this file.

saved_tgids:

If the option "record-tgid" is set, on each scheduling context switch the Task Group ID of a task is saved in a table mapping the PID of the thread to its TGID. By default, the "record-tgid" option is disabled.

snapshot:

This displays the "snapshot" buffer and also lets the user take a snapshot of the current running trace. See the "Snapshot" section below for more details.

stack_max_size:

When the stack tracer is activated, this will display the maximum stack size it has encountered. See the "Stack Trace" section below.

stack_trace:

This displays the stack back trace of the largest stack that was encountered when the stack tracer is activated. See the "Stack Trace" section below.

stack_trace_filter:

This is similar to "set_ftrace_filter" but it limits what functions the stack tracer will check.

trace_clock:

Whenever an event is recorded into the ring buffer, a "timestamp" is added. This stamp comes from a specified clock. By default, ftrace uses the "local" clock. This clock is very fast and strictly per cpu, but on some systems it may not be monotonic with respect to other CPUs. In other words, the local clocks may not be in sync with local clocks on other CPUs.

Usual clocks for tracing:

```
# cat trace_clock
[local] global counter x86-tsc
```

The clock with the square brackets around it is the one in effect.

local:
: Default clock, but may not be in sync across CPUs

global:
: This clock is in sync with all CPUs but may be a bit slower than the local clock.

counter:
: This is not a clock at all, but literally an atomic counter. It counts up one by one, but is in sync with all CPUs. This is useful when you need to know exactly the order events occurred with respect to each other on different CPUs.

uptime:
: This uses the jiffies counter and the time stamp is relative to the time since boot up.

perf:
: This makes ftrace use the same clock that perf uses. Eventually perf will be able to read ftrace buffers and this will help out in interleaving the data.

x86-tsc:
: Architectures may define their own clocks. For example, x86 uses its own TSC cycle clock here.

ppc-tb:
: This uses the powerpc timebase register value. This is in sync across CPUs and can also be used to correlate events across hypervisor/guest if tb_offset is known.

mono:
: This uses the fast monotonic clock (CLOCK_MONOTONIC) which is monotonic and is subject to NTP rate adjustments.

mono_raw:
: This is the raw monotonic clock (CLOCK_MONOTONIC_RAW) which is monotonic but is not subject to any rate adjustments and ticks at the same rate as the hardware clocksource.

boot:
: This is the boot clock (CLOCK_BOOTTIME) and is based on the fast monotonic clock, but also accounts for time spent in suspend. Since the clock access is designed for use in tracing in the suspend path, some side effects are possible if clock is accessed after the suspend time is accounted before the fast mono clock is updated. In this case, the clock update appears to happen slightly sooner than it normally would have. Also on 32-bit systems, it's possible that the 64-bit boot offset sees a partial update. These effects are rare and post processing should be able to handle them. See comments in the ktime_get_boot_fast_ns() function for more information.

To set a clock, simply echo the clock name into this file:

```
# echo global > trace_clock
```

Setting a clock clears the ring buffer content as well as the "snapshot" buffer.

trace_marker:

This is a very useful file for synchronizing user space with events happening in the kernel. Writing strings into this file will be written into the ftrace buffer.

It is useful in applications to open this file at the start of the application and just reference the file descriptor for the file:

```
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);
}
```

start:

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

Note: Writing into the trace_marker file can also initiate triggers
: that are written into /sys/kernel/tracing/events/ftrace/print/trigger See "Event triggers" in Documentation/trace/events.rst and an example in Documentation/trace/histogram.rst (Section 3.)

trace_marker_raw:

This is similar to trace_marker above, but is meant for binary data to be written to it, where a tool can be used to parse the data from trace_pipe_raw.

uprobe_events:

Add dynamic tracepoints in programs. See uprobetracer.rst

uprobe_profile:

Uprobe statistics. See uprobetrace.txt

instances:

This is a way to make multiple trace buffers where different events can be recorded in different buffers. See "Instances" section below.

events:

This is the trace event directory. It holds event tracepoints (also known as static tracepoints) that have been compiled into the kernel. It shows what event tracepoints exist and how they are grouped by system. There are "enable" files at various levels that can enable the tracepoints when a "1" is written to them.

See events.rst for more information.

set_event:

By echoing in the event into this file, will enable that event.

See events.rst for more information.

available_events:

A list of events that can be enabled in tracing.

See events.rst for more information.

timestamp_mode:

Certain tracers may change the timestamp mode used when logging trace events into the event buffer. Events with different modes can coexist within a buffer but the mode in effect when an event is logged determines which timestamp mode is used for that event. The default timestamp mode is 'delta'.

Usual timestamp modes for tracing:

cat timestamp_mode [delta] absolute

The timestamp mode with the square brackets around it is the one in effect.

delta: Default timestamp mode - timestamp is a delta against
a per-buffer timestamp.

absolute: The timestamp is a full timestamp, not a delta
against some other value. As such it takes up more space and is less efficient.

hwlat_detector:

Directory for the Hardware Latency Detector. See "Hardware Latency Detector" section below.

per_cpu:

This is a directory that contains the trace per_cpu information.

per_cpu/cpu0/buffer_size_kb:

The ftrace buffer is defined per_cpu. That is, there's a separate buffer for each CPU to allow writes to be done atomically, and free from cache bouncing. These buffers may have different size buffers. This file is similar to the buffer_size_kb file, but it only displays or sets the buffer size for the specific CPU. (here cpu0).

per_cpu/cpu0/trace:

This is similar to the "trace" file, but it will only display the data specific for the CPU. If written to, it only clears the specific CPU buffer.

per_cpu/cpu0/trace_pipe

This is similar to the "trace_pipe" file, and is a consuming read, but it will only display (and consume) the data specific for the CPU.

per_cpu/cpu0/trace_pipe_raw

For tools that can parse the ftrace ring buffer binary format, the trace_pipe_raw file can be used to extract the data from the ring buffer directly. With the use of the splice() system call, the buffer data can be quickly transferred to a file or to the network where a server is collecting the data.

Like trace_pipe, this is a consuming reader, where multiple reads will always produce different data.

per_cpu/cpu0/snapshot:

This is similar to the main "snapshot" file, but will only snapshot the current CPU (if supported). It only displays the content of the snapshot for a given CPU, and if written to, only clears this CPU buffer.

per_cpu/cpu0/snapshot_raw:

Similar to the trace_pipe_raw, but will read the binary format from the snapshot buffer for the given CPU.

per_cpu/cpu0/stats:

This displays certain stats about the ring buffer:

entries:
The number of events that are still in the buffer.

overrun:
The number of lost events due to overwriting when the buffer was full.

commit overrun:
Should always be zero. This gets set if so many events happened within a nested event (ring buffer is re-entrant), that it fills the buffer and starts dropping events.

bytes:
Bytes actually read (not overwritten).

oldest event ts:
The oldest timestamp in the buffer

now ts:
The current timestamp

dropped events:
Events lost due to overwrite option being off.

read events:
The number of events read.

The Tracers

Here is the list of current tracers that may be configured.

"function"

Function call tracer to trace all kernel functions.

"function_graph"

Similar to the function tracer except that the function tracer probes the functions on their entry whereas the function graph tracer traces on both entry and exit of the functions. It then provides the ability to draw a graph of function calls similar to C code source.

"blk"

The block tracer. The tracer used by the blktrace user application.

"hwlat"

The Hardware Latency tracer is used to detect if the hardware produces any latency. See "Hardware Latency Detector" section below.

"irqsoff"

Traces the areas that disable interrupts and saves the trace with the longest max latency. See tracing_max_latency. When a new max is recorded, it replaces the old trace. It is best to view this trace with the latency-format option enabled, which happens automatically when the tracer is selected.

"preemptoff"

Similar to irqsoff but traces and records the amount of time for which preemption is disabled.

"preemptirqsoff"

Similar to irqsoff and preemptoff, but traces and records the largest time for which irqs and/or preemption is disabled.

"wakeup"

Traces and records the max latency that it takes for the highest priority task to get scheduled after it has been woken up. Traces all tasks as an average developer would expect.

"wakeup_rt"

Traces and records the max latency that it takes for just RT tasks (as the current "wakeup" does). This is useful for those interested in wake up timings of RT tasks.

"wakeup_dl"

Traces and records the max latency that it takes for a SCHED_DEADLINE task to be woken (as the "wakeup" and "wakeup_rt" does).

"mmiotrace"

A special tracer that is used to trace binary module. It will trace all the calls that a module makes to the hardware. Everything it writes and reads from the I/O as well.

"branch"

This tracer can be configured when tracing likely/unlikely calls within the kernel. It will trace when a likely and unlikely branch is hit and if it was correct in its prediction of being correct.

"nop"

This is the "trace nothing" tracer. To remove all tracers from tracing simply echo "nop" into current_tracer.

Error conditions

For most ftrace commands, failure modes are obvious and communicated using standard return codes.

For other more involved commands, extended error information may be available via the tracing/error_log file. For the commands that support it, reading the tracing/error_log file after an error will display more detailed information about what went wrong, if information is available. The tracing/error_log file is a circular error log displaying a small number (currently, 8) of ftrace errors for the last (8) failed commands.

The extended error information and usage takes the form shown in this example:

```
# echo xxx > /sys/kernel/debug/tracing/events/sched/sched_wakeup/trigger
echo: write error: Invalid argument

# cat /sys/kernel/debug/tracing/error_log
[ 5348.887237] location: error: Couldn't yyy: zzz
  Command: xxx
           ^
[ 7517.023364] location: error: Bad rrr: sss
  Command: ppp qqq
               ^
```

To clear the error log, echo the empty string into it:

```
# echo > /sys/kernel/debug/tracing/error_log
```

Examples of using the tracer

Here are typical examples of using the tracers when controlling them only with the tracefs interface (without using any user-land utilities).

Output format:

Here is an example of the output format of the file "trace":

```
# tracer: function
#
# entries-in-buffer/entries-written: 140080/250280   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
            bash-1977  [000] .... 17284.993652: sys_close <-system_call_fastpath
            bash-1977  [000] .... 17284.993653: __close_fd <-sys_close
            bash-1977  [000] .... 17284.993653: _raw_spin_lock <-__close_fd
            sshd-1974  [003] .... 17284.993653: __srcu_read_unlock <-fsnotify
            bash-1977  [000] .... 17284.993654: add_preempt_count <-_raw_spin_lock
            bash-1977  [000] ...1 17284.993655: _raw_spin_unlock <-__close_fd
            bash-1977  [000] ...1 17284.993656: sub_preempt_count <-_raw_spin_unlock
            bash-1977  [000] .... 17284.993657: filp_close <-__close_fd
            bash-1977  [000] .... 17284.993657: dnotify_flush <-filp_close
            sshd-1974  [003] .... 17284.993658: sys_select <-system_call_fastpath
            ....
```

A header is printed with the tracer name that is represented by the trace. In this case the tracer is "function". Then it shows the number of events in the buffer as well as the total number of entries that were written. The difference is the number of entries that were lost due to the buffer filling up (250280 - 140080 = 110200 events lost).

The header explains the content of the events. Task name "bash", the task PID "1977", the CPU that it was running on "000", the latency format (explained below), the timestamp in <secs>.<usecs> format, the function name that was traced "sys_close" and the parent function that called this function "system_call_fastpath". The timestamp is the time at which the function was entered.

Latency trace format

When the latency-format option is enabled or when one of the latency tracers is set, the trace file gives somewhat more information to see why a latency happened. Here is a typical trace:

```
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.8.0-test+
```

```
# -----------------------------------------------------------------------
# latency: 259 us, #4/4, CPU#2 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: ps-6143 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: __lock_task_sighand
#  => ended at:   _raw_spin_unlock_irqrestore
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
      ps-6143    2d...    0us!: trace_hardirqs_off <-__lock_task_sighand
      ps-6143    2d..1  259us+: trace_hardirqs_on <-_raw_spin_unlock_irqrestore
      ps-6143    2d..1  263us+: time_hardirqs_on <-_raw_spin_unlock_irqrestore
      ps-6143    2d..1  306us : <stack trace>
 => trace_hardirqs_on_caller
 => trace_hardirqs_on
 => _raw_spin_unlock_irqrestore
 => do_task_stat
 => proc_tgid_stat
 => proc_single_show
 => seq_read
 => vfs_read
 => sys_read
 => system_call_fastpath
```

This shows that the current tracer is "irqsoff" tracing the time for which interrupts were disabled. It gives the trace version (which never changes) and the version of the kernel upon which this was executed on (3.8). Then it displays the max latency in microseconds (259 us). The number of trace entries displayed and the total number (both are four: #4/4). VP, KP, SP, and HP are always zero and are reserved for later use. #P is the number of online CPUs (#P:4).

The task is the process that was running when the latency occurred. (ps pid: 6143).

The start and stop (the functions in which the interrupts were disabled and enabled respectively) that caused the latencies:

- __lock_task_sighand is where the interrupts were disabled.
- _raw_spin_unlock_irqrestore is where they were enabled again.

The next lines after the header are the trace itself. The header explains which is which.

cmd: The name of the process in the trace.

pid: The PID of that process.

CPU#: The CPU which the process was running on.

irqs-off: 'd' interrupts are disabled. '.' otherwise.

> **Caution!**
>
> If the architecture does not support a way to read the irq flags variable, an 'X' will always be printed here.

need-resched:

- 'N' both TIF_NEED_RESCHED and PREEMPT_NEED_RESCHED is set,
- 'n' only TIF_NEED_RESCHED is set,
- 'p' only PREEMPT_NEED_RESCHED is set,
- '.' otherwise.

hardirq/softirq:

- 'Z' - NMI occurred inside a hardirq
- 'z' - NMI is running
- 'H' - hard irq occurred inside a softirq.
- 'h' - hard irq is running
- 's' - soft irq is running
- '.' - normal context.

preempt-depth: The level of preempt_disabled

The above is mostly meaningful for kernel developers.

time:

When the latency-format option is enabled, the trace file output includes a timestamp relative to the start of the trace. This differs from the output when latency-format is disabled, which includes an absolute timestamp.

delay:

This is just to help catch your eye a bit better. And needs to be fixed to be only relative to the same CPU. The marks are determined by the difference between this current trace and the next trace.

- '$' - greater than 1 second
- '@' - greater than 100 millisecond
- '*' - greater than 10 millisecond
- '#' - greater than 1000 microsecond
- '!' - greater than 100 microsecond
- '+' - greater than 10 microsecond
- ' ' - less than or equal to 10 microsecond.

The rest is the same as the 'trace' file.

Note, the latency tracers will usually end with a back trace to easily find where the latency occurred.

trace_options

The trace_options file (or the options directory) is used to control what gets printed in the trace output, or manipulate the tracers. To see what is available, simply cat the file:

```
cat trace_options
      print-parent
      nosym-offset
      nosym-addr
      noverbose
      noraw
      nohex
      nobin
      noblock
      trace_printk
      annotate
      nouserstacktrace
      nosym-userobj
      noprintk-msg-only
      context-info
      nolatency-format
      record-cmd
      norecord-tgid
```

```
        overwrite
        nodisable_on_free
        irq-info
        markers
        noevent-fork
        function-trace
        nofunction-fork
        nodisplay-graph
        nostacktrace
        nobranch
```

To disable one of the options, echo in the option prepended with "no":

```
echo noprint-parent > trace_options
```

To enable an option, leave off the "no":

```
echo sym-offset > trace_options
```

Here are the available options:

print-parent

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

```
print-parent:
 bash-4000  [01]  1477.606694: simple_strtoul <-kstrtoul

noprint-parent:
 bash-4000  [01]  1477.606694: simple_strtoul
```

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-4000  [01]  1477.606694: simple_strtoul+0x6/0xa0
```

sym-addr

: This will also display the function address as well as the function name.

```
sym-addr:
 bash-4000  [01]  1477.606694: simple_strtoul <c0339346>
```

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 (kstrtoul)
```

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.

hex

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

bin

: This will print out the formats in raw binary.

block

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

trace_printk

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

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     [001] dNs4 21169.031481: wake_up_idle_cpu <-add_timer_on
          <idle>-0     [001] dNs4 21169.031482: _raw_spin_unlock_irqrestore <-add_timer_on
          <idle>-0     [001] .Ns4 21169.031484: sub_preempt_count <-_raw_spin_unlock_irqrestore
##### CPU 2 buffer started ####
          <idle>-0     [002] .N.1 21169.031484: rcu_idle_exit <-cpu_idle
          <idle>-0     [001] .Ns3 21169.031484: _raw_spin_unlock <-clocksource_watchdog
          <idle>-0     [001] .Ns3 21169.031485: sub_preempt_count <-_raw_spin_unlock
```

userstacktrace

: This option changes the trace. It records a stacktrace of the current user space thread after each trace event.

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[+0
x494] <- /root/a.out[+0x4a8] <- /lib/libc-2.7.so[+0x1e1a6]
```

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()).

context-info

: Show only the event data. Hides the comm, PID, timestamp, CPU, and other useful data.

latency-format

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

pause-on-trace

: When set, opening the trace file for read, will pause writing to the ring buffer (as if tracing_on was set to zero). This simulates the original behavior of the trace file. When the file is closed, tracing will be enabled again.

hash-ptr

: When set, "%p" in the event printk format displays the hashed pointer value instead of real address. This will be useful if you want to find out which hashed value is corresponding to the real value in trace log.

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. See "saved_cmdlines".

record-tgid

When any event or tracer is enabled, a hook is enabled in the sched_switch trace point to fill the cache of mapped Thread Group IDs (TGID) mapping to pids. See "saved_tgids".

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)

disable_on_free

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

irq-info

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

```
# tracer: function
#
# entries-in-buffer/entries-written: 144405/9452052   #P:4
#
#           TASK-PID   CPU#      TIMESTAMP  FUNCTION
#              | |       |          |         |
          <idle>-0     [002]  23636.756054: ttwu_do_activate.constprop.89 <-try_to_wake_up
          <idle>-0     [002]  23636.756054: activate_task <-ttwu_do_activate.constprop.89
          <idle>-0     [002]  23636.756055: enqueue_task <-activate_task
```

markers

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

event-fork

When set, tasks with PIDs listed in set_event_pid will have the PIDs of their children added to set_event_pid when those tasks fork. Also, when tasks with PIDs in set_event_pid exit, their PIDs will be removed from the file.

This affects PIDs listed in set_event_notrace_pid as well.

function-trace

The latency tracers will enable function tracing if this option is enabled (default it is). When it is disabled, the latency tracers do not trace functions. This keeps the overhead of the tracer down when performing latency tests.

function-fork

When set, tasks with PIDs listed in set_ftrace_pid will have the PIDs of their children added to set_ftrace_pid when those tasks fork. Also, when tasks with PIDs in set_ftrace_pid exit, their PIDs will be removed from the file.

This affects PIDs in set_ftrace_notrace_pid as well.

display-graph

When set, the latency tracers (irqsoff, wakeup, etc) will use function graph tracing instead of function tracing.

stacktrace

When set, a stack trace is recorded after any trace event is recorded.

branch

Enable branch tracing with the tracer. This enables branch tracer along with the currently set tracer. Enabling this with the "nop" tracer is the same as just enabling the "branch" tracer.

> **Tip**
>
> Some tracers have their own options. They only appear in this file when the tracer is active. They always appear in the options directory.

Here are the per tracer options:

Options for function tracer:

func_stack_trace
: When set, a stack trace is recorded after every function that is recorded. NOTE! Limit the functions that are recorded before enabling this, with "set_ftrace_filter" otherwise the system performance will be critically degraded. Remember to disable this option before clearing the function filter.

Options for function_graph tracer:

Since the function_graph tracer has a slightly different output it has its own options to control what is displayed.

funcgraph-overrun
: When set, the "overrun" of the graph stack is displayed after each function traced. The overrun, is when the stack depth of the calls is greater than what is reserved for each task. Each task has a fixed array of functions to trace in the call graph. If the depth of the calls exceeds that, the function is not traced. The overrun is the number of functions missed due to exceeding this array.

funcgraph-cpu
: When set, the CPU number of the CPU where the trace occurred is displayed.

funcgraph-overhead
: When set, if the function takes longer than A certain amount, then a delay marker is displayed. See "delay" above, under the header description.

funcgraph-proc
: Unlike other tracers, the process' command line is not displayed by default, but instead only when a task is traced in and out during a context switch. Enabling this options has the command of each process displayed at every line.

funcgraph-duration
: At the end of each function (the return) the duration of the amount of time in the function is displayed in microseconds.

funcgraph-abstime
: When set, the timestamp is displayed at each line.

funcgraph-irqs
: When disabled, functions that happen inside an interrupt will not be traced.

funcgraph-tail
: When set, the return event will include the function that it represents. By default this is off, and only a closing curly bracket "}" is displayed for the return of a function.

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.

graph-time
: When running function profiler with 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.

Options for blk tracer:

blk_classic
: Shows a more minimalistic output.

irqsoff

When interrupts are disabled, the CPU can not react to any other external event (besides NMIs and SMIs). This prevents the timer interrupt from triggering or the mouse interrupt from letting the kernel know of a new mouse event. The result is a latency with the reaction time.

The irqsoff tracer tracks the time for which interrupts are disabled. When a new maximum latency is hit, the tracer saves the trace leading up to that latency point so that every time a new maximum is reached, the old saved trace is discarded and the new trace is saved.

To reset the maximum, echo 0 into tracing_max_latency. Here is an example:

```
# echo 0 > options/function-trace
# echo irqsoff > current_tracer
# echo 1 > tracing_on
# echo 0 > tracing_max_latency
# ls -ltr
[...]
# echo 0 > tracing_on
# cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.8.0-test+
# --------------------------------------------------------------------
# latency: 16 us, #4/4, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: swapper/0-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: run_timer_softirq
#  => ended at:   run_timer_softirq
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>-0       0d.s2    0us+: _raw_spin_lock_irq <-run_timer_softirq
  <idle>-0       0dNs3   17us : _raw_spin_unlock_irq <-run_timer_softirq
  <idle>-0       0dNs3   17us+: trace_hardirqs_on <-run_timer_softirq
  <idle>-0       0dNs3   25us : <stack trace>
 => _raw_spin_unlock_irq
 => run_timer_softirq
 => __do_softirq
 => call_softirq
 => do_softirq
 => irq_exit
 => smp_apic_timer_interrupt
 => apic_timer_interrupt
 => rcu_idle_exit
 => cpu_idle
 => rest_init
 => start_kernel
 => x86_64_start_reservations
 => x86_64_start_kernel
```

Here we see that we had a latency of 16 microseconds (which is very good). The _raw_spin_lock_irq in run_timer_softirq disabled interrupts. The difference between the 16 and the displayed timestamp 25us occurred because the clock was incremented between the time of recording the max latency and the time of recording the function that had that latency.

Note the above example had function-trace not set. If we set function-trace, we get a much larger output:

```
with echo 1 > options/function-trace

 # tracer: irqsoff
 #
 # irqsoff latency trace v1.1.5 on 3.8.0-test+
 # --------------------------------------------------------------------
 # latency: 71 us, #168/168, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
 #    -----------------
 #    | task: bash-2042 (uid:0 nice:0 policy:0 rt_prio:0)
 #    -----------------
 #  => started at: ata_scsi_queuecmd
 #  => ended at:   ata_scsi_queuecmd
 #
 #
 #                  _------=> CPU#
 #                 / _-----=> irqs-off
 #                | / _----=> need-resched
 #                || / _---=> hardirq/softirq
 #                ||| / _--=> preempt-depth
 #                |||| /     delay
 #  cmd     pid   ||||| time  |   caller
 #     \   /      |||||  \    |   /
     bash-2042    3d...    0us : _raw_spin_lock_irqsave <-ata_scsi_queuecmd
     bash-2042    3d...    0us : add_preempt_count <-_raw_spin_lock_irqsave
     bash-2042    3d..1    1us : ata_scsi_find_dev <-ata_scsi_queuecmd
     bash-2042    3d..1    1us : __ata_scsi_find_dev <-ata_scsi_find_dev
     bash-2042    3d..1    2us : ata_find_dev.part.14 <-__ata_scsi_find_dev
     bash-2042    3d..1    2us : ata_qc_new_init <-__ata_scsi_queuecmd
     bash-2042    3d..1    3us : ata_sg_init <-__ata_scsi_queuecmd
     bash-2042    3d..1    4us : ata_scsi_rw_xlat <-__ata_scsi_queuecmd
     bash-2042    3d..1    4us : ata_build_rw_tf <-ata_scsi_rw_xlat
 [...]
     bash-2042    3d..1   67us : delay_tsc <-__delay
     bash-2042    3d..1   67us : add_preempt_count <-delay_tsc
     bash-2042    3d..2   67us : sub_preempt_count <-delay_tsc
     bash-2042    3d..1   67us : add_preempt_count <-delay_tsc
     bash-2042    3d..2   68us : sub_preempt_count <-delay_tsc
     bash-2042    3d..1   68us+: ata_bmdma_start <-ata_bmdma_qc_issue
     bash-2042    3d..1   71us : _raw_spin_unlock_irqrestore <-ata_scsi_queuecmd
     bash-2042    3d..1   71us : _raw_spin_unlock_irqrestore <-ata_scsi_queuecmd
     bash-2042    3d..1   72us+: trace_hardirqs_on <-ata_scsi_queuecmd
     bash-2042    3d..1  120us : <stack trace>
  => _raw_spin_unlock_irqrestore
  => ata_scsi_queuecmd
  => scsi_dispatch_cmd
  => scsi_request_fn
  => __blk_run_queue_uncond
  => __blk_run_queue
  => blk_queue_bio
  => submit_bio_noacct
  => submit_bio
  => submit_bh
  => __ext3_get_inode_loc
  => ext3_iget
  => ext3_lookup
  => lookup_real
  => __lookup_hash
  => walk_component
  => lookup_last
  => path_lookupat
  => filename_lookup
  => user_path_at_empty
  => user_path_at
  => vfs_fstatat
  => vfs_stat
  => sys_newstat
  => system_call_fastpath
```

Here we traced a 71 microsecond latency. But we also see all the functions that were called during that time. Note that by enabling function tracing, we incur an added overhead. This overhead may extend the latency times. But nevertheless, this trace has provided some very helpful debugging information.

If we prefer function graph output instead of function, we can set display-graph option:

```
with echo 1 > options/display-graph

 # tracer: irqsoff
 #
 # irqsoff latency trace v1.1.5 on 4.20.0-rc6+
 # --------------------------------------------------------------------
 # latency: 3751 us, #274/274, CPU#0 | (M:desktop VP:0, KP:0, SP:0 HP:0 #P:4)
 #    -----------------
 #    | task: bash-1507 (uid:0 nice:0 policy:0 rt_prio:0)
 #    -----------------
 #  => started at: free_debug_processing
 #  => ended at:   return_to_handler
 #
 #
 #                                       _-----=> irqs-off
 #                                      / _----=> need-resched
 #                                     | / _---=> hardirq/softirq
 #                                     || / _--=> preempt-depth
 #                                     ||| /
 #   REL TIME      CPU  TASK/PID       ||||     DURATION                  FUNCTION CALLS
 #      |          |     |    |        ||||      |   |                     |   |   |   |
         0 us |   0)   bash-1507    |  d... |   0.000 us    |  _raw_spin_lock_irqsave();
         0 us |   0)   bash-1507    |  d..1 |   0.378 us    |    do_raw_spin_trylock();
         1 us |   0)   bash-1507    |  d..2 |               |    set_track() {
         2 us |   0)   bash-1507    |  d..2 |               |      save_stack_trace() {
         2 us |   0)   bash-1507    |  d..2 |               |        __save_stack_trace() {
         3 us |   0)   bash-1507    |  d..2 |               |          __unwind_start() {
         3 us |   0)   bash-1507    |  d..2 |               |            get_stack_info() {
         3 us |   0)   bash-1507    |  d..2 |   0.351 us    |              in_task_stack();
         4 us |   0)   bash-1507    |  d..2 |   1.107 us    |            }
 [...]
      3750 us |   0)   bash-1507    |  d..1 |   0.516 us    |      do_raw_spin_unlock();
      3750 us |   0)   bash-1507    |  d..1 |   0.000 us    |  _raw_spin_unlock_irqrestore();
      3764 us |   0)   bash-1507    |  d..1 |   0.000 us    |  tracer_hardirqs_on();
     bash-1507    0d..1 3792us : <stack trace>
  => free_debug_processing
  => __slab_free
  => kmem_cache_free
  => vm_area_free
  => remove_vma
  => exit_mmap
  => mmput
  => begin_new_exec
  => load_elf_binary
  => search_binary_handler
  => __do_execve_file.isra.32
  => __x64_sys_execve
  => do_syscall_64
  => entry_SYSCALL_64_after_hwframe
```

preemptoff

When preemption is disabled, we may be able to receive interrupts but the task cannot be preempted and a higher priority task must wait for preemption to be enabled again before it can preempt a lower priority task.

The preemptoff tracer traces the places that disable preemption. Like the irqsoff tracer, it records the maximum latency for which preemption was disabled. The control of preemptoff tracer is much like the irqsoff tracer.

```
# echo 0 > options/function-trace
# echo preemptoff > current_tracer
# echo 1 > tracing_on
# echo 0 > tracing_max_latency
# ls -ltr
[...]
# echo 0 > tracing_on
# cat trace
# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.8.0-test+
# --------------------------------------------------------------------
# latency: 46 us, #4/4, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: sshd-1991 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: do_IRQ
#  => ended at:   do_IRQ
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
    sshd-1991    1d.h.    0us+: irq_enter <-do_IRQ
    sshd-1991    1d..1   46us : irq_exit <-do_IRQ
    sshd-1991    1d..1   47us+: trace_preempt_on <-do_IRQ
    sshd-1991    1d..1   52us : <stack trace>
 => sub_preempt_count
 => irq_exit
 => do_IRQ
 => ret_from_intr
```

This has some more changes. Preemption was disabled when an interrupt came in (notice the 'h'), and was enabled on exit. But we also see that interrupts have been disabled when entering the preempt off section and leaving it (the 'd'). We do not know if interrupts were enabled in the mean time or shortly after this was over.

```
# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.8.0-test+
# --------------------------------------------------------------------
# latency: 83 us, #241/241, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: bash-1994 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: wake_up_new_task
#  => ended at:   task_rq_unlock
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
    bash-1994    1d..1    0us : _raw_spin_lock_irqsave <-wake_up_new_task
    bash-1994    1d..1    0us : select_task_rq_fair <-select_task_rq
    bash-1994    1d..1    1us : __rcu_read_lock <-select_task_rq_fair
    bash-1994    1d..1    1us : source_load <-select_task_rq_fair
```

```
    bash-1994    1d..1    1us : source_load <-select_task_rq_fair
[...]
    bash-1994    1d..1   12us : irq_enter <-smp_apic_timer_interrupt
    bash-1994    1d..1   12us : rcu_irq_enter <-irq_enter
    bash-1994    1d..1   13us : add_preempt_count <-irq_enter
    bash-1994    1d.h1   13us : exit_idle <-smp_apic_timer_interrupt
    bash-1994    1d.h1   13us : hrtimer_interrupt <-smp_apic_timer_interrupt
    bash-1994    1d.h1   13us : _raw_spin_lock <-hrtimer_interrupt
    bash-1994    1d.h1   14us : add_preempt_count <-_raw_spin_lock
    bash-1994    1d.h2   14us : ktime_get_update_offsets <-hrtimer_interrupt
[...]
    bash-1994    1d.h1   35us : lapic_next_event <-clockevents_program_event
    bash-1994    1d.h1   35us : irq_exit <-smp_apic_timer_interrupt
    bash-1994    1d.h1   36us : sub_preempt_count <-irq_exit
    bash-1994    1d..2   36us : do_softirq <-irq_exit
    bash-1994    1d..2   36us : __do_softirq <-call_softirq
    bash-1994    1d..2   36us : __local_bh_disable <-__do_softirq
    bash-1994    1d.s2   37us : add_preempt_count <-_raw_spin_lock_irq
    bash-1994    1d.s3   38us : _raw_spin_unlock <-run_timer_softirq
    bash-1994    1d.s3   39us : sub_preempt_count <-_raw_spin_unlock
    bash-1994    1d.s2   39us : call_timer_fn <-run_timer_softirq
[...]
    bash-1994    1dNs2   81us : cpu_needs_another_gp <-rcu_process_callbacks
    bash-1994    1dNs2   82us : __local_bh_enable <-__do_softirq
    bash-1994    1dNs2   82us : sub_preempt_count <-__local_bh_enable
    bash-1994    1dN.2   82us : idle_cpu <-irq_exit
    bash-1994    1dN.2   83us : rcu_irq_exit <-irq_exit
    bash-1994    1dN.2   83us : sub_preempt_count <-irq_exit
    bash-1994    1.N.1   84us : _raw_spin_unlock_irqrestore <-task_rq_unlock
    bash-1994    1.N.1   84us+: trace_preempt_on <-task_rq_unlock
    bash-1994    1.N.1  104us : <stack trace>
 => sub_preempt_count
 => _raw_spin_unlock_irqrestore
 => task_rq_unlock
 => wake_up_new_task
 => do_fork
 => sys_clone
 => stub_clone
```

The above is an example of the preemptoff trace with function-trace set. Here we see that interrupts were not disabled the entire time. The irq_enter code lets us know that we entered an interrupt 'h'. Before that, the functions being traced still show that it is not in an interrupt, but we can see from the functions themselves that this is not the case.

preemptirqsoff

Knowing the locations that have interrupts disabled or preemption disabled for the longest times is helpful. But sometimes we would like to know when either preemption and/or interrupts are disabled.

Consider the following code:

```
local_irq_disable();
call_function_with_irqs_off();
preempt_disable();
call_function_with_irqs_and_preemption_off();
local_irq_enable();
call_function_with_preemption_off();
preempt_enable();
```

The irqsoff tracer will record the total length of call_function_with_irqs_off() and call_function_with_irqs_and_preemption_off().

The preemptoff tracer will record the total length of call_function_with_irqs_and_preemption_off() and call_function_with_preemption_off().

But neither will trace the time that interrupts and/or preemption is disabled. This total time is the time that we can not schedule. To record this time, use the preemptirqsoff tracer.

Again, using this trace is much like the irqsoff and preemptoff tracers.

```
# echo 0 > options/function-trace
# echo preemptirqsoff > current_tracer
# echo 1 > tracing_on
# echo 0 > tracing_max_latency
# ls -ltr
[...]
# echo 0 > tracing_on
# cat trace
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.8.0-test+
# --------------------------------------------------------------------
# latency: 100 us, #4/4, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: ls-2230 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: ata_scsi_queuecmd
#  => ended at:   ata_scsi_queuecmd
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
      ls-2230    3d...    0us+: _raw_spin_lock_irqsave <-ata_scsi_queuecmd
      ls-2230    3...1  100us : _raw_spin_unlock_irqrestore <-ata_scsi_queuecmd
      ls-2230    3...1  101us+: trace_preempt_on <-ata_scsi_queuecmd
      ls-2230    3...1  111us : <stack trace>
 => sub_preempt_count
 => _raw_spin_unlock_irqrestore
 => ata_scsi_queuecmd
 => scsi_dispatch_cmd
 => scsi_request_fn
 => __blk_run_queue_uncond
 => __blk_run_queue
 => blk_queue_bio
 => submit_bio_noacct
 => submit_bio
 => submit_bh
 => ext3_bread
 => ext3_dir_bread
 => htree_dirblock_to_tree
 => ext3_htree_fill_tree
 => ext3_readdir
 => vfs_readdir
 => sys_getdents
 => system_call_fastpath
```

The trace_hardirqs_off_thunk is called from assembly on x86 when interrupts are disabled in the assembly code. Without the function tracing, we do not know if interrupts were enabled within the preemption points. We do see that it started with preemption enabled.

Here is a trace with function-trace set:

```
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.8.0-test+
# --------------------------------------------------------------------
```

```
# latency: 161 us, #339/339, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: ls-2269 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: schedule
#  => ended at:   mutex_unlock
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
kworker/-59      3...1    0us : __schedule <-schedule
kworker/-59      3d..1    0us : rcu_preempt_qs <-rcu_note_context_switch
kworker/-59      3d..1    1us : add_preempt_count <-_raw_spin_lock_irq
kworker/-59      3d..2    1us : deactivate_task <-__schedule
kworker/-59      3d..2    1us : dequeue_task <-deactivate_task
kworker/-59      3d..2    2us : update_rq_clock <-dequeue_task
kworker/-59      3d..2    2us : dequeue_task_fair <-dequeue_task
kworker/-59      3d..2    2us : update_curr <-dequeue_task_fair
kworker/-59      3d..2    2us : update_min_vruntime <-update_curr
kworker/-59      3d..2    3us : cpuacct_charge <-update_curr
kworker/-59      3d..2    3us : __rcu_read_lock <-cpuacct_charge
kworker/-59      3d..2    3us : __rcu_read_unlock <-cpuacct_charge
kworker/-59      3d..2    3us : update_cfs_rq_blocked_load <-dequeue_task_fair
kworker/-59      3d..2    4us : clear_buddies <-dequeue_task_fair
kworker/-59      3d..2    4us : account_entity_dequeue <-dequeue_task_fair
kworker/-59      3d..2    4us : update_min_vruntime <-dequeue_task_fair
kworker/-59      3d..2    4us : update_cfs_shares <-dequeue_task_fair
kworker/-59      3d..2    5us : hrtick_update <-dequeue_task_fair
kworker/-59      3d..2    5us : wq_worker_sleeping <-__schedule
kworker/-59      3d..2    5us : kthread_data <-wq_worker_sleeping
kworker/-59      3d..2    5us : put_prev_task_fair <-__schedule
kworker/-59      3d..2    6us : pick_next_task_fair <-pick_next_task
kworker/-59      3d..2    6us : clear_buddies <-pick_next_task_fair
kworker/-59      3d..2    6us : set_next_entity <-pick_next_task_fair
kworker/-59      3d..2    6us : update_stats_wait_end <-set_next_entity
      ls-2269    3d..2    7us : finish_task_switch <-__schedule
      ls-2269    3d..2    7us : _raw_spin_unlock_irq <-finish_task_switch
      ls-2269    3d..2    8us : do_IRQ <-ret_from_intr
      ls-2269    3d..2    8us : irq_enter <-do_IRQ
      ls-2269    3d..2    8us : rcu_irq_enter <-irq_enter
      ls-2269    3d..2    9us : add_preempt_count <-irq_enter
      ls-2269    3d.h2    9us : exit_idle <-do_IRQ
[...]
      ls-2269    3d.h3   20us : sub_preempt_count <-_raw_spin_unlock
      ls-2269    3d.h2   20us : irq_exit <-do_IRQ
      ls-2269    3d.h2   21us : sub_preempt_count <-irq_exit
      ls-2269    3d..3   21us : do_softirq <-irq_exit
      ls-2269    3d..3   21us : __do_softirq <-call_softirq
      ls-2269    3d..3   21us+: __local_bh_disable <-__do_softirq
      ls-2269    3d.s4   29us : sub_preempt_count <-_local_bh_enable_ip
      ls-2269    3d.s5   29us : sub_preempt_count <-_local_bh_enable_ip
      ls-2269    3d.s5   31us : do_IRQ <-ret_from_intr
      ls-2269    3d.s5   31us : irq_enter <-do_IRQ
      ls-2269    3d.s5   31us : rcu_irq_enter <-irq_enter
[...]
      ls-2269    3d.s5   31us : rcu_irq_enter <-irq_enter
      ls-2269    3d.s5   32us : add_preempt_count <-irq_enter
      ls-2269    3d.H5   32us : exit_idle <-do_IRQ
      ls-2269    3d.H5   32us : handle_irq <-do_IRQ
      ls-2269    3d.H5   32us : irq_to_desc <-handle_irq
      ls-2269    3d.H5   33us : handle_fasteoi_irq <-handle_irq
[...]
      ls-2269    3d.s5  158us : _raw_spin_unlock_irqrestore <-rtl8139_poll
      ls-2269    3d.s3  158us : net_rps_action_and_irq_enable.isra.65 <-net_rx_action
      ls-2269    3d.s3  159us : __local_bh_enable <-__do_softirq
      ls-2269    3d.s3  159us : sub_preempt_count <-__local_bh_enable
      ls-2269    3d..3  159us : idle_cpu <-irq_exit
      ls-2269    3d..3  159us : rcu_irq_exit <-irq_exit
      ls-2269    3d..3  160us : sub_preempt_count <-irq_exit
      ls-2269    3d...  161us : __mutex_unlock_slowpath <-mutex_unlock
      ls-2269    3d...  162us+: trace_hardirqs_on <-mutex_unlock
      ls-2269    3d...  186us : <stack trace>
 => __mutex_unlock_slowpath
 => mutex_unlock
 => process_output
 => n_tty_write
 => tty_write
 => vfs_write
 => sys_write
 => system_call_fastpath
```

This is an interesting trace. It started with kworker running and scheduling out and ls taking over. But as soon as ls released the rq lock and enabled interrupts (but not preemption) an interrupt triggered. When the interrupt finished, it started running softirqs. But while the softirq was running, another interrupt triggered. When an interrupt is running inside a softirq, the annotation is 'H'.

wakeup

One common case that people are interested in tracing is the time it takes for a task that is woken to actually wake up. Now for non Real-Time tasks, this can be arbitrary. But tracing it none the less can be interesting.

Without function tracing:

```
# echo 0 > options/function-trace
# echo wakeup > current_tracer
# echo 1 > tracing_on
# echo 0 > tracing_max_latency
# chrt -f 5 sleep 1
# echo 0 > tracing_on
# cat trace
# tracer: wakeup
#
# wakeup latency trace v1.1.5 on 3.8.0-test+
# --------------------------------------------------------------------
# latency: 15 us, #4/4, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: kworker/3:1H-312 (uid:0 nice:-20 policy:0 rt_prio:0)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>-0       3dNs7    0us :      0:120:R   + [003]   312:100:R kworker/3:1H
  <idle>-0       3dNs7    1us+: ttwu_do_activate.constprop.87 <-try_to_wake_up
  <idle>-0       3d..3   15us : __schedule <-schedule
  <idle>-0       3d..3   15us :      0:120:R ==> [003]   312:100:R kworker/3:1H
```

The tracer only traces the highest priority task in the system to avoid tracing the normal circumstances. Here we see that the kworker with a nice priority of -20 (not very nice), took just 15 microseconds from the time it woke up, to the time it ran.

Non Real-Time tasks are not that interesting. A more interesting trace is to concentrate only on Real-Time tasks.

wakeup_rt

In a Real-Time environment it is very important to know the wakeup time it takes for the highest priority task that is woken up to the time that it executes. This is also known as "schedule latency". I stress the point that this is about RT tasks. It is also important to know the scheduling latency of non-RT tasks, but the average schedule latency is better for non-RT tasks. Tools like LatencyTop are more appropriate for such measurements.

Real-Time environments are interested in the worst case latency. That is the longest latency it takes for something to happen, and not the average. We can have a very fast scheduler that may only have a large latency once in a while, but that would not work well with Real-Time tasks. The wakeup_rt tracer was designed to record the worst case wakeups of RT tasks. Non-RT tasks are not recorded because the tracer only records one worst case and tracing non-RT tasks that are unpredictable will overwrite the worst case latency of RT tasks (just run the normal wakeup tracer for a while to see that effect).

Since this tracer only deals with RT tasks, we will run this slightly differently than we did with the previous tracers. Instead of performing an 'ls', we will run 'sleep 1' under 'chrt' which changes the priority of the task.

```
  # echo 0 > options/function-trace
  # echo wakeup_rt > current_tracer
  # echo 1 > tracing_on
  # echo 0 > tracing_max_latency
  # chrt -f 5 sleep 1
  # echo 0 > tracing_on
  # cat trace
  # tracer: wakeup
  #
  # tracer: wakeup_rt
  #
  # wakeup_rt latency trace v1.1.5 on 3.8.0-test+
  # --------------------------------------------------------------------
  # latency: 5 us, #4/4, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
  #    -----------------
  #    | task: sleep-2389 (uid:0 nice:0 policy:1 rt_prio:5)
  #    -----------------
  #
  #                  _------=> CPU#            
  #                 / _-----=> irqs-off        
  #                | / _----=> need-resched    
  #                || / _---=> hardirq/softirq 
  #                ||| / _--=> preempt-depth   
  #                |||| /     delay             
  #  cmd     pid   ||||| time  |   caller      
  #     \   /      |||||  \    |   /           
    <idle>-0       3d.h4    0us :      0:120:R   + [003]  2389: 94:R sleep
    <idle>-0       3d.h4    1us+: ttwu_do_activate.constprop.87 <-try_to_wake_up
    <idle>-0       3d..3    5us : __schedule <-schedule
    <idle>-0       3d..3    5us :      0:120:R ==> [003]  2389: 94:R sleep
```

Running this on an idle system, we see that it only took 5 microseconds to perform the task switch. Note, since the trace point in the schedule is before the actual "switch", we stop the tracing when the recorded task is about to schedule in. This may change if we add a new marker at the end of the scheduler.

Notice that the recorded task is 'sleep' with the PID of 2389 and it has an rt_prio of 5. This priority is user-space priority and not the internal kernel priority. The policy is 1 for SCHED_FIFO and 2 for SCHED_RR.

Note, that the trace data shows the internal priority (99 - rtprio).

```
  <idle>-0       3d..3    5us :      0:120:R ==> [003]  2389: 94:R sleep
```

The 0:120:R means idle was running with a nice priority of 0 (120 - 120) and in the running state 'R'. The sleep task was scheduled in with 2389: 94:R. That is the priority is the kernel rtprio (99 - 5 = 94) and it too is in the running state.

Doing the same with chrt -r 5 and function-trace set.

```
  echo 1 > options/function-trace

  # tracer: wakeup_rt
  #
  # wakeup_rt latency trace v1.1.5 on 3.8.0-test+
  # --------------------------------------------------------------------
  # latency: 29 us, #85/85, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
  #    -----------------
  #    | task: sleep-2448 (uid:0 nice:0 policy:1 rt_prio:5)
  #    -----------------
  #
  #                  _------=> CPU#            
  #                 / _-----=> irqs-off        
  #                | / _----=> need-resched    
  #                || / _---=> hardirq/softirq 
  #                ||| / _--=> preempt-depth   
  #                |||| /     delay             
  #  cmd     pid   ||||| time  |   caller      
  #     \   /      |||||  \    |   /           
    <idle>-0       3d.h4    1us+:      0:120:R   + [003]  2448: 94:R sleep
    <idle>-0       3d.h4    2us : ttwu_do_activate.constprop.87 <-try_to_wake_up
    <idle>-0       3d.h3    3us : check_preempt_curr <-ttwu_do_wakeup
    <idle>-0       3d.h3    3us : resched_curr <-check_preempt_curr
    <idle>-0       3dNh3    4us : task_woken_rt <-ttwu_do_wakeup
    <idle>-0       3dNh3    4us : _raw_spin_unlock <-try_to_wake_up
    <idle>-0       3dNh3    4us : sub_preempt_count <-_raw_spin_unlock
    <idle>-0       3dNh2    5us : ttwu_stat <-try_to_wake_up
    <idle>-0       3dNh2    5us : _raw_spin_unlock_irqrestore <-try_to_wake_up
    <idle>-0       3dNh2    6us : sub_preempt_count <-_raw_spin_unlock_irqrestore
    <idle>-0       3dNh1    6us : _raw_spin_lock <-__run_hrtimer
    <idle>-0       3dNh1    6us : add_preempt_count <-_raw_spin_lock
    <idle>-0       3dNh2    7us : _raw_spin_unlock <-hrtimer_interrupt
    <idle>-0       3dNh2    7us : sub_preempt_count <-_raw_spin_unlock
    <idle>-0       3dNh1    7us : tick_program_event <-hrtimer_interrupt
    <idle>-0       3dNh1    7us : clockevents_program_event <-tick_program_event
    <idle>-0       3dNh1    8us : ktime_get <-clockevents_program_event
    <idle>-0       3dNh1    8us : lapic_next_event <-clockevents_program_event
    <idle>-0       3dNh1    8us : irq_exit <-smp_apic_timer_interrupt
    <idle>-0       3dNh1    9us : sub_preempt_count <-irq_exit
    <idle>-0       3dN.2    9us : idle_cpu <-irq_exit
    <idle>-0       3dN.2    9us : rcu_irq_exit <-irq_exit
    <idle>-0       3dN.2   10us : rcu_eqs_enter_common.isra.45 <-rcu_irq_exit
    <idle>-0       3dN.2   10us : sub_preempt_count <-irq_exit
    <idle>-0       3.N.1   11us : rcu_idle_exit <-cpu_idle
    <idle>-0       3dN.1   11us : rcu_eqs_exit_common.isra.43 <-rcu_idle_exit
    <idle>-0       3.N.1   11us : tick_nohz_idle_exit <-cpu_idle
    <idle>-0       3dN.1   12us : menu_hrtimer_cancel <-tick_nohz_idle_exit
    <idle>-0       3dN.1   12us : ktime_get <-tick_nohz_idle_exit
    <idle>-0       3dN.1   12us : tick_do_update_jiffies64 <-tick_nohz_idle_exit
    <idle>-0       3dN.1   13us : cpu_load_update_nohz <-tick_nohz_idle_exit
    <idle>-0       3dN.1   13us : _raw_spin_lock <-cpu_load_update_nohz
    <idle>-0       3dN.1   13us : add_preempt_count <-_raw_spin_lock
    <idle>-0       3dN.2   13us : __cpu_load_update <-cpu_load_update_nohz
    <idle>-0       3dN.2   14us : sched_avg_update <-__cpu_load_update
    <idle>-0       3dN.2   14us : _raw_spin_unlock <-cpu_load_update_nohz
    <idle>-0       3dN.2   14us : sub_preempt_count <-_raw_spin_unlock
    <idle>-0       3dN.1   15us : calc_load_nohz_stop <-tick_nohz_idle_exit
    <idle>-0       3dN.1   15us : touch_softlockup_watchdog <-tick_nohz_idle_exit
    <idle>-0       3dN.1   15us : hrtimer_cancel <-tick_nohz_idle_exit
    <idle>-0       3dN.1   15us : hrtimer_try_to_cancel <-hrtimer_cancel
    <idle>-0       3dN.1   16us : lock_hrtimer_base.isra.18 <-hrtimer_try_to_cancel
    <idle>-0       3dN.1   16us : _raw_spin_lock_irqsave <-lock_hrtimer_base.isra.18
    <idle>-0       3dN.1   16us : add_preempt_count <-_raw_spin_lock_irqsave
```

```
    <idle>-0       3dN.2   17us : __remove_hrtimer <-remove_hrtimer.part.16
    <idle>-0       3dN.2   17us : hrtimer_force_reprogram <-__remove_hrtimer
    <idle>-0       3dN.2   17us : tick_program_event <-hrtimer_force_reprogram
    <idle>-0       3dN.2   18us : clockevents_program_event <-tick_program_event
    <idle>-0       3dN.2   18us : ktime_get <-clockevents_program_event
    <idle>-0       3dN.2   18us : lapic_next_event <-clockevents_program_event
    <idle>-0       3dN.2   19us : _raw_spin_unlock_irqrestore <-hrtimer_try_to_cancel
    <idle>-0       3dN.2   19us : sub_preempt_count <-_raw_spin_unlock_irqrestore
    <idle>-0       3dN.1   19us : hrtimer_forward <-tick_nohz_idle_exit
    <idle>-0       3dN.1   20us : ktime_add_safe <-hrtimer_forward
    <idle>-0       3dN.1   20us : ktime_add_safe <-hrtimer_forward
    <idle>-0       3dN.1   20us : hrtimer_start_range_ns <-hrtimer_start_expires.constprop.11
    <idle>-0       3dN.1   20us : __hrtimer_start_range_ns <-hrtimer_start_range_ns
    <idle>-0       3dN.1   21us : lock_hrtimer_base.isra.18 <-__hrtimer_start_range_ns
    <idle>-0       3dN.1   21us : _raw_spin_lock_irqsave <-lock_hrtimer_base.isra.18
    <idle>-0       3dN.1   21us : add_preempt_count <-_raw_spin_lock_irqsave
    <idle>-0       3dN.2   22us : ktime_add_safe <-__hrtimer_start_range_ns
    <idle>-0       3dN.2   22us : enqueue_hrtimer <-__hrtimer_start_range_ns
    <idle>-0       3dN.2   22us : tick_program_event <-__hrtimer_start_range_ns
    <idle>-0       3dN.2   23us : clockevents_program_event <-tick_program_event
    <idle>-0       3dN.2   23us : ktime_get <-clockevents_program_event
    <idle>-0       3dN.2   23us : lapic_next_event <-clockevents_program_event
    <idle>-0       3dN.2   24us : _raw_spin_unlock_irqrestore <-__hrtimer_start_range_ns
    <idle>-0       3dN.2   24us : sub_preempt_count <-_raw_spin_unlock_irqrestore
    <idle>-0       3dN.1   24us : account_idle_ticks <-tick_nohz_idle_exit
    <idle>-0       3dN.1   24us : account_idle_time <-account_idle_ticks
    <idle>-0       3.N.1   25us : sub_preempt_count <-cpu_idle
    <idle>-0       3.N..   25us : schedule <-cpu_idle
    <idle>-0       3.N..   25us : __schedule <-preempt_schedule
    <idle>-0       3.N..   26us : add_preempt_count <-__schedule
    <idle>-0       3.N.1   26us : rcu_note_context_switch <-__schedule
    <idle>-0       3.N.1   26us : rcu_sched_qs <-rcu_note_context_switch
    <idle>-0       3dN.1   27us : rcu_preempt_qs <-rcu_note_context_switch
    <idle>-0       3.N.1   27us : _raw_spin_lock_irq <-__schedule
    <idle>-0       3dN.1   27us : add_preempt_count <-_raw_spin_lock_irq
    <idle>-0       3dN.2   28us : put_prev_task_idle <-__schedule
    <idle>-0       3dN.2   28us : pick_next_task_stop <-pick_next_task
    <idle>-0       3dN.2   28us : pick_next_task_rt <-pick_next_task
    <idle>-0       3dN.2   29us : dequeue_pushable_task <-pick_next_task_rt
    <idle>-0       3d..3   29us : __schedule <-preempt_schedule
    <idle>-0       3d..3   30us :      0:120:R ==> [003]  2448: 94:R sleep
```

This isn't that big of a trace, even with function tracing enabled, so I included the entire trace.

The interrupt went off while when the system was idle. Somewhere before task_woken_rt() was called, the NEED_RESCHED flag was set, this is indicated by the first occurrence of the 'N' flag.

Latency tracing and events

As function tracing can induce a much larger latency, but without seeing what happens within the latency it is hard to know what caused it. There is a middle ground, and that is with enabling events.

```
  # echo 0 > options/function-trace
  # echo wakeup_rt > current_tracer
  # echo 1 > events/enable
  # echo 1 > tracing_on
  # echo 0 > tracing_max_latency
  # chrt -f 5 sleep 1
  # echo 0 > tracing_on
  # cat trace
  # tracer: wakeup_rt
  #
  # wakeup_rt latency trace v1.1.5 on 3.8.0-test+
  # --------------------------------------------------------------------
  # latency: 6 us, #12/12, CPU#2 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
  #    -----------------
  #    | task: sleep-5882 (uid:0 nice:0 policy:1 rt_prio:5)
  #    -----------------
  #
  #                  _------=> CPU#
  #                 / _-----=> irqs-off
  #                | / _----=> need-resched
  #                || / _---=> hardirq/softirq
  #                ||| / _--=> preempt-depth
  #                |||| /     delay
  #  cmd     pid   ||||| time  |   caller
  #     \   /      |||||  \    |   /
    <idle>-0       2d.h4    0us :      0:120:R   + [002]  5882: 94:R sleep
    <idle>-0       2d.h4    0us : ttwu_do_activate.constprop.87 <-try_to_wake_up
    <idle>-0       2d.h4    1us : sched_wakeup: comm=sleep pid=5882 prio=94 success=1 target_cpu=002
    <idle>-0       2dNh2    1us : hrtimer_expire_exit: hrtimer=ffff88007796feb8
    <idle>-0       2.N.2    2us : power_end: cpu_id=2
    <idle>-0       2.N.2    3us : cpu_idle: state=4294967295 cpu_id=2
    <idle>-0       2dN.3    4us : hrtimer_cancel: hrtimer=ffff88007d50d5e0
    <idle>-0       2dN.3    4us : hrtimer_start: hrtimer=ffff88007d50d5e0 function=tick_sched_timer expires=34311211000000 softexpires=3431
    <idle>-0       2.N.2    5us : rcu_utilization: Start context switch
    <idle>-0       2.N.2    5us : rcu_utilization: End context switch
    <idle>-0       2d..3    6us : __schedule <-schedule
    <idle>-0       2d..3    6us :      0:120:R ==> [002]  5882: 94:R sleep
```

Hardware Latency Detector

The hardware latency detector is executed by enabling the "hwlat" tracer.

NOTE, this tracer will affect the performance of the system as it will periodically make a CPU constantly busy with interrupts disabled.

```
  # echo hwlat > current_tracer
  # sleep 100
  # cat trace
  # tracer: hwlat
  #
  # entries-in-buffer/entries-written: 13/13   #P:8
  #
  #                              _-----=> irqs-off
  #                             / _----=> need-resched
  #                            | / _---=> hardirq/softirq
  #                            || / _--=> preempt-depth
  #                            ||| /     delay
  #           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
  #              | |       |   ||||       |         |
           <...>-1729  [001] d...   678.473449: #1     inner/outer(us):   11/12    ts:1581527483.343962693 count:6
           <...>-1729  [004] d...   689.556542: #2     inner/outer(us):   16/9     ts:1581527494.889008092 count:1
           <...>-1729  [005] d...   714.756290: #3     inner/outer(us):   16/16    ts:1581527519.678961629 count:5
           <...>-1729  [001] d...   718.788247: #4     inner/outer(us):    9/17    ts:1581527523.889012713 count:1
           <...>-1729  [002] d...   719.796341: #5     inner/outer(us):   13/9     ts:1581527524.912872606 count:1
           <...>-1729  [006] d...   844.787091: #6     inner/outer(us):    9/12    ts:1581527649.889048502 count:2
           <...>-1729  [003] d...   849.827033: #7     inner/outer(us):   18/9     ts:1581527654.889013793 count:1
           <...>-1729  [007] d...   853.859002: #8     inner/outer(us):    9/12    ts:1581527658.889065736 count:1
           <...>-1729  [001] d...   855.874978: #9     inner/outer(us):    9/11    ts:1581527660.861991877 count:1
           <...>-1729  [001] d...   863.938932: #10    inner/outer(us):    9/11    ts:1581527668.970010500 count:1 nmi-total:7 nmi-count
           <...>-1729  [007] d...   878.050780: #11    inner/outer(us):    9/12    ts:1581527683.385002600 count:1 nmi-total:5 nmi-count
           <...>-1729  [007] d...   886.114702: #12    inner/outer(us):    9/12    ts:1581527691.385001600 count:1
```

The above output is somewhat the same in the header. All events will have interrupts disabled 'd'. Under the FUNCTION title there is:

#1

This is the count of events recorded that were greater than the tracing_threshold (See below).

inner/outer(us): 11/11

This shows two numbers as "inner latency" and "outer latency". The test runs in a loop checking a timestamp twice. The latency detected within the two timestamps is the "inner latency" and the latency detected after the previous timestamp and the next timestamp in the loop is the "outer latency".

ts:1581527483.343962693

The absolute timestamp that the first latency was recorded in the window.

count:6

The number of times a latency was detected during the window.

nmi-total:7 nmi-count:1

On architectures that support it, if an NMI comes in during the test, the time spent in NMI is reported in "nmi-total" (in microseconds).

All architectures that have NMIs will show the "nmi-count" if an NMI comes in during the test.

hwlat files:

tracing_threshold

This gets automatically set to "10" to represent 10 microseconds. This is the threshold of latency that needs to be detected before the trace will be recorded.

Note, when hwlat tracer is finished (another tracer is written into "current_tracer"), the original value for tracing_threshold is placed back into this file.

hwlat_detector/width

The length of time the test runs with interrupts disabled.

hwlat_detector/window

The length of time of the window which the test runs. That is, the test will run for "width" microseconds per "window" microseconds

tracing_cpumask

When the test is started. A kernel thread is created that runs the test. This thread will alternate between CPUs listed in the tracing_cpumask between each period (one "window"). To limit the test to specific CPUs set the mask in this file to only the CPUs that the test should run on.

function

This tracer is the function tracer. Enabling the function tracer can be done from the debug file system. Make sure the ftrace_enabled is set; otherwise this tracer is a nop. See the "ftrace_enabled" section below.

```
# sysctl kernel.ftrace_enabled=1
# echo function > current_tracer
# echo 1 > tracing_on
# usleep 1
# echo 0 > tracing_on
# cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 24799/24799   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
            bash-1994  [002] ....  3082.063030: mutex_unlock <-rb_simple_write
            bash-1994  [002] ....  3082.063031: __mutex_unlock_slowpath <-mutex_unlock
            bash-1994  [002] ....  3082.063031: __fsnotify_parent <-fsnotify_modify
            bash-1994  [002] ....  3082.063032: fsnotify <-fsnotify_modify
            bash-1994  [002] ....  3082.063032: __srcu_read_lock <-fsnotify
            bash-1994  [002] ....  3082.063032: add_preempt_count <-__srcu_read_lock
            bash-1994  [002] ...1  3082.063032: sub_preempt_count <-__srcu_read_lock
            bash-1994  [002] ....  3082.063033: __srcu_read_unlock <-fsnotify
[...]
```

Note: function tracer uses ring buffers to store the above entries. The newest data may overwrite the oldest data. Sometimes using echo to stop the trace is not sufficient because the tracing could have overwritten the data that you wanted to record. For this reason, it is sometimes better to disable tracing directly from a program. This allows you to stop the tracing at the point that you hit the part that you are interested in. To disable the tracing directly from a C program, something like following code snippet can be used:

```
int trace_fd;
[...]
int main(int argc, char *argv[]) {
        [...]
        trace_fd = open(tracing_file("tracing_on"), O_WRONLY);
        [...]
        if (condition_hit()) {
                write(trace_fd, "0", 1);
        }
        [...]
}
```

Single thread tracing

By writing into set_ftrace_pid you can trace a single thread. For example:

```
# cat set_ftrace_pid
no pid
# echo 3111 > set_ftrace_pid
# cat set_ftrace_pid
3111
# echo function > current_tracer
# cat trace | head
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
    yum-updatesd-3111  [003]  1637.254676: finish_task_switch <-thread_return
    yum-updatesd-3111  [003]  1637.254681: hrtimer_cancel <-schedule_hrtimeout_range
    yum-updatesd-3111  [003]  1637.254682: hrtimer_try_to_cancel <-hrtimer_cancel
    yum-updatesd-3111  [003]  1637.254683: lock_hrtimer_base <-hrtimer_try_to_cancel
    yum-updatesd-3111  [003]  1637.254685: fget_light <-do_sys_poll
    yum-updatesd-3111  [003]  1637.254686: pipe_poll <-do_sys_poll
# echo > set_ftrace_pid
# cat trace |head
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
##### CPU 3 buffer started ####
    yum-updatesd-3111  [003]  1701.957688: free_poll_entry <-poll_freewait
```

```
     yum-updatesd-3111  [003]  1701.957689: remove_wait_queue <-free_poll_entry
     yum-updatesd-3111  [003]  1701.957691: fput <-free_poll_entry
     yum-updatesd-3111  [003]  1701.957692: audit_syscall_exit <-sysret_audit
     yum-updatesd-3111  [003]  1701.957693: path_put <-audit_syscall_exit
```

If you want to trace a function when executing, you could use something like this simple program.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define _STR(x) #x
#define STR(x) _STR(x)
#define MAX_PATH 256

const char *find_tracefs(void)
{
       static char tracefs[MAX_PATH+1];
       static int tracefs_found;
       char type[100];
       FILE *fp;

       if (tracefs_found)
               return tracefs;

       if ((fp = fopen("/proc/mounts","r")) == NULL) {
               perror("/proc/mounts");
               return NULL;
       }

       while (fscanf(fp, "%*s %"
                     STR(MAX_PATH)
                     "s %99s %*s %*d %*d\n",
                     tracefs, type) == 2) {
               if (strcmp(type, "tracefs") == 0)
                       break;
       }
       fclose(fp);

       if (strcmp(type, "tracefs") != 0) {
               fprintf(stderr, "tracefs not mounted");
               return NULL;
       }

       strcat(tracefs, "/tracing/");
       tracefs_found = 1;

       return tracefs;
}

const char *tracing_file(const char *file_name)
{
       static char trace_file[MAX_PATH+1];
       snprintf(trace_file, MAX_PATH, "%s/%s", find_tracefs(), file_name);
       return trace_file;
}

int main (int argc, char **argv)
{
        if (argc < 1)
                exit(-1);

        if (fork() > 0) {
                int fd, ffd;
                char line[64];
                int s;

                ffd = open(tracing_file("current_tracer"), O_WRONLY);
                if (ffd < 0)
                        exit(-1);
                write(ffd, "nop", 3);

                fd = open(tracing_file("set_ftrace_pid"), O_WRONLY);
                s = sprintf(line, "%d\n", getpid());
                write(fd, line, s);

                write(ffd, "function", 8);

                close(fd);
                close(ffd);

                execvp(argv[1], argv+1);
        }

        return 0;
}
```

Or this simple script!

```
#!/bin/bash

tracefs=`sed -ne 's/^tracefs \(.*\) tracefs.*/\1/p' /proc/mounts`
echo 0 > $tracefs/tracing_on
echo $$ > $tracefs/set_ftrace_pid
echo function > $tracefs/current_tracer
echo 1 > $tracefs/tracing_on
exec "$@"
```

function graph tracer

This tracer is similar to the function tracer except that it probes a function on its entry and its exit. This is done by using a dynamically allocated stack of return addresses in each task_struct. On function entry the tracer overwrites the return address of each function traced to set a custom probe. Thus the original return address is stored on the stack of return address in the task_struct.

Probing on both ends of a function leads to special features such as:

- measure of a function's time execution
- having a reliable call stack to draw function calls graph

This tracer is useful in several situations:

- you want to find the reason of a strange kernel behavior and need to see what happens in detail on any areas (or specific ones).
- you are experiencing weird latencies but it's difficult to find its origin.
- you want to find quickly which path is taken by a specific function
- you just want to peek inside a working kernel and want to see what happens there.

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |

 0)               |  sys_open() {
 0)               |    do_sys_open() {
 0)               |      getname() {
```

```
0)               |          kmem_cache_alloc() {
0)   1.382 us    |            __might_sleep();
0)   2.478 us    |          }
0)               |          strncpy_from_user() {
0)               |            might_fault() {
0)   1.389 us    |              __might_sleep();
0)   2.553 us    |            }
0)   3.807 us    |          }
0)   7.876 us    |        }
0)               |        alloc_fd() {
0)   0.668 us    |          _spin_lock();
0)   0.570 us    |          expand_files();
0)   0.586 us    |          _spin_unlock();
```

There are several columns that can be dynamically enabled/disabled. You can use every combination of options you want, depending on your needs.

- The cpu number on which the function executed is default enabled. It is sometimes better to only trace one cpu (see tracing_cpu_mask file) or you might sometimes see unordered function calls while cpu tracing switch.
    - hide: echo nofuncgraph-cpu > trace_options
    - show: echo funcgraph-cpu > trace_options
- The duration (function's time of execution) is displayed on the closing bracket line of a function or on the same line than the current function in case of a leaf one. It is default enabled.
    - hide: echo nofuncgraph-duration > trace_options
    - show: echo funcgraph-duration > trace_options
- The overhead field precedes the duration field in case of reached duration thresholds.
    - hide: echo nofuncgraph-overhead > trace_options
    - show: echo funcgraph-overhead > trace_options
    - depends on: funcgraph-duration

  ie:

```
3) # 1837.709 us |          } /* __switch_to */
3)               |          finish_task_switch() {
3)   0.313 us    |            _raw_spin_unlock_irq();
3)   3.177 us    |          }
3) # 1889.063 us |        } /* __schedule */
3) ! 140.417 us  |      } /* __schedule */
3) # 2034.948 us |    } /* schedule */
3) * 33998.59 us |  } /* schedule_preempt_disabled */

[...]

1)   0.260 us    |              msecs_to_jiffies();
1)   0.313 us    |              __rcu_read_unlock();
1) + 61.770 us   |            }
1) + 64.479 us   |          }
1)   0.313 us    |          rcu_bh_qs();
1)   0.313 us    |          __local_bh_enable();
1) ! 217.240 us  |        }
1)   0.365 us    |        idle_cpu();
1)               |        rcu_irq_exit() {
1)   0.417 us    |          rcu_eqs_enter_common.isra.47();
1)   3.125 us    |        }
1) ! 227.812 us  |      }
1) ! 457.395 us  |    }
1) @ 119760.2 us |  }

[...]

2)               |    handle_IPI() {
1)   6.979 us    |                  }
2)   0.417 us    |      scheduler_ipi();
1)   9.791 us    |                }
1) + 12.917 us   |              }
2)   3.490 us    |    }
1) + 15.729 us   |            }
1) + 18.542 us   |          }
2) $ 3594274 us  |  }
```

Flags:

```
+ means that the function exceeded 10 usecs.
! means that the function exceeded 100 usecs.
# means that the function exceeded 1000 usecs.
* means that the function exceeded 10 msecs.
@ means that the function exceeded 100 msecs.
$ means that the function exceeded 1 sec.
```

- The task/pid field displays the thread cmdline and pid which executed the function. It is default disabled.
    - hide: echo nofuncgraph-proc > trace_options
    - show: echo funcgraph-proc > trace_options

  ie:

```
# tracer: function_graph
#
# CPU  TASK/PID        DURATION                  FUNCTION CALLS
# |    |    |           |   |                     |   |   |   |
0)    sh-4802     |               |                  d_free() {
0)    sh-4802     |               |                    call_rcu() {
0)    sh-4802     |               |                      __call_rcu() {
0)    sh-4802     |   0.616 us    |                        rcu_process_gp_end();
0)    sh-4802     |   0.586 us    |                        check_for_new_grace_period();
0)    sh-4802     |   2.899 us    |                      }
0)    sh-4802     |   4.040 us    |                    }
0)    sh-4802     |   5.151 us    |                  }
0)    sh-4802     | + 49.370 us   |                }
```

- The absolute time field is an absolute timestamp given by the system clock since it started. A snapshot of this time is given on each entry/exit of functions
    - hide: echo nofuncgraph-abstime > trace_options
    - show: echo funcgraph-abstime > trace_options

  ie:

```
#
#      TIME       CPU  DURATION                  FUNCTION CALLS
#       |         |     |   |                     |   |   |   |
360.774522 |   1)   0.541 us    |                                          }
360.774522 |   1)   4.663 us    |                                        }
360.774523 |   1)   0.541 us    |                                        __wake_up_bit();
360.774524 |   1)   6.796 us    |                                      }
360.774524 |   1)   7.952 us    |                                    }
360.774525 |   1)   9.063 us    |                                  }
360.774525 |   1)   0.615 us    |                                  journal_mark_dirty();
360.774527 |   1)   0.578 us    |                                  __brelse();
360.774528 |   1)               |                                  reiserfs_prepare_for_journal() {
360.774528 |   1)               |                                    unlock_buffer() {
```

```
     360.774529 |   1)               |                                                      wake_up_bit() {
     360.774529 |   1)               |                                                        bit_waitqueue() {
     360.774530 |   1)   0.594 us    |                                                          __phys_addr();
```

The function name is always displayed after the closing bracket for a function if the start of that function is not in the trace buffer.

Display of the function name after the closing bracket may be enabled for functions whose start is in the trace buffer, allowing easier searching with grep for function durations. It is default disabled.

- hide: echo nofuncgraph-tail > trace_options
- show: echo funcgraph-tail > trace_options

Example with nofuncgraph-tail (default):

```
0)               |      putname() {
0)               |        kmem_cache_free() {
0)   0.518 us    |          __phys_addr();
0)   1.757 us    |        }
0)   2.861 us    |      }
```

Example with funcgraph-tail:

```
0)               |      putname() {
0)               |        kmem_cache_free() {
0)   0.518 us    |          __phys_addr();
0)   1.757 us    |        } /* kmem_cache_free() */
0)   2.861 us    |      } /* putname() */
```

You can put some comments on specific functions by using trace_printk() For example, if you want to put a comment inside the __might_sleep() function, you just have to include <linux/ftrace.h> and call trace_printk() inside __might_sleep():

```
trace_printk("I'm a comment!\n")
```

will produce:

```
1)               |             __might_sleep() {
1)               |                /* I'm a comment! */
1)   1.449 us    |             }
```

You might find other useful features for this tracer in the following "dynamic ftrace" section such as tracing only specific functions or tasks.

dynamic ftrace

If CONFIG_DYNAMIC_FTRACE is set, the system will run with virtually no overhead when function tracing is disabled. The way this works is the mcount function call (placed at the start of every kernel function, produced by the -pg switch in gcc), starts of pointing to a simple return. (Enabling FTRACE will include the -pg switch in the compiling of the kernel.)

At compile time every C file object is run through the recordmcount program (located in the scripts directory). This program will parse the ELF headers in the C object to find all the locations in the .text section that call mcount. Starting with gcc version 4.6, the -mfentry has been added for x86, which calls "__fentry__" instead of "mcount". Which is called before the creation of the stack frame.

Note, not all sections are traced. They may be prevented by either a notrace, or blocked another way and all inline functions are not traced. Check the "available_filter_functions" file to see what functions can be traced.

A section called "__mcount_loc" is created that holds references to all the mcount/fentry call sites in the .text section. The recordmcount program re-links this section back into the original object. The final linking stage of the kernel will add all these references into a single table.

On boot up, before SMP is initialized, the dynamic ftrace code scans this table and updates all the locations into nops. It also records the locations, which are added to the available_filter_functions list. Modules are processed as they are loaded and before they are executed. When a module is unloaded, it also removes its functions from the ftrace function list. This is automatic in the module unload code, and the module author does not need to worry about it.

When tracing is enabled, the process of modifying the function tracepoints is dependent on architecture. The old method is to use kstop_machine to prevent races with the CPUs executing code being modified (which can cause the CPU to do undesirable things, especially if the modified code crosses cache (or page) boundaries), and the nops are patched back to calls. But this time, they do not call mcount (which is just a function stub). They now call into the ftrace infrastructure.

The new method of modifying the function tracepoints is to place a breakpoint at the location to be modified, sync all CPUs, modify the rest of the instruction not covered by the breakpoint. Sync all CPUs again, and then remove the breakpoint with the finished version to the ftrace call site.

Some archs do not even need to monkey around with the synchronization, and can just slap the new code on top of the old without any problems with other CPUs executing it at the same time.

One special side-effect to the recording of the functions being traced is that we can now selectively choose which functions we wish to trace and which ones we want the mcount calls to remain as nops.

Two files are used, one for enabling and one for disabling the tracing of specified functions. They are:

set_ftrace_filter

and

set_ftrace_notrace

A list of available functions that you can add to these files is listed in:

available_filter_functions

```
# cat available_filter_functions
put_prev_task_idle
kmem_cache_create
pick_next_task_rt
cpus_read_lock
pick_next_task_fair
mutex_lock
[...]
```

If I am only interested in sys_nanosleep and hrtimer_interrupt:

```
# echo sys_nanosleep hrtimer_interrupt > set_ftrace_filter
# echo function > current_tracer
# echo 1 > tracing_on
# usleep 1
# echo 0 > tracing_on
# cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 5/5   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          usleep-2665  [001] ....  4186.475355: sys_nanosleep <-system_call_fastpath
          <idle>-0     [001] d.h1  4186.475409: hrtimer_interrupt <-smp_apic_timer_interrupt
          usleep-2665  [001] d.h1  4186.475426: hrtimer_interrupt <-smp_apic_timer_interrupt
          <idle>-0     [003] d.h1  4186.475426: hrtimer_interrupt <-smp_apic_timer_interrupt
          <idle>-0     [002] d.h1  4186.475427: hrtimer_interrupt <-smp_apic_timer_interrupt
```

To see which functions are being traced, you can cat the file:

```
# cat set_ftrace_filter
hrtimer_interrupt
sys_nanosleep
```

Perhaps this is not enough. The filters also allow glob(7) matching.

`<match>*`
: will match functions that begin with <match>

`*<match>`
: will match functions that end with <match>

`*<match>*`
: will match functions that have <match> in it

`<match1>*<match2>`
: will match functions that begin with <match1> and end with <match2>

> **Note**
>
> It is better to use quotes to enclose the wild cards, otherwise the shell may expand the parameters into names of files in the local directory.

```
# echo 'hrtimer_*' > set_ftrace_filter
```

Produces:

```
# tracer: function
#
# entries-in-buffer/entries-written: 897/897   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          <idle>-0     [003] dN.1  4228.547803: hrtimer_cancel <-tick_nohz_idle_exit
          <idle>-0     [003] dN.1  4228.547804: hrtimer_try_to_cancel <-hrtimer_cancel
          <idle>-0     [003] dN.2  4228.547805: hrtimer_force_reprogram <-__remove_hrtimer
          <idle>-0     [003] dN.1  4228.547805: hrtimer_forward <-tick_nohz_idle_exit
          <idle>-0     [003] dN.1  4228.547805: hrtimer_start_range_ns <-hrtimer_start_expires.constprop.11
          <idle>-0     [003] d..1  4228.547858: hrtimer_get_next_event <-get_next_timer_interrupt
          <idle>-0     [003] d..1  4228.547859: hrtimer_start <-__tick_nohz_idle_enter
          <idle>-0     [003] d..2  4228.547860: hrtimer_force_reprogram <-__rem
```

Notice that we lost the sys_nanosleep.

```
# cat set_ftrace_filter
hrtimer_run_queues
hrtimer_run_pending
hrtimer_init
hrtimer_cancel
hrtimer_try_to_cancel
hrtimer_forward
hrtimer_start
hrtimer_reprogram
hrtimer_force_reprogram
hrtimer_get_next_event
hrtimer_interrupt
hrtimer_nanosleep
hrtimer_wakeup
hrtimer_get_remaining
hrtimer_get_res
hrtimer_init_sleeper
```

This is because the '>' and '>>' act just like they do in bash. To rewrite the filters, use '>' To append to the filters, use '>>'

To clear out a filter so that all functions will be recorded again:

```
# echo > set_ftrace_filter
# cat set_ftrace_filter
#
```

Again, now we want to append.

```
# echo sys_nanosleep > set_ftrace_filter
# cat set_ftrace_filter
sys_nanosleep
# echo 'hrtimer_*' >> set_ftrace_filter
# cat set_ftrace_filter
hrtimer_run_queues
hrtimer_run_pending
hrtimer_init
hrtimer_cancel
hrtimer_try_to_cancel
hrtimer_forward
hrtimer_start
hrtimer_reprogram
hrtimer_force_reprogram
hrtimer_get_next_event
hrtimer_interrupt
sys_nanosleep
hrtimer_nanosleep
hrtimer_wakeup
hrtimer_get_remaining
hrtimer_get_res
hrtimer_init_sleeper
```

The set_ftrace_notrace prevents those functions from being traced.

```
# echo '*preempt*' '*lock*' > set_ftrace_notrace
```

Produces:

```
# tracer: function
#
# entries-in-buffer/entries-written: 39608/39608   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
            bash-1994  [000] ....  4342.324896: file_ra_state_init <-do_dentry_open
            bash-1994  [000] ....  4342.324897: open_check_o_direct <-do_last
            bash-1994  [000] ....  4342.324897: ima_file_check <-do_last
            bash-1994  [000] ....  4342.324898: process_measurement <-ima_file_check
            bash-1994  [000] ....  4342.324898: ima_get_action <-process_measurement
            bash-1994  [000] ....  4342.324898: ima_match_policy <-ima_get_action
            bash-1994  [000] ....  4342.324899: do_truncate <-do_last
            bash-1994  [000] ....  4342.324899: should_remove_suid <-do_truncate
            bash-1994  [000] ....  4342.324899: notify_change <-do_truncate
            bash-1994  [000] ....  4342.324900: current_fs_time <-notify_change
            bash-1994  [000] ....  4342.324900: current_kernel_time <-current_fs_time
            bash-1994  [000] ....  4342.324900: timespec_trunc <-current_fs_time
```

We can see that there's no more lock or preempt tracing.

Selecting function filters via index

Because processing of strings is expensive (the address of the function needs to be looked up before comparing to the string being passed in), an index can be used as well to enable functions. This is useful in the case of setting thousands of specific functions at a time. By passing in a list of numbers, no string processing will occur. Instead, the function at the specific location in the internal array (which corresponds to the functions in the "available_filter_functions" file), is selected.

```
# echo 1 > set_ftrace_filter
```

Will select the first function listed in "available_filter_functions"

```
# head -1 available_filter_functions
trace_initcall_finish_cb

# cat set_ftrace_filter
trace_initcall_finish_cb

# head -50 available_filter_functions | tail -1
x86_pmu_commit_txn

# echo 1 50 > set_ftrace_filter
# cat set_ftrace_filter
trace_initcall_finish_cb
x86_pmu_commit_txn
```

Dynamic ftrace with the function graph tracer

Although what has been explained above concerns both the function tracer and the function-graph-tracer, there are some special features only available in the function-graph tracer.

If you want to trace only one function and all of its children, you just have to echo its name into set_graph_function:

```
echo __do_fault > set_graph_function
```

will produce the following "expanded" trace of the __do_fault() function:

```
0)                |  __do_fault() {
0)                |    filemap_fault() {
0)                |      find_lock_page() {
0)   0.804 us     |        find_get_page();
0)                |        __might_sleep() {
0)   1.329 us     |        }
0)   3.904 us     |      }
0)   4.979 us     |    }
0)   0.653 us     |    _spin_lock();
0)   0.578 us     |    page_add_file_rmap();
0)   0.525 us     |    native_set_pte_at();
0)   0.585 us     |    _spin_unlock();
0)                |    unlock_page() {
0)   0.541 us     |      page_waitqueue();
0)   0.639 us     |      __wake_up_bit();
0)   2.786 us     |    }
0) + 14.237 us    |  }
0)                |  __do_fault() {
0)                |    filemap_fault() {
0)                |      find_lock_page() {
0)   0.698 us     |        find_get_page();
0)                |        __might_sleep() {
0)   1.412 us     |        }
0)   3.950 us     |      }
0)   5.098 us     |    }
0)   0.631 us     |    _spin_lock();
0)   0.571 us     |    page_add_file_rmap();
0)   0.526 us     |    native_set_pte_at();
0)   0.586 us     |    _spin_unlock();
0)                |    unlock_page() {
0)   0.533 us     |      page_waitqueue();
0)   0.638 us     |      __wake_up_bit();
0)   2.793 us     |    }
0) + 14.012 us    |  }
```

You can also expand several functions at once:

```
echo sys_open > set_graph_function
echo sys_close >> set_graph_function
```

Now if you want to go back to trace all functions you can clear this special filter via:

```
echo > set_graph_function
```

ftrace_enabled

Note, the proc sysctl ftrace_enable is a big on/off switch for the function tracer. By default it is enabled (when function tracing is enabled in the kernel). If it is disabled, all function tracing is disabled. This includes not only the function tracers for ftrace, but also for any other uses (perf, kprobes, stack tracing, profiling, etc). It cannot be disabled if there is a callback with FTRACE_OPS_FL_PERMANENT set registered.

Please disable this with care.

This can be disable (and enabled) with:

```
 sysctl kernel.ftrace_enabled=0
 sysctl kernel.ftrace_enabled=1

or

 echo 0 > /proc/sys/kernel/ftrace_enabled
 echo 1 > /proc/sys/kernel/ftrace_enabled
```

Filter commands

A few commands are supported by the set_ftrace_filter interface. Trace commands have the following format:

```
<function>:<command>:<parameter>
```

The following commands are supported:

- mod: This command enables function filtering per module. The parameter defines the module. For example, if only the write* functions in the ext3 module are desired, run:

  echo 'write*:mod:ext3' > set_ftrace_filter

  This command interacts with the filter in the same way as filtering based on function names. Thus, adding more functions in a different module is accomplished by appending (>>) to the filter file. Remove specific module functions by prepending '!':

  ```
  echo '!writeback*:mod:ext3' >> set_ftrace_filter
  ```

  Mod command supports module globbing. Disable tracing for all functions except a specific module:

  ```
  echo '!*:mod:!ext3' >> set_ftrace_filter
  ```

  Disable tracing for all modules, but still trace kernel:

  ```
  echo '!*:mod:*' >> set_ftrace_filter
  ```

Enable filter only for kernel:

```
echo '*write*:mod:!*' >> set_ftrace_filter
```

Enable filter for module globbing:

```
echo '*write*:mod:*snd*' >> set_ftrace_filter
```

- traceon/traceoff: These commands turn tracing on and off when the specified functions are hit. The parameter determines how many times the tracing system is turned on and off. If unspecified, there is no limit. For example, to disable tracing when a schedule bug is hit the first 5 times, run:

  ```
  echo '__schedule_bug:traceoff:5' > set_ftrace_filter
  ```

  To always disable tracing when __schedule_bug is hit:

  ```
  echo '__schedule_bug:traceoff' > set_ftrace_filter
  ```

  These commands are cumulative whether or not they are appended to set_ftrace_filter. To remove a command, prepend it by '!' and drop the parameter:

  ```
  echo '!__schedule_bug:traceoff:0' > set_ftrace_filter
  ```

  The above removes the traceoff command for __schedule_bug that have a counter. To remove commands without counters:

  ```
  echo '!__schedule_bug:traceoff' > set_ftrace_filter
  ```

- snapshot: Will cause a snapshot to be triggered when the function is hit.

  ```
  echo 'native_flush_tlb_others:snapshot' > set_ftrace_filter
  ```

  To only snapshot once:

  ```
  echo 'native_flush_tlb_others:snapshot:1' > set_ftrace_filter
  ```

  To remove the above commands:

  ```
  echo '!native_flush_tlb_others:snapshot' > set_ftrace_filter
  echo '!native_flush_tlb_others:snapshot:0' > set_ftrace_filter
  ```

- enable_event/disable_event: These commands can enable or disable a trace event. Note, because function tracing callbacks are very sensitive, when these commands are registered, the trace point is activated, but disabled in a "soft" mode. That is, the tracepoint will be called, but just will not be traced. The event tracepoint stays in this mode as long as there's a command that triggers it.

  ```
  echo 'try_to_wake_up:enable_event:sched:sched_switch:2' > \
        set_ftrace_filter
  ```

  The format is:

  ```
  <function>:enable_event:<system>:<event>[:count]
  <function>:disable_event:<system>:<event>[:count]
  ```

  To remove the events commands:

  ```
  echo '!try_to_wake_up:enable_event:sched:sched_switch:0' > \
        set_ftrace_filter
  echo '!schedule:disable_event:sched:sched_switch' > \
        set_ftrace_filter
  ```

- dump: When the function is hit, it will dump the contents of the ftrace ring buffer to the console. This is useful if you need to debug something, and want to dump the trace when a certain function is hit. Perhaps it's a function that is called before a triple fault happens and does not allow you to get a regular dump.
- cpudump: When the function is hit, it will dump the contents of the ftrace ring buffer for the current CPU to the console. Unlike the "dump" command, it only prints out the contents of the ring buffer for the CPU that executed the function that triggered the dump.
- stacktrace: When the function is hit, a stack trace is recorded.

trace_pipe

The trace_pipe outputs the same content as the trace file, but the effect on the tracing is different. Every read from trace_pipe is consumed. This means that subsequent reads will be different. The trace is live.

```
# echo function > current_tracer
# cat trace_pipe > /tmp/trace.out &
[1] 4153
# echo 1 > tracing_on
# usleep 1
# echo 0 > tracing_on
# cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 0/0   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |

#
# cat /tmp/trace.out
           bash-1994  [000] ....  5281.568961: mutex_unlock <-rb_simple_write
           bash-1994  [000] ....  5281.568963: __mutex_unlock_slowpath <-mutex_unlock
           bash-1994  [000] ....  5281.568963: __fsnotify_parent <-fsnotify_modify
           bash-1994  [000] ....  5281.568964: fsnotify <-fsnotify_modify
           bash-1994  [000] ....  5281.568964: __srcu_read_lock <-fsnotify
           bash-1994  [000] ....  5281.568964: add_preempt_count <-__srcu_read_lock
           bash-1994  [000] ...1  5281.568965: sub_preempt_count <-__srcu_read_lock
           bash-1994  [000] ....  5281.568965: __srcu_read_unlock <-fsnotify
           bash-1994  [000] ....  5281.568967: sys_dup2 <-system_call_fastpath
```

Note, reading the trace_pipe file will block until more input is added. This is contrary to the trace file. If any process opened the trace file for reading, it will actually disable tracing and prevent new entries from being added. The trace_pipe file does not have this limitation.

trace entries

Having too much or not enough data can be troublesome in diagnosing an issue in the kernel. The file buffer_size_kb is used to modify the size of the internal trace buffers. The number listed is the number of entries that can be recorded per CPU. To know the full size, multiply the number of possible CPUs with the number of entries.

```
# cat buffer_size_kb
1408 (units kilobytes)
```

Or simply read buffer_total_size_kb

```
# cat buffer_total_size_kb
5632
```

To modify the buffer, simple echo in a number (in 1024 byte segments).

```
# echo 10000 > buffer_size_kb
```

```
# cat buffer_size_kb
10000 (units kilobytes)
```

It will try to allocate as much as possible. If you allocate too much, it can cause Out-Of-Memory to trigger.

```
# echo 1000000000000 > buffer_size_kb
-bash: echo: write error: Cannot allocate memory
# cat buffer_size_kb
85
```

The per_cpu buffers can be changed individually as well:

```
# echo 10000 > per_cpu/cpu0/buffer_size_kb
# echo 100 > per_cpu/cpu1/buffer_size_kb
```

When the per_cpu buffers are not the same, the buffer_size_kb at the top level will just show an X

```
# cat buffer_size_kb
X
```

This is where the buffer_total_size_kb is useful:

```
# cat buffer_total_size_kb
12916
```

Writing to the top level buffer_size_kb will reset all the buffers to be the same again.

Snapshot

CONFIG_TRACER_SNAPSHOT makes a generic snapshot feature available to all non latency tracers. (Latency tracers which record max latency, such as "irqsoff" or "wakeup", can't use this feature, since those are already using the snapshot mechanism internally.)

Snapshot preserves a current trace buffer at a particular point in time without stopping tracing. Ftrace swaps the current buffer with a spare buffer, and tracing continues in the new current (=previous spare) buffer.

The following tracefs files in "tracing" are related to this feature:

snapshot:

This is used to take a snapshot and to read the output of the snapshot. Echo 1 into this file to allocate a spare buffer and to take a snapshot (swap), then read the snapshot from this file in the same format as "trace" (described above in the section "The File System"). Both reads snapshot and tracing are executable in parallel. When the spare buffer is allocated, echoing 0 frees it, and echoing else (positive) values clear the snapshot contents. More details are shown in the table below.

status\input	0	1	else
not allocated	(do nothing)	alloc+swap	(do nothing)
allocated	free	swap	clear

Here is an example of using the snapshot feature.

```
# echo 1 > events/sched/enable
# echo 1 > snapshot
# cat snapshot
# tracer: nop
#
# entries-in-buffer/entries-written: 71/71   #P:8
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          <idle>-0     [005] d...  2440.603828: sched_switch: prev_comm=swapper/5 prev_pid=0 prev_prio=120   prev_state=R ==> next_comm=s
           sleep-2242  [005] d...  2440.603846: sched_switch: prev_comm=snapshot-test-2 prev_pid=2242 prev_prio=120   prev_state=R ==> ne
[...]
        <idle>-0     [002] d...  2440.707230: sched_switch: prev_comm=swapper/2 prev_pid=0 prev_prio=120 prev_state=R ==> next_comm=snaps
```

```
# cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 77/77   #P:8
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          <idle>-0     [007] d...  2440.707395: sched_switch: prev_comm=swapper/7 prev_pid=0 prev_prio=120 prev_state=R ==> next_comm=sna
 snapshot-test-2-2229  [002] d...  2440.707438: sched_switch: prev_comm=snapshot-test-2 prev_pid=2229 prev_prio=120 prev_state=S ==> next
[...]
```

If you try to use this snapshot feature when current tracer is one of the latency tracers, you will get the following results.

```
# echo wakeup > current_tracer
# echo 1 > snapshot
bash: echo: write error: Device or resource busy
# cat snapshot
cat: snapshot: Device or resource busy
```

Instances

In the tracefs tracing directory, there is a directory called "instances". This directory can have new directories created inside of it using mkdir, and removing directories with rmdir. The directory created with mkdir in this directory will already contain files and other directories after it is created.

```
# mkdir instances/foo
# ls instances/foo
buffer_size_kb  buffer_total_size_kb  events  free_buffer  per_cpu
set_event  snapshot  trace  trace_clock  trace_marker  trace_options
trace_pipe  tracing_on
```

As you can see, the new directory looks similar to the tracing directory itself. In fact, it is very similar, except that the buffer and events are agnostic from the main directory, or from any other instances that are created.

The files in the new directory work just like the files with the same name in the tracing directory except the buffer that is used is a separate and new buffer. The files affect that buffer but do not affect the main buffer with the exception of trace_options. Currently, the trace_options affect all instances and the top level buffer the same, but this may change in future releases. That is, options may become specific to the instance they reside in.

Notice that none of the function tracer files are there, nor is current_tracer and available_tracers. This is because the buffers can currently only have events enabled for them.

```
# mkdir instances/foo
# mkdir instances/bar
# mkdir instances/zoot
# echo 100000 > buffer_size_kb
# echo 1000 > instances/foo/buffer_size_kb
# echo 5000 > instances/bar/per_cpu/cpu1/buffer_size_kb
```

```
# echo function > current_trace
# echo 1 > instances/foo/events/sched/sched_wakeup/enable
# echo 1 > instances/foo/events/sched/sched_wakeup_new/enable
# echo 1 > instances/foo/events/sched/sched_switch/enable
# echo 1 > instances/bar/events/irq/enable
# echo 1 > instances/zoot/events/syscalls/enable
# cat trace_pipe
CPU:2 [LOST 11745 EVENTS]
            bash-2044  [002] .... 10594.481032: _raw_spin_lock_irqsave <-get_page_from_freelist
            bash-2044  [002] d... 10594.481032: add_preempt_count <-_raw_spin_lock_irqsave
            bash-2044  [002] d..1 10594.481032: __rmqueue <-get_page_from_freelist
            bash-2044  [002] d..1 10594.481033: _raw_spin_unlock <-get_page_from_freelist
            bash-2044  [002] d..1 10594.481033: sub_preempt_count <-_raw_spin_unlock
            bash-2044  [002] d... 10594.481033: get_pageblock_flags_group <-get_pageblock_migratetype
            bash-2044  [002] d... 10594.481034: __mod_zone_page_state <-get_page_from_freelist
            bash-2044  [002] d... 10594.481034: zone_statistics <-get_page_from_freelist
            bash-2044  [002] d... 10594.481034: __inc_zone_state <-zone_statistics
            bash-2044  [002] d... 10594.481034: __inc_zone_state <-zone_statistics
            bash-2044  [002] .... 10594.481035: arch_dup_task_struct <-copy_process
[...]

# cat instances/foo/trace_pipe
            bash-1998  [000] d..4   136.676759: sched_wakeup: comm=kworker/0:1 pid=59 prio=120 success=1 target_cpu=000
            bash-1998  [000] dN.4   136.676760: sched_wakeup: comm=bash pid=1998 prio=120 success=1 target_cpu=000
          <idle>-0     [003] d.h3   136.676906: sched_wakeup: comm=rcu_preempt pid=9 prio=120 success=1 target_cpu=003
          <idle>-0     [003] d..3   136.676909: sched_switch: prev_comm=swapper/3 prev_pid=0 prev_prio=120 prev_state=R ==> next_comm=rcu
     rcu_preempt-9     [003] d..3   136.676916: sched_switch: prev_comm=rcu_preempt prev_pid=9 prev_prio=120 prev_state=S ==> next_comm=s
            bash-1998  [000] d..4   136.677014: sched_wakeup: comm=kworker/0:1 pid=59 prio=120 success=1 target_cpu=000
            bash-1998  [000] dN.4   136.677016: sched_wakeup: comm=bash pid=1998 prio=120 success=1 target_cpu=000
            bash-1998  [000] d..3   136.677018: sched_switch: prev_comm=bash prev_pid=1998 prev_prio=120 prev_state=R+ ==> next_comm=kwor
     kworker/0:1-59    [000] d..4   136.677022: sched_wakeup: comm=sshd pid=1995 prio=120 success=1 target_cpu=001
     kworker/0:1-59    [000] d..3   136.677025: sched_switch: prev_comm=kworker/0:1 prev_pid=59 prev_prio=120 prev_state=S ==> next_comm=
[...]

# cat instances/bar/trace_pipe
     migration/1-14    [001] d.h3   138.732674: softirq_raise: vec=3 [action=NET_RX]
          <idle>-0     [001] dNh3   138.732725: softirq_raise: vec=3 [action=NET_RX]
            bash-1998  [000] d.h1   138.733101: softirq_raise: vec=1 [action=TIMER]
            bash-1998  [000] d.h1   138.733102: softirq_raise: vec=9 [action=RCU]
            bash-1998  [000] ..s2   138.733105: softirq_entry: vec=1 [action=TIMER]
            bash-1998  [000] ..s2   138.733106: softirq_exit: vec=1 [action=TIMER]
            bash-1998  [000] ..s2   138.733106: softirq_entry: vec=9 [action=RCU]
            bash-1998  [000] ..s2   138.733109: softirq_exit: vec=9 [action=RCU]
            sshd-1995  [001] d.h1   138.733278: irq_handler_entry: irq=21 name=uhci_hcd:usb4
            sshd-1995  [001] d.h1   138.733280: irq_handler_exit: irq=21 ret=unhandled
            sshd-1995  [001] d.h1   138.733281: irq_handler_entry: irq=21 name=eth0
            sshd-1995  [001] d.h1   138.733283: irq_handler_exit: irq=21 ret=handled
[...]

# cat instances/zoot/trace
# tracer: nop
#
# entries-in-buffer/entries-written: 18996/18996   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
            bash-1998  [000] d...   140.733501: sys_write -> 0x2
            bash-1998  [000] d...   140.733504: sys_dup2(oldfd: a, newfd: 1)
            bash-1998  [000] d...   140.733506: sys_dup2 -> 0x1
            bash-1998  [000] d...   140.733508: sys_fcntl(fd: a, cmd: 1, arg: 0)
            bash-1998  [000] d...   140.733509: sys_fcntl -> 0x1
            bash-1998  [000] d...   140.733510: sys_close(fd: a)
            bash-1998  [000] d...   140.733510: sys_close -> 0x0
            bash-1998  [000] d...   140.733514: sys_rt_sigprocmask(how: 0, nset: 0, oset: 6e2768, sigsetsize: 8)
            bash-1998  [000] d...   140.733515: sys_rt_sigprocmask -> 0x0
            bash-1998  [000] d...   140.733516: sys_rt_sigaction(sig: 2, act: 7fff718846f0, oact: 7fff71884650, sigsetsize: 8)
            bash-1998  [000] d...   140.733516: sys_rt_sigaction -> 0x0
```

You can see that the trace of the top most trace buffer shows only the function tracing. The foo instance displays wakeups and task switches.

To remove the instances, simply delete their directories:

```
# rmdir instances/foo
# rmdir instances/bar
# rmdir instances/zoot
```

Note, if a process has a trace file open in one of the instance directories, the rmdir will fail with EBUSY.

Stack trace

Since the kernel has a fixed sized stack, it is important not to waste it in functions. A kernel developer must be conscience of what they allocate on the stack. If they add too much, the system can be in danger of a stack overflow, and corruption will occur, usually leading to a system panic.

There are some tools that check this, usually with interrupts periodically checking usage. But if you can perform a check at every function call that will become very useful. As ftrace provides a function tracer, it makes it convenient to check the stack size at every function call. This is enabled via the stack tracer.

CONFIG_STACK_TRACER enables the ftrace stack tracing functionality. To enable it, write a '1' into /proc/sys/kernel/stack_tracer_enabled.

```
# echo 1 > /proc/sys/kernel/stack_tracer_enabled
```

You can also enable it from the kernel command line to trace the stack size of the kernel during boot up, by adding "stacktrace" to the kernel command line parameter.

After running it for a few minutes, the output looks like:

```
# cat stack_max_size
2928

# cat stack_trace
        Depth    Size   Location    (18 entries)
        -----    ----   --------
  0)     2928     224   update_sd_lb_stats+0xbc/0x4ac
  1)     2704     160   find_busiest_group+0x31/0x1f1
  2)     2544     256   load_balance+0xd9/0x662
  3)     2288      80   idle_balance+0xbb/0x130
  4)     2208     128   __schedule+0x26e/0x5b9
  5)     2080      16   schedule+0x64/0x66
  6)     2064     128   schedule_timeout+0x34/0xe0
  7)     1936     112   wait_for_common+0x97/0xf1
  8)     1824      16   wait_for_completion+0x1d/0x1f
  9)     1808     128   flush_work+0xfe/0x119
 10)     1680      16   tty_flush_to_ldisc+0x1e/0x20
 11)     1664      48   input_available_p+0x1d/0x5c
 12)     1616      48   n_tty_poll+0x6d/0x134
 13)     1568      64   tty_poll+0x64/0x7f
 14)     1504     880   do_select+0x31e/0x511
 15)      624     400   core_sys_select+0x177/0x216
 16)      224      96   sys_select+0x91/0xb9
 17)      128     128   system_call_fastpath+0x16/0x1b
```

Note, if -mfentry is being used by gcc, functions get traced before they set up the stack frame. This means that leaf level functions are not tested by the stack tracer when -mfentry is used.

Currently, -mfentry is used by gcc 4.6.0 and above on x86 only.

More

More details can be found in the source code, in the *kernel/trace/*.c* files.