ftrace - Function Tracer

Copyright 2008 Red Hat Inc.
Author: Steven Rostedt <srostedt@redhat.com>
License: The GNU Free Documentation License, Version 1.2
(dual licensed under the GPL v2)
Reviewers: Elias Oltmanns, Randy Dunlap, Andrew Morton,
John Kacur, and David Teigland.
Written for: 2.6.28-rc2

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 the function tracer, it also includes an infrastructure that allows for other types of tracing. Some of the tracers that are currently in ftrace include a tracer to trace context switches, the time it takes for a high priority task to run after it was woken up, the time interrupts are disabled, and more (ftrace allows for tracer plugins, which means that the list of tracers can always grow).

Implementation Details

See ftrace-design.txt for details for arch porters and such.

The File System

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

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

```
 debugfs       /sys/kernel/debug          debugfs defaults        0       0
```

Or you can mount it at run time with:

```
 mount -t debugfs nodev /sys/kernel/debug
```

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

```
 ln -s /sys/kernel/debug /debug
```

Any selected ftrace option will also create a directory called tracing

```
within the debugfs. The rest of the document will assume that you are in
the ftrace directory (cd /sys/kernel/debug/tracing) and will only concentrate
on the files within that directory and not distract from the content with
the extended "/sys/kernel/debug/tracing" path name.

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

After mounting the debugfs, you can see a directory called
"tracing".  This directory contains 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.

  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_enabled:

	This sets or displays whether the current_tracer
	is activated and tracing or not. Echo 0 into this
	file to disable the tracer or 1 to enable it.

  trace:

	This file holds the output of the trace in a human
	readable format (described below).

  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,they will display the same
	information every time they are read.

  trace_options:

	This file lets the user control the amount of data
	that is displayed in one of the above output
	files.
```

```
tracing_max_latency:

	Some of the tracers record the max latency.
	For example, the time interrupts are disabled.
	This 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)

buffer_size_kb:

	This sets or displays the number of kilobytes each CPU
	buffer can hold. The tracer 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).
	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.
	( Note, the size may not be a multiple of the page size
	  due to buffer management overhead. )

	This can only be updated when the current_tracer
	is set to "nop".

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

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 a single thread.

  set_graph_function:

    Set a "trigger" function where tracing should start
    with the function graph tracer (See the section
    "dynamic ftrace" for more details).

  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" or
    "set_ftrace_notrace". (See the section "dynamic ftrace"
    below for more details.)

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.

  "sched_switch"

    Traces the context switches and wakeups between tasks.

  "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.

  "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.

"hw-branch-tracer"

Uses the BTS CPU feature on x86 CPUs to traces all
branches executed.

"nop"

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

Examples of using the tracer

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

Output format:

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

```
                             --------
# tracer: function
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |
            bash-4251  [01] 10152.583854: path_put <-path_walk
            bash-4251  [01] 10152.583855: dput <-path_put
            bash-4251  [01] 10152.583855: _atomic_dec_and_lock <-dput
                             --------
```

A header is printed with the tracer name that is represented by
the trace. In this case the tracer is "function". Then a header
showing the format. Task name "bash", the task PID "4251", the
CPU that it was running on "01", the timestamp in <secs>.<usecs>
format, the function name that was traced "path_put" and the
parent function that called this function "path_walk". The
timestamp is the time at which the function was entered.

The sched_switch tracer also includes tracing of task wakeups
and context switches.

```
     ksoftirqd/1-7     [01]  1453.070013:      7:115:R   +  2916:115:S
     ksoftirqd/1-7     [01]  1453.070013:      7:115:R   +    10:115:S
```

```
     ksoftirqd/1-7     [01]  1453.070013:      7:115:R ==>    10:115:R
        events/1-10    [01]  1453.070013:     10:115:S ==>  2916:115:R
     kondemand/1-2916  [01]  1453.070013:   2916:115:S ==>     7:115:R
     ksoftirqd/1-7     [01]  1453.070013:      7:115:S ==>     0:140:R
```

Wake ups are represented by a "+" and the context switches are
shown as "==>". The format is:

```
 Context switches:

       Previous task              Next Task

  <pid>:<prio>:<state>  ==>  <pid>:<prio>:<state>

 Wake ups:

       Current task               Task waking up

  <pid>:<prio>:<state>    +  <pid>:<prio>:<state>
```

The prio is the internal kernel priority, which is the inverse
of the priority that is usually displayed by user-space tools.
Zero represents the highest priority (99). Prio 100 starts the
"nice" priorities with 100 being equal to nice -20 and 139 being
nice 19. The prio "140" is reserved for the idle task which is
the lowest priority thread (pid 0).

Latency trace format

When the latency-format option is enabled, 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 2.6.26-rc8
--------------------------------------------------------------------
 latency: 97 us, #3/3, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: swapper-0 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: apic_timer_interrupt
 => ended at:   do_softirq

#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /
#              |||||     delay
#  cmd     pid ||||| time  |   caller
#     \   /    |||||   \   |   /
  <idle>-0     0d..1    0us+: trace_hardirqs_off_thunk (apic_timer_interrupt)
```

```
  <idle>-0     0d.s.   97us : __do_softirq (do_softirq)
  <idle>-0     0d.s1   98us : trace_hardirqs_on (do_softirq)
```

This shows that the current tracer is "irqsoff" tracing the time for which interrupts were disabled. It gives the trace version and the version of the kernel upon which this was executed on (2.6.26-rc8). Then it displays the max latency in microsecs (97 us). The number of trace entries displayed and the total number recorded (both are three: #3/3). The type of preemption that was used (PREEMPT). VP, KP, SP, and HP are always zero and are reserved for later use. #P is the number of online CPUS (#P:2).

The task is the process that was running when the latency occurred. (swapper pid: 0).

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

```
  apic_timer_interrupt is where the interrupts were disabled.
  do_softirq 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.
            Note: If the architecture does not support a way to
                  read the irq flags variable, an 'X' will always
                  be printed here.

  need-resched: 'N' task need_resched is set, '.' otherwise.

  hardirq/softirq:
        '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 preempt_mark_thresh (default 100)
         '+' - greater than 1 microsecond
         ' ' - less than or equal to 1 microsecond.
```

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

trace_options

The trace_options file is used to control what gets printed in the trace output. To see what is available, simply cat the file:

```
  cat trace_options
  print-parent nosym-offset nosym-addr noverbose noraw nohex nobin \
  noblock nostacktrace nosched-tree nouserstacktrace nosym-userobj
```

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 <-strict_strtoul

  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 (strict_strtoul)

  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 - TBD (needs update)

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

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

  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]

  sched-tree - trace all tasks that are on the runqueue, at
               every scheduling event. Will add overhead if
               there's a lot of tasks running at once.

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

sched_switch

This tracer simply records schedule switches. Here is an example
of how to use it.

```
 # echo sched_switch > current_tracer
 # echo 1 > tracing_enabled
 # sleep 1
 # echo 0 > tracing_enabled
```

```
 # cat trace

# tracer: sched_switch
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |
            bash-3997  [01]   240.132281:   3997:120:R   +  4055:120:R
            bash-3997  [01]   240.132284:   3997:120:R ==>  4055:120:R
           sleep-4055  [01]   240.132371:   4055:120:S ==>  3997:120:R
            bash-3997  [01]   240.132454:   3997:120:R   +  4055:120:S
            bash-3997  [01]   240.132457:   3997:120:R ==>  4055:120:R
           sleep-4055  [01]   240.132460:   4055:120:D ==>  3997:120:R
            bash-3997  [01]   240.132463:   3997:120:R   +  4055:120:D
            bash-3997  [01]   240.132465:   3997:120:R ==>  4055:120:R
          <idle>-0     [00]   240.132589:      0:140:R   +     4:115:S
          <idle>-0     [00]   240.132591:      0:140:R ==>     4:115:R
     ksoftirqd/0-4     [00]   240.132595:      4:115:S ==>     0:140:R
          <idle>-0     [00]   240.132598:      0:140:R   +     4:115:S
          <idle>-0     [00]   240.132599:      0:140:R ==>     4:115:R
     ksoftirqd/0-4     [00]   240.132603:      4:115:S ==>     0:140:R
           sleep-4055  [01]   240.133058:   4055:120:S ==>  3997:120:R
 [...]
```

As we have discussed previously about this format, the header shows the name of the trace and points to the options. The "FUNCTION" is a misnomer since here it represents the wake ups and context switches.

The sched_switch file only lists the wake ups (represented with '+') and context switches ('==>') with the previous task or current task first followed by the next task or task waking up. The format for both of these is PID:KERNEL-PRIO:TASK-STATE. Remember that the KERNEL-PRIO is the inverse of the actual priority with zero (0) being the highest priority and the nice values starting at 100 (nice -20). Below is a quick chart to map the kernel priority to user land priorities.

Kernel Space	User Space
0(high) to 98(low)	user RT priority 99(high) to 1(low) with SCHED_RR or SCHED_FIFO
99	sched_priority is not used in scheduling decisions(it must be specified as 0)
100(high) to 139(low)	user nice -20(high) to 19(low)
140	idle task priority

The task states are:

```
 R - running : wants to run, may not actually be running
 S - sleep   : process is waiting to be woken up (handles signals)
 D - disk sleep (uninterruptible sleep) : process must be woken up
```

```
                              (ignores signals)
 T - stopped : process suspended
 t - traced  : process is being traced (with something like gdb)
 Z - zombie  : process waiting to be cleaned up
 X - unknown
```

ftrace_enabled

The following tracers (listed below) give different output depending on whether or not the sysctl ftrace_enabled is set. To set ftrace_enabled, one can either use the sysctl function or set it via the proc file system interface.

```
  sysctl kernel.ftrace_enabled=1

 or

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

To disable ftrace_enabled simply replace the '1' with '0' in the above commands.

When ftrace_enabled is set the tracers will also record the functions that are within the trace. The descriptions of the tracers will also show an example with ftrace enabled.

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 irqsoff > current_tracer
 # echo latency-format > trace_options
 # echo 0 > tracing_max_latency
 # echo 1 > tracing_enabled
 # ls -ltr
 [...]
 # echo 0 > tracing_enabled
 # cat trace
# tracer: irqsoff
```

```
#
irqsoff latency trace v1.1.5 on 2.6.26
--------------------------------------------------------------------
 latency: 12 us, #3/3, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: bash-3730 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: sys_setpgid
 => ended at:   sys_setpgid

#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /
#              |||||     delay
#  cmd     pid ||||| time  |   caller
#     \   /    |||||   \   |   /
    bash-3730  1d...    0us : _write_lock_irq (sys_setpgid)
    bash-3730  1d..1    1us+: _write_unlock_irq (sys_setpgid)
    bash-3730  1d..2   14us : trace_hardirqs_on (sys_setpgid)
```

Here we see that that we had a latency of 12 microsecs (which is very good). The _write_lock_irq in sys_setpgid disabled interrupts. The difference between the 12 and the displayed timestamp 14us 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 ftrace_enabled not set. If we set the ftrace_enabled, we get a much larger output:

```
# tracer: irqsoff
#
irqsoff latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 50 us, #101/101, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: ls-4339 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: __alloc_pages_internal
 => ended at:   __alloc_pages_internal

#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /
#              |||||     delay
#  cmd     pid ||||| time  |   caller
#     \   /    |||||   \   |   /
      ls-4339  0...1    0us+: get_page_from_freelist (__alloc_pages_internal)
      ls-4339  0d..1    3us : rmqueue_bulk (get_page_from_freelist)
```

```
      ls-4339  0d..1    3us : _spin_lock (rmqueue_bulk)
      ls-4339  0d..1    4us : add_preempt_count (_spin_lock)
      ls-4339  0d..2    4us : __rmqueue (rmqueue_bulk)
      ls-4339  0d..2    5us : __rmqueue_smallest (__rmqueue)
      ls-4339  0d..2    5us : __mod_zone_page_state (__rmqueue_smallest)
      ls-4339  0d..2    6us : __rmqueue (rmqueue_bulk)
      ls-4339  0d..2    6us : __rmqueue_smallest (__rmqueue)
      ls-4339  0d..2    7us : __mod_zone_page_state (__rmqueue_smallest)
      ls-4339  0d..2    7us : __rmqueue (rmqueue_bulk)
      ls-4339  0d..2    8us : __rmqueue_smallest (__rmqueue)
[...]
      ls-4339  0d..2   46us : __rmqueue_smallest (__rmqueue)
      ls-4339  0d..2   47us : __mod_zone_page_state (__rmqueue_smallest)
      ls-4339  0d..2   47us : __rmqueue (rmqueue_bulk)
      ls-4339  0d..2   48us : __rmqueue_smallest (__rmqueue)
      ls-4339  0d..2   48us : __mod_zone_page_state (__rmqueue_smallest)
      ls-4339  0d..2   49us : _spin_unlock (rmqueue_bulk)
      ls-4339  0d..2   49us : sub_preempt_count (_spin_unlock)
      ls-4339  0d..1   50us : get_page_from_freelist (__alloc_pages_internal)
      ls-4339  0d..2   51us : trace_hardirqs_on (__alloc_pages_internal)
```

Here we traced a 50 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.

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 preemptoff > current_tracer
 # echo latency-format > trace_options
 # echo 0 > tracing_max_latency
 # echo 1 > tracing_enabled
 # ls -ltr
 [...]
 # echo 0 > tracing_enabled
 # cat trace
# tracer: preemptoff
#
preemptoff latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 29 us, #3/3, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
```

```
    -----------------
    | task: sshd-4261 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: do_IRQ
 => ended at:   __do_softirq

#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /
#              |||||     delay
#  cmd     pid ||||| time  |   caller
#     \   /    |||||   \   |   /
    sshd-4261  0d.h.    0us+: irq_enter (do_IRQ)
    sshd-4261  0d.s.   29us : _local_bh_enable (__do_softirq)
    sshd-4261  0d.s1   30us : trace_preempt_on (__do_softirq)
```

This has some more changes. Preemption was disabled when an
interrupt came in (notice the 'h'), and was enabled while doing
a softirq. (notice the 's'). 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.

```
# tracer: preemptoff
#
preemptoff latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 63 us, #87/87, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: sshd-4261 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: remove_wait_queue
 => ended at:   __do_softirq

#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /
#              |||||     delay
#  cmd     pid ||||| time  |   caller
#     \   /    |||||   \   |   /
    sshd-4261  0d..1    0us : _spin_lock_irqsave (remove_wait_queue)
    sshd-4261  0d..1    1us : _spin_unlock_irqrestore (remove_wait_queue)
    sshd-4261  0d..1    2us : do_IRQ (common_interrupt)
    sshd-4261  0d..1    2us : irq_enter (do_IRQ)
    sshd-4261  0d..1    2us : idle_cpu (irq_enter)
    sshd-4261  0d..1    3us : add_preempt_count (irq_enter)
    sshd-4261  0d.h1    3us : idle_cpu (irq_enter)
    sshd-4261  0d.h.    4us : handle_fasteoi_irq (do_IRQ)
[...]
```

```
    sshd-4261  0d.h.   12us : add_preempt_count (_spin_lock)
    sshd-4261  0d.h1   12us : ack_ioapic_quirk_irq (handle_fasteoi_irq)
    sshd-4261  0d.h1   13us : move_native_irq (ack_ioapic_quirk_irq)
    sshd-4261  0d.h1   13us : _spin_unlock (handle_fasteoi_irq)
    sshd-4261  0d.h1   14us : sub_preempt_count (_spin_unlock)
    sshd-4261  0d.h1   14us : irq_exit (do_IRQ)
    sshd-4261  0d.h1   15us : sub_preempt_count (irq_exit)
    sshd-4261  0d..2   15us : do_softirq (irq_exit)
    sshd-4261  0d...   15us : __do_softirq (do_softirq)
    sshd-4261  0d...   16us : __local_bh_disable (__do_softirq)
    sshd-4261  0d...   16us+: add_preempt_count (__local_bh_disable)
    sshd-4261  0d.s4   20us : add_preempt_count (__local_bh_disable)
    sshd-4261  0d.s4   21us : sub_preempt_count (local_bh_enable)
    sshd-4261  0d.s5   21us : sub_preempt_count (local_bh_enable)
[...]
    sshd-4261  0d.s6   41us : add_preempt_count (__local_bh_disable)
    sshd-4261  0d.s6   42us : sub_preempt_count (local_bh_enable)
    sshd-4261  0d.s7   42us : sub_preempt_count (local_bh_enable)
    sshd-4261  0d.s5   43us : add_preempt_count (__local_bh_disable)
    sshd-4261  0d.s5   43us : sub_preempt_count (local_bh_enable_ip)
    sshd-4261  0d.s6   44us : sub_preempt_count (local_bh_enable_ip)
    sshd-4261  0d.s5   44us : add_preempt_count (__local_bh_disable)
    sshd-4261  0d.s5   45us : sub_preempt_count (local_bh_enable)
[...]
    sshd-4261  0d.s.   63us : _local_bh_enable (__do_softirq)
    sshd-4261  0d.s1   64us : trace_preempt_on (__do_softirq)
```

The above is an example of the preemptoff trace with ftrace_enabled set. Here we see that interrupts were 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.

Notice that __do_softirq when called does not have a preempt_count. It may seem that we missed a preempt enabling. What really happened is that the preempt count is held on the thread's stack and we switched to the softirq stack (4K stacks in effect). The code does not copy the preempt count, but because interrupts are disabled, we do not need to worry about it. Having a tracer like this is good for letting people know what really happens inside the kernel.

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 preemptirqsoff > current_tracer
 # echo latency-format > trace_options
 # echo 0 > tracing_max_latency
 # echo 1 > tracing_enabled
 # ls -ltr
 [...]
 # echo 0 > tracing_enabled
 # cat trace
# tracer: preemptirqsoff
#
preemptirqsoff latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 293 us, #3/3, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: ls-4860 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: apic_timer_interrupt
 => ended at:   __do_softirq

#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /
#              |||||     delay
#  cmd     pid |||||   time  |   caller
#     \   /    |||||     \   |   /
      ls-4860  0d...    0us!: trace_hardirqs_off_thunk (apic_timer_interrupt)
      ls-4860  0d.s.  294us : _local_bh_enable (__do_softirq)
      ls-4860  0d.s1  294us : trace_preempt_on (__do_softirq)
```

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 ftrace_enabled set:

```
# tracer: preemptirqsoff
#
preemptirqsoff latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 105 us, #183/183, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: sshd-4261 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: write_chan
 => ended at:   __do_softirq

#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /
#              |||||     delay
#  cmd     pid |||||   time  |   caller
#     \   /    |||||     \   |   /
      ls-4473  0.N..    0us : preempt_schedule (write_chan)
      ls-4473  0dN.1    1us : _spin_lock (schedule)
      ls-4473  0dN.1    2us : add_preempt_count (_spin_lock)
      ls-4473  0d..2    2us : put_prev_task_fair (schedule)
[...]
      ls-4473  0d..2   13us : set_normalized_timespec (ktime_get_ts)
      ls-4473  0d..2   13us : __switch_to (schedule)
    sshd-4261  0d..2   14us : finish_task_switch (schedule)
    sshd-4261  0d..2   14us : _spin_unlock_irq (finish_task_switch)
    sshd-4261  0d..1   15us : add_preempt_count (_spin_lock_irqsave)
    sshd-4261  0d..2   16us : _spin_unlock_irqrestore (hrtick_set)
    sshd-4261  0d..2   16us : do_IRQ (common_interrupt)
    sshd-4261  0d..2   17us : irq_enter (do_IRQ)
    sshd-4261  0d..2   17us : idle_cpu (irq_enter)
    sshd-4261  0d..2   18us : add_preempt_count (irq_enter)
    sshd-4261  0d.h2   18us : idle_cpu (irq_enter)
    sshd-4261  0d.h.   18us : handle_fasteoi_irq (do_IRQ)
    sshd-4261  0d.h.   19us : _spin_lock (handle_fasteoi_irq)
    sshd-4261  0d.h.   19us : add_preempt_count (_spin_lock)
    sshd-4261  0d.h1   20us : _spin_unlock (handle_fasteoi_irq)
    sshd-4261  0d.h1   20us : sub_preempt_count (_spin_unlock)
[...]
    sshd-4261  0d.h1   28us : _spin_unlock (handle_fasteoi_irq)
    sshd-4261  0d.h1   29us : sub_preempt_count (_spin_unlock)
    sshd-4261  0d.h2   29us : irq_exit (do_IRQ)
    sshd-4261  0d.h2   29us : sub_preempt_count (irq_exit)
```

```
    sshd-4261  0d..3   30us : do_softirq (irq_exit)
    sshd-4261  0d...   30us : __do_softirq (do_softirq)
    sshd-4261  0d...   31us : __local_bh_disable (__do_softirq)
    sshd-4261  0d...   31us+: add_preempt_count (__local_bh_disable)
    sshd-4261  0d.s4   34us : add_preempt_count (__local_bh_disable)
[...]
    sshd-4261  0d.s3   43us : sub_preempt_count (local_bh_enable_ip)
    sshd-4261  0d.s4   44us : sub_preempt_count (local_bh_enable_ip)
    sshd-4261  0d.s3   44us : smp_apic_timer_interrupt (apic_timer_interrupt)
    sshd-4261  0d.s3   45us : irq_enter (smp_apic_timer_interrupt)
    sshd-4261  0d.s3   45us : idle_cpu (irq_enter)
    sshd-4261  0d.s3   46us : add_preempt_count (irq_enter)
    sshd-4261  0d.H3   46us : idle_cpu (irq_enter)
    sshd-4261  0d.H3   47us : hrtimer_interrupt (smp_apic_timer_interrupt)
    sshd-4261  0d.H3   47us : ktime_get (hrtimer_interrupt)
[...]
    sshd-4261  0d.H3   81us : tick_program_event (hrtimer_interrupt)
    sshd-4261  0d.H3   82us : ktime_get (tick_program_event)
    sshd-4261  0d.H3   82us : ktime_get_ts (ktime_get)
    sshd-4261  0d.H3   83us : getnstimeofday (ktime_get_ts)
    sshd-4261  0d.H3   83us : set_normalized_timespec (ktime_get_ts)
    sshd-4261  0d.H3   84us : clockevents_program_event (tick_program_event)
    sshd-4261  0d.H3   84us : lapic_next_event (clockevents_program_event)
    sshd-4261  0d.H3   85us : irq_exit (smp_apic_timer_interrupt)
    sshd-4261  0d.H3   85us : sub_preempt_count (irq_exit)
    sshd-4261  0d.s4   86us : sub_preempt_count (irq_exit)
    sshd-4261  0d.s3   86us : add_preempt_count (__local_bh_disable)
[...]
    sshd-4261  0d.s1   98us : sub_preempt_count (net_rx_action)
    sshd-4261  0d.s.   99us : add_preempt_count (_spin_lock_irq)
    sshd-4261  0d.s1   99us+: _spin_unlock_irq (run_timer_softirq)
    sshd-4261  0d.s.  104us : _local_bh_enable (__do_softirq)
    sshd-4261  0d.s.  104us : sub_preempt_count (_local_bh_enable)
    sshd-4261  0d.s.  105us : _local_bh_enable (__do_softirq)
    sshd-4261  0d.s1  105us : trace_preempt_on (__do_softirq)
```

This is a very interesting trace. It started with the preemption of the ls task. We see that the task had the "need_resched" bit set via the 'N' in the trace. Interrupts were disabled before the spin_lock at the beginning of the trace. We see that a schedule took place to run sshd. When the interrupts were enabled, we took an interrupt. On return from the interrupt handler, the softirq ran. We took another interrupt while running the softirq as we see from the capital 'H'.

wakeup

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 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.

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 wakeup > current_tracer
 # echo latency-format > trace_options
 # echo 0 > tracing_max_latency
 # echo 1 > tracing_enabled
 # chrt -f 5 sleep 1
 # echo 0 > tracing_enabled
 # cat trace
# tracer: wakeup
#
wakeup latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 4 us, #2/2, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: sleep-4901 (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     1d.h4    0us+: try_to_wake_up (wake_up_process)
  <idle>-0     1d..4    4us : schedule (cpu_idle)
```

Running this on an idle system, we see that it only took 4 microseconds to perform the task switch. Note, since the trace marker 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 4901 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.

Doing the same with chrt -r 5 and ftrace_enabled set.

```
# tracer: wakeup
#
wakeup latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 50 us, #60/60, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: sleep-4068 (uid:0 nice:0 policy:2 rt_prio:5)
    -----------------

#                 _------=> CPU#
#                / _-----=> irqs-off
#               | / _----=> need-resched
#               || / _---=> hardirq/softirq
#               ||| / _--=> preempt-depth
#               |||| /
#               |||||     delay
#  cmd     pid  ||||| time  |   caller
#     \   /     |||||   \   |   /
ksoftirq-7     1d.H3    0us : try_to_wake_up (wake_up_process)
ksoftirq-7     1d.H4    1us : sub_preempt_count (marker_probe_cb)
ksoftirq-7     1d.H3    2us : check_preempt_wakeup (try_to_wake_up)
ksoftirq-7     1d.H3    3us : update_curr (check_preempt_wakeup)
ksoftirq-7     1d.H3    4us : calc_delta_mine (update_curr)
ksoftirq-7     1d.H3    5us : __resched_task (check_preempt_wakeup)
ksoftirq-7     1d.H3    6us : task_wake_up_rt (try_to_wake_up)
ksoftirq-7     1d.H3    7us : _spin_unlock_irqrestore (try_to_wake_up)
[...]
ksoftirq-7     1d.H2   17us : irq_exit (smp_apic_timer_interrupt)
ksoftirq-7     1d.H2   18us : sub_preempt_count (irq_exit)
ksoftirq-7     1d.s3   19us : sub_preempt_count (irq_exit)
ksoftirq-7     1..s2   20us : rcu_process_callbacks (__do_softirq)
[...]
ksoftirq-7     1..s2   26us : __rcu_process_callbacks (rcu_process_callbacks)
ksoftirq-7     1d.s2   27us : _local_bh_enable (__do_softirq)
ksoftirq-7     1d.s2   28us : sub_preempt_count (_local_bh_enable)
ksoftirq-7     1.N.3   29us : sub_preempt_count (ksoftirqd)
ksoftirq-7     1.N.2   30us : _cond_resched (ksoftirqd)
ksoftirq-7     1.N.2   31us : __cond_resched (_cond_resched)
ksoftirq-7     1.N.2   32us : add_preempt_count (__cond_resched)
ksoftirq-7     1.N.2   33us : schedule (__cond_resched)
ksoftirq-7     1.N.2   33us : add_preempt_count (schedule)
ksoftirq-7     1.N.3   34us : hrtick_clear (schedule)
ksoftirq-7     1dN.3   35us : _spin_lock (schedule)
ksoftirq-7     1dN.3   36us : add_preempt_count (_spin_lock)
ksoftirq-7     1d..4   37us : put_prev_task_fair (schedule)
ksoftirq-7     1d..4   38us : update_curr (put_prev_task_fair)
[...]
ksoftirq-7     1d..5   47us : _spin_trylock (tracing_record_cmdline)
ksoftirq-7     1d..5   48us : add_preempt_count (_spin_trylock)
ksoftirq-7     1d..6   49us : _spin_unlock (tracing_record_cmdline)
ksoftirq-7     1d..6   49us : sub_preempt_count (_spin_unlock)
ksoftirq-7     1d..4   50us : schedule (__cond_resched)
```

The interrupt went off while running ksoftirqd. This task runs at SCHED_OTHER. Why did not we see the 'N' set early? This may be a harmless bug with x86_32 and 4K stacks. On x86_32 with 4K stacks configured, the interrupt and softirq run with their own stack. Some information is held on the top of the task's stack (need_resched and preempt_count are both stored there). The setting of the NEED_RESCHED bit is done directly to the task's stack, but the reading of the NEED_RESCHED is done by looking at the current stack, which in this case is the stack for the hard interrupt. This hides the fact that NEED_RESCHED has been set. We do not see the 'N' until we switch back to the task's assigned stack.

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.

```
 # sysctl kernel.ftrace_enabled=1
 # echo function > current_tracer
 # echo 1 > tracing_enabled
 # usleep 1
 # echo 0 > tracing_enabled
 # cat trace
# tracer: function
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |
            bash-4003  [00]   123.638713: finish_task_switch <-schedule
            bash-4003  [00]   123.638714: _spin_unlock_irq <-finish_task_switch
            bash-4003  [00]   123.638714: sub_preempt_count <-_spin_unlock_irq
            bash-4003  [00]   123.638715: hrtick_set <-schedule
            bash-4003  [00]   123.638715: _spin_lock_irqsave <-hrtick_set
            bash-4003  [00]   123.638716: add_preempt_count <-_spin_lock_irqsave
            bash-4003  [00]   123.638716: _spin_unlock_irqrestore <-hrtick_set
            bash-4003  [00]   123.638717: sub_preempt_count
<-_spin_unlock_irqrestore
            bash-4003  [00]   123.638717: hrtick_clear <-hrtick_set
            bash-4003  [00]   123.638718: sub_preempt_count <-schedule
            bash-4003  [00]   123.638718: sub_preempt_count <-preempt_schedule
            bash-4003  [00]   123.638719: wait_for_completion
<-__stop_machine_run
            bash-4003  [00]   123.638719: wait_for_common <-wait_for_completion
            bash-4003  [00]   123.638720: _spin_lock_irq <-wait_for_common
            bash-4003  [00]   123.638720: add_preempt_count <-_spin_lock_irq
[...]
```

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_enabled"), 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 -1 > 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_debugfs(void)
{
	static char debugfs[MAX_PATH+1];
	static int debugfs_found;
	char type[100];
	FILE *fp;

	if (debugfs_found)
		return debugfs;

	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",
		      debugfs, type) == 2) {
		if (strcmp(type, "debugfs") == 0)
			break;
	}
	fclose(fp);

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

	strcat(debugfs, "/tracing/");
	debugfs_found = 1;

	return debugfs;
}

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

hw-branch-tracer (x86 only)

This tracer uses the x86 last branch tracing hardware feature to collect a branch trace on all cpus with relatively low overhead.

The tracer uses a fixed-size circular buffer per cpu and only traces ring 0 branches. The trace file dumps that buffer in the following format:

```
# tracer: hw-branch-tracer
#
# CPU#        TO  <-  FROM
   0  scheduler_tick+0xb5/0x1bf   <-  task_tick_idle+0x5/0x6
   2  run_posix_cpu_timers+0x2b/0x72a     <-  run_posix_cpu_timers+0x25/0x72a
   0  scheduler_tick+0x139/0x1bf          <-  scheduler_tick+0xed/0x1bf
   0  scheduler_tick+0x17c/0x1bf          <-  scheduler_tick+0x148/0x1bf
   2  run_posix_cpu_timers+0x9e/0x72a     <-  run_posix_cpu_timers+0x5e/0x72a
   0  scheduler_tick+0x1b6/0x1bf          <-  scheduler_tick+0x1aa/0x1bf
```

The tracer may be used to dump the trace for the oops'ing cpu on a kernel oops into the system log. To enable this, ftrace_dump_on_oops must be set. To set ftrace_dump_on_oops, one can either use the sysctl function or set it via the proc system interface.

```
  sysctl kernel.ftrace_dump_on_oops=n
```

or

```
  echo n > /proc/sys/kernel/ftrace_dump_on_oops
```

If n = 1, ftrace will dump buffers of all CPUs, if n = 2 ftrace will only dump the buffer of the CPU that triggered the oops.

Here's an example of such a dump after a null pointer dereference in a kernel module:

```
[57848.105921] BUG: unable to handle kernel NULL pointer dereference at
0000000000000000
[57848.106019] IP: [<ffffffffa0000006>] open+0x6/0x14 [oops]
[57848.106019] PGD 2354e9067 PUD 2375e7067 PMD 0
[57848.106019] Oops: 0002 [#1] SMP
[57848.106019] last sysfs file:
/sys/devices/pci0000:00/0000:00:1e.0/0000:20:05.0/local_cpus
[57848.106019] Dumping ftrace buffer:
[57848.106019] ---------------------------------
[...]
[57848.106019]    0  chrdev_open+0xe6/0x165       <-  cdev_put+0x23/0x24
[57848.106019]    0  chrdev_open+0x117/0x165      <-  chrdev_open+0xfa/0x165
[57848.106019]    0  chrdev_open+0x120/0x165      <-  chrdev_open+0x11c/0x165
[57848.106019]    0  chrdev_open+0x134/0x165      <-  chrdev_open+0x12b/0x165
[57848.106019]    0  open+0x0/0x14 [oops]         <-  chrdev_open+0x144/0x165
[57848.106019]    0  page_fault+0x0/0x30          <-  open+0x6/0x14 [oops]
[57848.106019]    0  error_entry+0x0/0x5b         <-  page_fault+0x4/0x30
[57848.106019]    0  error_kernelspace+0x0/0x31   <-  error_entry+0x59/0x5b
[57848.106019]    0  error_sti+0x0/0x1    <-  error_kernelspace+0x2d/0x31
[57848.106019]    0  page_fault+0x9/0x30          <-  error_sti+0x0/0x1
[57848.106019]    0  do_page_fault+0x0/0x881      <-  page_fault+0x1a/0x30
[...]
[57848.106019]    0  do_page_fault+0x66b/0x881    <-  is_prefetch+0x1ee/0x1f2
[57848.106019]    0  do_page_fault+0x6e0/0x881    <-  do_page_fault+0x67a/0x881
[57848.106019]    0  oops_begin+0x0/0x96          <-  do_page_fault+0x6e0/0x881
[57848.106019]    0  trace_hw_branch_oops+0x0/0x2d        <-
oops_begin+0x9/0x96
[...]
[57848.106019]    0  ds_suspend_bts+0x2a/0xe3     <-  ds_suspend_bts+0x1a/0xe3
[57848.106019] ---------------------------------
[57848.106019] CPU 0
[57848.106019] Modules linked in: oops
[57848.106019] Pid: 5542, comm: cat Tainted: G        W  2.6.28 #23
[57848.106019] RIP: 0010:[<ffffffffa0000006>]  [<ffffffffa0000006>]
open+0x6/0x14 [oops]
[57848.106019] RSP: 0018:ffff880235457d48  EFLAGS: 00010246
[...]
```

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:

```
  0)               |    up_write() {
  0)    0.646 us   |      _spin_lock_irqsave();
  0)    0.684 us   |      _spin_unlock_irqrestore();
  0)    3.123 us   |    }
  0)    0.548 us   |    fput();
  0) + 58.628 us   |  }

  [...]

  0)               |      putname() {
  0)               |        kmem_cache_free() {
  0)    0.518 us   |          __phys_addr();
  0)    1.757 us   |        }
  0)    2.861 us   |      }
  0) ! 115.305 us  |    }
  0) ! 116.402 us  |  }
```

  + means that the function exceeded 10 usecs.
  ! means that the function exceeded 100 usecs.

- 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();
```

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.pl script (located in the scripts directory). This script will process the C object using objdump to find all the locations in the .text section that call mcount. (Note, only the .text section is processed, since processing other sections like .init.text may cause races due to those sections being freed).

A new section called "__mcount_loc" is created that holds references to all the mcount call sites in the .text section. This section is compiled back into the original object. The final linker 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, kstop_machine is called to prevent races with the CPUS executing code being modified (which can cause the CPU to do undesirable things), 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.

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
get_online_cpus
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_enabled
 # usleep 1
 # echo 0 > tracing_enabled
 # cat trace
# tracer: ftrace
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |
          usleep-4134  [00]  1317.070017: hrtimer_interrupt
<-smp_apic_timer_interrupt
          usleep-4134  [00]  1317.070111: sys_nanosleep <-syscall_call
          <idle>-0     [00]  1317.070115: 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 simple wild
cards. Only the following are currently available

```
  <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
```

These are the only wild cards which are supported.

```
  <match>*<match> will not work.
```

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: ftrace
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |
            bash-4003  [00]  1480.611794: hrtimer_init <-copy_process
            bash-4003  [00]  1480.611941: hrtimer_start <-hrtick_set
            bash-4003  [00]  1480.611956: hrtimer_cancel <-hrtick_clear
            bash-4003  [00]  1480.611956: hrtimer_try_to_cancel <-hrtimer_cancel
          <idle>-0     [00]  1480.612019: hrtimer_get_next_event
<-get_next_timer_interrupt
          <idle>-0     [00]  1480.612025: hrtimer_get_next_event
<-get_next_timer_interrupt
          <idle>-0     [00]  1480.612032: hrtimer_get_next_event
<-get_next_timer_interrupt
          <idle>-0     [00]  1480.612037: hrtimer_get_next_event
<-get_next_timer_interrupt
          <idle>-0     [00]  1480.612382: hrtimer_get_next_event
<-get_next_timer_interrupt
```

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: ftrace
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |
            bash-4043  [01]   115.281644: finish_task_switch <-schedule
            bash-4043  [01]   115.281645: hrtick_set <-schedule
            bash-4043  [01]   115.281645: hrtick_clear <-hrtick_set
            bash-4043  [01]   115.281646: wait_for_completion
<-__stop_machine_run
            bash-4043  [01]   115.281647: wait_for_common <-wait_for_completion
            bash-4043  [01]   115.281647: kthread_stop <-stop_machine_run
            bash-4043  [01]   115.281648: init_waitqueue_head <-kthread_stop
            bash-4043  [01]   115.281648: wake_up_process <-kthread_stop
            bash-4043  [01]   115.281649: try_to_wake_up <-wake_up_process
```

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

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
```

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
  ```

- 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
  ```

  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' > set_ftrace_filter
  ```

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_enabled
 # usleep 1
 # echo 0 > tracing_enabled
 # cat trace
# tracer: function
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |

 #
 # cat /tmp/trace.out
            bash-4043  [00] 41.267106: finish_task_switch <-schedule
            bash-4043  [00] 41.267106: hrtick_set <-schedule
            bash-4043  [00] 41.267107: hrtick_clear <-hrtick_set
            bash-4043  [00] 41.267108: wait_for_completion <-__stop_machine_run
            bash-4043  [00] 41.267108: wait_for_common <-wait_for_completion
            bash-4043  [00] 41.267109: kthread_stop <-stop_machine_run
            bash-4043  [00] 41.267109: init_waitqueue_head <-kthread_stop
            bash-4043  [00] 41.267110: wake_up_process <-kthread_stop
            bash-4043  [00] 41.267110: try_to_wake_up <-wake_up_process
            bash-4043  [00] 41.267111: select_task_rq_rt <-try_to_wake_up
```

Note, reading the trace_pipe file will block until more input is added. By changing the tracer, trace_pipe will issue an EOF. We needed to set the function tracer _before_ we "cat" the trace_pipe file.

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)
```

Note, to modify this, you must have tracing completely disabled. To do that, echo "nop" into the current_tracer. If the current_tracer is not set to "nop", an EINVAL error will be returned.

```
 # echo nop > current_tracer
 # echo 10000 > buffer_size_kb
 # cat buffer_size_kb
10000 (units kilobytes)
```

The number of pages which will be allocated is limited to a percentage of available memory. Allocating too much will produce an error.

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

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