1. Ftrace

Ftrace は、カーネル 2.6.27 からカーネルに実装されたトレーサです。

本章では、カーネル 3.1.1 での実行例を交えて、Ftrace の機能と使い方について説明します。

1.1. 概要

Ftrace はカーネル内部に実装されたトレーサで、すべてのカーネル関数をトレースすることができます。また、各種プラグインによって、単なる関数コールのトレースだけでなく、割り込み禁止時間の計測やスケジューリングのレイテンシを測定したり、スタックトレースなども行えます。さらに、トレース対象とする関数を指定したり、逆に特定の関数をトレース対象から除外する、あるいは特定の関数から呼ばれる子関数のネストに限定してトレースすることもできます。

サンプリング型のプロファイラより正確な分析が行え、またトレースを停止している時にはオーバーヘッドがなく、トレース実行中もチューニングされた実装により高速に動作します。

Ftrace を利用するには、カーネルが対応している必要があります。測定対象のカーネルで Ftrace が有効化されていない場合には、カーネルの再構築が必要になります。また制御インターフェイスとして使用するため、debugfs も必要です。

1.2. カーネルのコンフィグレーション

Ftrace を有効にするために必要となるコンフィグレーションについて次の表にまとめます。

menuconfig の項目名	マクロ名	説明
kernel hacking	-	-
Debug Filesystem	CONFIG_DEBUG_FS	debugfs を有効にします。
Tracers	CONFIG_FTRACE	Ftrace を有効にします。
Kernel Function Tracer	CONFIG_FUNCTION_TRACER	function トレーサを有効にします。
Kernel Function Graph Tracer	CONFIG_FUNCTION_GRAPH_TRACER	function_graph トレーサを有効にします。
Interrupts-off Latency Tracer	CONFIG_IRQSOFF_TRACER	irqsoff トレーサを有効にします。
Preemption-off Latency Tracer	CONFIG_PREEMPT_TRACER	preemptoff トレーサを有効にします。
Scheduling Latency Tracer	CONFIG_SCHED_TRACER	wakeup トレーサを有効にします。
Trace process context switches and events *	CONFIG_ENABLE_DEFAULT_TRACERS	デフォルトトレーサを有効にします。
Branch Profiling	-	次の 3 項目から 1 つを選びます。

No branch profiling	CONFIG_BRANCH_PROFILE_NONE	分岐プロファイラを無効にします。
Trace likely/unlikely profiler	CONFIG_PROFILE_ANNOTATED_BRANCHES	likely/unlikely プロファイラを有効にします。カーネル内のすべての likely または unlikely マクロをトレースします。
Profile all if conditionals	CONFIG_PROFILE_ALL_BRANCHES	全分岐プロファイラを有効にします。if 文の条件にヒットしたかどうかをすべて記録します。また、likely/unlikely プロファイラも有効になります。
Trace max stack	CONFIG_STACK_TRACER	stack トレーサを有効にします。
Support for tracing block IO actions	CONFIG_BLK_DEV_IO_TRACER	blk トレーサを有効にします。
enable/disable ftrace tracepoints dynamically **	CONFIG_DYNAMIC_FTRACE	動的 Ftrace を有効にします。
Kernel function profiler **	CONFIG_FUNCTION_PROFILER	function プロファイラを有効にします。

* デフォルトトレーサは他のトレーサと同時に有効にすることはできません

** function トレーサが有効な場合のみ有効にすることができます

1.3. trace-cmd

Ftrace は、debugfs 上の制御インターフェイスのファイルに対して echo や cat などのコマンドを使って動作させることができますが、より簡単で便利に操作することができる trace-cmd というツールが開発されています。

ここでは trace-cmd の使い方について説明します。

1.3.1. インストール

trace-cmd は git.kernel.org から入手できます。現時点では、v1.2 がリリースされています。

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git
trace-cmd-v1.2
Cloning into trace-cmd-v1.2...
remote: Counting objects: 4104, done.
remote: Compressing objects: 100% (1265/1265), done.
remote: Total 4104 (delta 2948), reused 3942 (delta 2835)
Receiving objects: 100% (4104/4104), 2.46 MiB | 1.27 MiB/s, done.
Resolving deltas: 100% (2948/2948), done.
$ ls
```

```
trace-cmd-v1.2
$
```

デフォルトでは/usr/local/以下にインストールされます。

```
$ make
$ sudo make install
```

GUI ツールを使用する場合は、以下のコマンドを追加で実行します。

```
$ make gui
$ sudo make install_gui
```

次のファイルがインストールされます。

```
/usr/local/bin/kernelshark
/usr/local/bin/trace-graph
/usr/local/bin/trace-cmd
/usr/local/bin/trace-view
/usr/local/share/kernelshark/html/images/*.png
/usr/local/share/kernelshark/html/index.html
/usr/local/share/man/man5/trace-cmd.dat.5
/usr/local/share/man/man1/trace-cmd-listen.1
/usr/local/share/man/man1/trace-cmd-check-events.1
/usr/local/share/man/man1/trace-cmd.1
/usr/local/share/man/man1/trace-cmd-record.1
/usr/local/share/man/man1/trace-cmd-report.1
/usr/local/share/man/man1/trace-cmd-options.1
/usr/local/share/man/man1/trace-cmd-list.1
/usr/local/share/man/man1/trace-cmd-extract.1
/usr/local/share/man/man1/trace-cmd-start.1
/usr/local/share/man/man1/trace-cmd-reset.1
/usr/local/share/man/man1/trace-cmd-stack.1
/usr/local/share/man/man1/trace-cmd-restore.1
/usr/local/share/man/man1/trace-cmd-stop.1
/usr/local/share/man/man1/trace-cmd-split.1
/usr/local/share/trace-cmd/plugins/plugin_mac80211.so
/usr/local/share/trace-cmd/plugins/plugin_jbd2.so
/usr/local/share/trace-cmd/plugins/plugin_kmem.so
/usr/local/share/trace-cmd/plugins/plugin_hrtimer.so
/usr/local/share/trace-cmd/plugins/plugin_function.so
/usr/local/share/trace-cmd/plugins/plugin_kvm.so
/usr/local/share/trace-cmd/plugins/plugin_sched_switch.so
/usr/local/share/trace-cmd/plugins/plugin_blk.so
```

trace-cmd はコマンドラインツール、kernelshark、trace-graph、trace-view の 3 つは GUI ツールです。

インストールパスを変更するには"prefix"でパスを指定します。

```
$ make prefix=$HOME/trace-tool
```

```
$ sudo make prefix=$HOME/trace-tool install
```

1.3.2. トレースの準備

実際に trace-cmd を実行する前に 2 点確認が必要です。

1 つ目は Ftrace の制御インターフェイスが存在しているかどうかです。

制御インターフェイスは debugfs 内に作られます。debugfs は通常、/sys/kernel/debug/にマウントされ、Ftrace の制御は/sys/kernel/debug/tracing/以下のファイルを使用します。

```
# cd /sys/kernel/debug/tracing
# ls
available_events            per_cpu             trace
available_filter_functions  printk_formats      trace_clock
available_tracers           README              trace_marker
buffer_size_kb              saved_cmdlines      trace_options
current_tracer              set_event           trace_pipe
dyn_ftrace_total_info       set_ftrace_filter   trace_stat
enabled_functions           set_ftrace_notrace  tracing_cpumask
events                      set_ftrace_pid      tracing_enabled
free_buffer                 set_graph_function  tracing_max_latency
function_profile_enabled    stack_max_size      tracing_on
options                     stack_trace         tracing_thresh
#
```

これらのファイルやディレクトリは、カーネルのコンフィグレーションによって上の例とは異なる場合があります。debugfs がマウントされていない場合はマウントしてください。

```
# mount -t debugfs none /sys/kernel/debug
```

2 つ目は sysctl の ftrace_enabled がセットされているかどうかです。関数コールトレースを実行するには ftrace_enabled に"1"がセットされていなければなりません。

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

1.3.3. トレースの実行

trace-cmd には 10 以上のサブコマンドがあります。

- `record`

  トレースを実行し、トレースデータをファイルに出力します。

- `report`

  トレースデータファイルを読み込み、人が読めるフォーマットで出力します。

- options

  report コマンドで使用できるプラグインオプションの一覧を表示します。

- start

  トレースデータのファイルへの出力を行わずにトレースを開始します。

- stop

  トレースを停止します。

- extract

  カーネルのトレースバッファの内容を取得し、ファイルに出力します。

- reset

  トレースを終了し、カーネルのトレースバッファもクリアします。

- split

  トレースデータファイルを小さなファイルに分割します。

- list

  利用可能なプラグインおよびイベントの一覧を表示します。

- listsen

  リモートホストからトレースデータを受信します。

- restore

  record コマンドが異常終了した時に残る CPU 毎のデータファイルからトレースデータファイルを構築します。

- stack

  スタックトレーサを実行します。

- check-events

  全イベントトレースのフォーマット文字列が解析可能かどうかをチェックします。

trace-cmd list

まず、利用可能なプラグイン(トレーサ)とイベントの一覧を確認します。

```
# trace-cmd list
events:
skb:kfree_skb
skb:consume_skb
skb:skb_copy_datagram_iovec
net:net_dev_xmit
```

```
net:net_dev_queue
net:netif_receive_skb
net:netif_rx
(省略)

plugins:
blk function_graph wakeup_rt wakeup preemptirqsoff preemptoff irqsoff function
nop

options:
print-parent
nosym-offset
nosym-addr
noverbose
(省略)
#
```

“events:”の後に表示されるものが利用可能なイベント、”plugins:”の次に表示されるものが利用可能なプラグイン、”options:”の後のものが利用可能なオプションです。内容はカーネルのコンフィグレーションによって異なる場合があります。

trace-cmd record

record コマンドは、指定されたプラグインまたはイベントのトレースと、トレースデータのファイルへの出力を開始します。そして指定されたコマンドを実行し、コマンドが終了するとトレースを停止します。コマンドを指定しなかった場合は<^C>が入力される間でトレースを実行します。

まず、function_graph プラグインを使用して、ls コマンド実行時のカーネル関数コールをトレースしてみます。

```
# trace-cmd record -p function_graph ls -ltr
  plugin 'function_graph'
total 0
Kernel buffer statistics:
  Note: "entries" are the entries left in the kernel ring buffer and are not
        recorded in the trace data. They should all be zero.

CPU: 0
entries: 0
overrun: 11561
commit overrun: 0

CPU: 1
entries: 0
overrun: 15844
commit overrun: 0
```

```
CPU0 data recorded at offset=0xeb000
    1585152 bytes in size
CPU1 data recorded at offset=0x26e000
    1572864 bytes in size
# ls -l
total 4024
-rw-r--r-- 1 root root 4120576 2011-11-23 20:05 trace.dat
#
```

トレースデータファイル"trace.dat"が作成されました。

このファイルはバイナリファイルで、そのままでは読むことができないので report コマンドでテキストに変換します。

```
# trace-cmd report
version = 6
cpus=2
             ls-3520  [000]   233.127763: funcgraph_entry:      + 11.761 us  |  __fsnotify_parent();
       trace-cmd-3519  [001]   233.127763: funcgraph_entry:      + 11.761 us  |  sub_preempt_count();
             ls-3520  [000]   233.127777: funcgraph_entry:                   |  fsnotify() {
       trace-cmd-3519  [001]   233.127778: funcgraph_entry:                   |  simple_lookup() {
             ls-3520  [000]   233.127782: funcgraph_entry:                   |    __srcu_read_lock() {
       trace-cmd-3519  [001]   233.127782: funcgraph_entry:        3.480 us   |    d_set_d_op();
             ls-3520  [000]   233.127785: funcgraph_entry:        3.120 us   |      add_preempt_count();
       trace-cmd-3519  [001]   233.127788: funcgraph_entry:                   |    d_instantiate() {
             ls-3520  [000]   233.127791: funcgraph_entry:        3.120 us   |      sub_preempt_count();
       trace-cmd-3519  [001]   233.127792: funcgraph_entry:                   |      __d_instantiate() {
       trace-cmd-3519  [001]   233.127795: funcgraph_entry:                   |        _raw_spin_lock() {
(省略)
#
```

タイムスタンプでソートされているため、2 つの CPU のトレースデータが混ざってしまっています。report コマンドのオプションで CPU を指定することができます。

```
# trace-cmd report --cpu 0|head
version = 6
cpus=2
             ls-3520  [000]   233.127763: funcgraph_entry:      + 11.761 us  |  __fsnotify_parent();
             ls-3520  [000]   233.127777: funcgraph_entry:                   |  fsnotify() {
             ls-3520  [000]   233.127782: funcgraph_entry:                   |    __srcu_read_lock() {
             ls-3520  [000]   233.127785: funcgraph_entry:        3.120 us   |      add_preempt_count();
             ls-3520  [000]   233.127791: funcgraph_entry:        3.120 us   |      sub_preempt_count();
             ls-3520  [000]   233.127797: funcgraph_exit:       + 15.600 us  |    }
             ls-3520  [000]   233.127800: funcgraph_entry:                   |    __srcu_read_unlock() {
             ls-3520  [000]   233.127803: funcgraph_entry:        3.120 us   |      add_preempt_count();
             ls-3520  [000]   233.127809: funcgraph_entry:        3.000 us   |      sub_preempt_count();
             ls-3520  [000]   233.127814: funcgraph_exit:       + 14.641 us  |    }
             ls-3520  [000]   233.127817: funcgraph_exit:       + 40.081 us  |  }
             ls-3520  [000]   233.127826: funcgraph_entry:                   |  do_page_fault() {
             ls-3520  [000]   233.127829: funcgraph_entry:                   |    down_read_trylock() {
             ls-3520  [000]   233.127833: funcgraph_entry:                   |      _raw_spin_lock_irqsave() {
```

```
            ls-3520  [000]   233.127836: funcgraph_entry:                   |        __raw_spin_lock_irqsave() {
(省略)
#
```

各カラムの意味は左から、タスク名-PID、CPU 番号、タイムスタンプ(<秒>.<μ 秒>)、データの種類、実行時間、関数名、となっています。

実行時間が 10μ 秒以上のものは'+'、100μ 秒以上のものには'!'が実行時間の前に表示されます。また、子関数の呼び出しがある関数の場合は"funcgraph_exit"の行に、子関数の呼び出しがないものは"funcgraph_entry"の行に関数の実行時間が表示されます。

次に、ext3 イベントを使用して、ls コマンド実行時のイベントをトレースしてみます。

```
# trace-cmd record -e ext3 ls -ltr
/sys/kernel/debug/tracing/events/ext3/filter
/sys/kernel/debug/tracing/events/*/ext3/filter
total 0
Kernel buffer statistics:
  Note: "entries" are the entries left in the kernel ring buffer and are not
        recorded in the trace data. They should all be zero.

CPU: 0
entries: 0
overrun: 0
commit overrun: 0

CPU: 1
entries: 0
overrun: 0
commit overrun: 0

CPU0 data recorded at offset=0xf2000
    4096 bytes in size
CPU1 data recorded at offset=0xf3000
    0 bytes in size
#
```

function_graph プラグインの時と同様に、report コマンドでトレースデータを表示します。

```
# trace-cmd report
version = 6
CPU 1 is empty
cpus=2
              ls-4808  [000]  1297.201561: ext3_get_blocks_enter: dev 179,26 ino 1072670 lblk 0 len 1 create 0
              ls-4808  [000]  1297.201576: ext3_get_blocks_exit: dev 179,26 ino 1072670 lblk 0 pblk 4304517 len 1 ret 1
              ls-4808  [000]  1297.201606: ext3_get_blocks_enter: dev 179,26 ino 1072670 lblk 2 len 1 create 0
              ls-4808  [000]  1297.201612: ext3_get_blocks_exit: dev 179,26 ino 1072670 lblk 2 pblk 4306894 len 1 ret 1
              ls-4808  [000]  1297.201680: ext3_get_blocks_enter: dev 179,26 ino 1062945 lblk 0 len 1 create 0
              ls-4808  [000]  1297.201685: ext3_get_blocks_exit: dev 179,26 ino 1062945 lblk 0 pblk 4268145 len 1 ret 1
              ls-4808  [000]  1297.201698: ext3_get_blocks_enter: dev 179,26 ino 1062945 lblk 6 len 1 create 0
```

```
            ls-4808  [000]  1297.201701: ext3_get_blocks_exit: dev 179,26 ino 1062945 lblk 6 pblk 4283066 len 1 ret 1
            ls-4808  [000]  1297.213276: ext3_get_blocks_enter: dev 179,26 ino 171706 lblk 0 len 1 create 0
            ls-4808  [000]  1297.213288: ext3_get_blocks_exit: dev 179,26 ino 171706 lblk 0 pblk 703559 len 1 ret 1
#
```

フォーマットは関数グラフトレーサと同様ですが、トレースデータの種別のところにイベント名、関数名のところにイベントの情報が表示されます。

trace-cmd listen

ターゲット上にトレースデータファイルを書き込むのに十分なストレージがない場合などに、トレースデータをネットワーク経由でリモートホストに送信することができます。その際、データを受けるホストで trace-cmd の listen コマンドを実行します。listen コマンドのオプションで待ち受けポート番号を指定します。

```
(remote)# trace-cmd listen -p 10101
```

ターゲット上では record コマンドのオプションでトレースデータを送信するホストを指定します。

```
# trace-cmd record -p function_graph -N 192.168.100.151:10101 ls -ltr
  plugin 'function_graph'
total 0
Kernel buffer statistics:
  Note: "entries" are the entries left in the kernel ring buffer and are not
        recorded in the trace data. They should all be zero.

CPU: 0
entries: 0
overrun: 22795
commit overrun: 0

CPU: 1
entries: 0
overrun: 28273
commit overrun: 0

#
```

リモートホスト側でトレースデータを受信し、データファイルが作成されます。

```
(remote)# trace-cmd listen -p 10101
connected!
Connected with armsmp.local:55191
cpus=2
pagesize=4096
version = 6
CPU0 data recorded at offset=0xeb000
    1449984 bytes in size
```

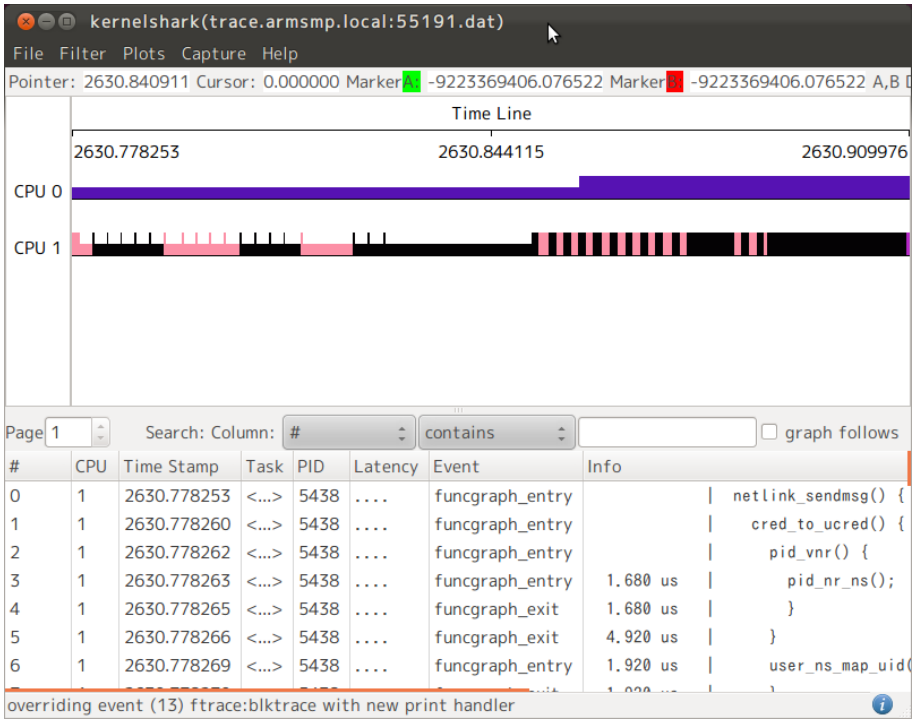
```
CPU1 data recorded at offset=0x24d000
    1556480 bytes in size
connected!
<^C>
(remote)# ls -l
合計 3876
-rw-r--r-- 1 root root 3969024 2011-11-24 05:45 trace.armsmp.local:55191.dat
(remote)#
```

リモートホスト上で report コマンドを実行してトレースデータを表示します。

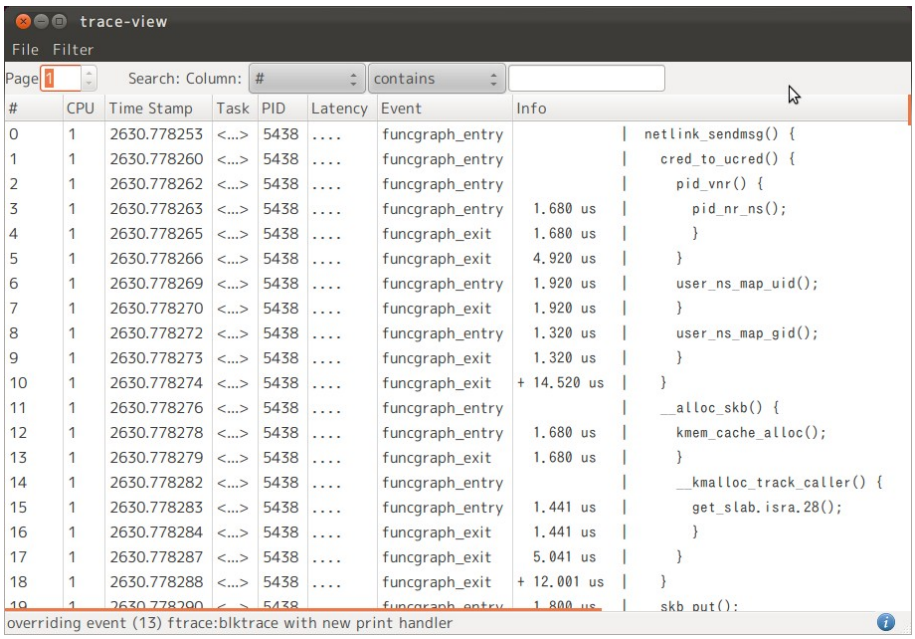
```
(remote)# trace-cmd report -i trace.armsmp.local\:55191.dat |head
version = 6
cpus=2
          <...>-5438  [001]  2630.778254: funcgraph_entry:                   |  netlink_sendmsg() {
          <...>-5438  [001]  2630.778260: funcgraph_entry:                   |    cred_to_ucred() {
          <...>-5438  [001]  2630.778262: funcgraph_entry:                   |      pid_vnr() {
          <...>-5438  [001]  2630.778264: funcgraph_entry:        1.680 us   |        pid_nr_ns();
          <...>-5438  [001]  2630.778267: funcgraph_exit:         4.920 us   |      }
          <...>-5438  [001]  2630.778269: funcgraph_entry:        1.920 us   |      user_ns_map_uid();
          <...>-5438  [001]  2630.778272: funcgraph_entry:        1.320 us   |      user_ns_map_gid();
          <...>-5438  [001]  2630.778275: funcgraph_exit:       + 14.520 us  |    }
(remote)#
```

また、リモートホスト上で GUI ツールが使えれば、受信データを kernelshark、trace-view、trace-graph で表示することもできます。

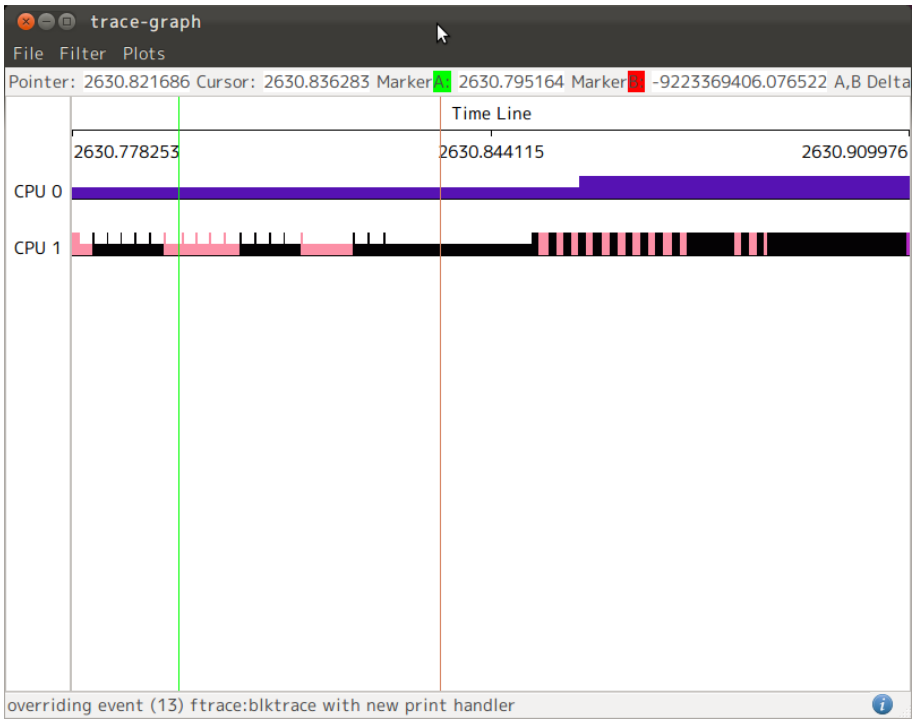
- kernelshark



- trace-view



- trace-graph



1.4. インターフェイス

Ftrace の制御インターフェイスについて説明します。

ここでは/sys/kernel/debug/tracing/の主なファイルについて説明します。

- `current_tracer`

  読み込むと現在選択されているトレーサ名が出力されます。また、トレーサ名を書き込むこ

とで使用するトレーサを指定します。指定できるトレーサ名は、available_tracers に列挙されます。

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

- available_tracers

使用できるトレーサの名称が読み込めます。使用できるトレーサは、カーネルのコンフィグレーションによって変わります。ここに列挙されたトレーサ名は、current_tracer に書き込むことで選択することができます。

```
# cat available_tracers
blk function_graph wakeup_rt wakeup preemptirqsoff preemptoff irqsoff function
nop
#
```

- tracing_on

"0"または"1"を書き込むことでトレースの実行を制御、読み込むことで状態を確認することができます。"0"を書き込むと停止、"1"を書き込むと実行します。

```
# cat tracing_on
1
# echo 0 > tracing_on
# cat tracing_on
0
#
```

- trace

トレース結果を取得することができます。このファイルでは、最も古いトレースデータから最新のデータまでがすべて読み込めます。また、新しいトレース結果が追加されない限り何度でも同じ内容が読み込めます。なお、トレーサを変更するとトレース結果は破棄されます。

```
# head trace
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
          <idle>-0     [001]  4989.846692: do_local_timer <-__irq_svc
          <idle>-0     [001]  4989.846698: twd_timer_ack <-do_local_timer
          <idle>-0     [001]  4989.846700: ipi_timer <-do_local_timer
          <idle>-0     [001]  4989.846703: irq_enter <-ipi_timer
          <idle>-0     [001]  4989.846705: rcu_irq_enter <-irq_enter
          <idle>-0     [001]  4989.846707: rcu_exit_nohz <-rcu_irq_enter
```

```
# tail trace
            bash-3377  [000]  4998.853160: _raw_spin_unlock <-sys_close
            bash-3377  [000]  4998.853163: sub_preempt_count <-_raw_spin_unlock
            bash-3377  [000]  4998.853165: filp_close <-sys_close
            bash-3377  [000]  4998.853167: dnotify_flush <-filp_close
            bash-3377  [000]  4998.853170: locks_remove_posix <-filp_close
            bash-3377  [000]  4998.853172: fput <-filp_close
            bash-3377  [000]  4998.853194: sys_write <-ret_fast_syscall
            bash-3377  [000]  4998.853196: fget_light <-sys_write
            bash-3377  [000]  4998.853199: vfs_write <-sys_write
            bash-3377  [000]  4998.853201: rw_verify_area <-vfs_write
#
```

- trace_pipe

  最新のトレース結果を継続して出力します。新しいトレースデータが追加されるまで読み込みはブロックされます。trace とは違い、このファイルはトレース結果を保持しません。つまり一度読み込んだトレースデータを再度読み込むことはできません。

- trace_options

  トレース結果に出力する項目、フォーマットをカスタマイズするためのオプションを設定または確認することができます。トレースオプションの詳細については「1.6 トレースオプション」および各トレーサの説明の中に記述します。

```
# cat trace_options
print-parent
nosym-offset
nosym-addr
noverbose
noraw
nohex
nobin
noblock
nostacktrace
trace_printk
noftrace_preempt
nobranch
annotate
nouserstacktrace
nosym-userobj
noprintk-msg-only
context-info
nolatency-format
sleep-time
graph-time
record-cmd
overwrite
```

```
nodisable_on_free
nofunc_stack_trace
#
```

- tracing_max_latency

いくつかのトレーサによって記録される最大待ち時間を取得できます。例えば irqsoff トレーサは最長の割り込み禁止時間を記録します。単位はμ秒です。"0"を書き込むことでリセットすることができます。

```
# echo 0 > tracing_max_latency
# echo irqsoff > current_tracer
# echo 1 > tracing_on
# cat tracing_max_latency
763
# cat tracing_max_latency
1245
#
```

- buffer_size_kb

CPU ごとに割り当てられるトレースバッファのサイズをキロバイト単位で設定、または表示します。各 CPU のバッファサイズは同一です。取得されるバッファサイズは 1 つの CPU に割り当てられるバッファサイズであり、全 CPU 分の合計値ではありません。

トレースバッファはページ単位で確保されるため、実際に割り当てられるバッファサイズは設定値より大きくなる場合があります。

current_tracer に nop が設定されている時のみバッファサイズを設定することができます。

```
# cat buffer_size_kb
1408
# echo 0 > buffer_size_kb
-bash: echo: write error: Invalid argument
#
```

- tracing_cpumask

トレースを実行する CPU を選択するためのビットマスクです。16 進文字列で指定します。

```
# cat tracing_cpumask
3
#
```

- set_ftrace_filter

カーネル関数名を書き込むと、その関数のみがトレースされます。複数の関数を指定することができます。

```
# echo function > current_tracer
# echo sys_open > set_ftrace_filter
# echo sys_close >> set_ftrace_filter
```

```
# echo 1 > tracing_on
# cat trace
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
            bash-3377  [001]  8083.339390: sys_close <-ret_fast_syscall
           sleep-6247  [001]  8086.899251: sys_close <-ret_fast_syscall
           sleep-6247  [001]  8086.899274: sys_close <-ret_fast_syscall
           sleep-6248  [001]  8086.911348: sys_open <-ret_fast_syscall
           sleep-6248  [001]  8086.911458: sys_close <-ret_fast_syscall
           sleep-6248  [001]  8086.911569: sys_open <-ret_fast_syscall
           sleep-6248  [001]  8086.911989: sys_close <-ret_fast_syscall
#
```

- **set_ftrace_notrace**

  set_ftrace_filter の逆で、ここに書き込まれた関数はトレースされません。set_ftrace_filter と set_ftrace_notrace の両方に書かれた場合、その関数はトレースされません。

- **set_ftrace_pid**

  PID を書き込むと function トレーサは該当スレッドのみをトレースします。

- **set_graph_function**

  function_graph トレーサがトレースを開始する起点となる関数を指定します。

- **available_filter_functions**

  Ftrace がトレースできる関数の一覧です。これらの関数名は set_ftrace_filter または set_ftrace_notrace に設定することができます。

```
# head available_filter_functions
do_one_initcall
rest_init
run_init_process
init_post
match_dev_by_uuid
name_to_dev_t
vfp_enable
vfp_pm_resume
vfp_pm_suspend
vfp_raise_sigfpe
# wc -l available_filter_functions
11780 available_filter_functions
#
```

1.5. トレーサ

ここでは、各トレーサについて説明します。

- function

  すべてのカーネル関数をトレースできる関数コールトレーサです。

- function_graph

  functionトレーサが関数の入り口でのみトレースするのに対し、function_graphトレーサは関数の入り口と出口でトレースします。これによって各関数の正確な実行時間を計測することができます。また、関数呼び出しのネストをC言語ライクなフォーマットで表示します。

- irqsoff

  割り込み禁止区間をトレースし、最大待ち時間を記録します。

- preemptoff

  irqsoffと同様に、プリエンプト禁止時間をトレース、記録します。

- preemptirqsoff

  割り込みあるいはプリエンプトの最大禁止時間をトレース、記録します。

- wakeup

  最優先タスクが起床してからスケジュールされるまでの最大待ち時間をトレース、記録します。

- nop

  なにもトレースしないトレーサです。トレーサを完全に止める時やトレースデータをクリアするのに使えます。

1.5.1. functionトレーサ

関数コールトレーサです。

```
# echo nop > current_tracer
# echo function > current_tracer
# echo 1 > tracing_on ; echo 0 > tracing_on
# cat trace
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
            bash-3345  [000]   729.397136: __fsnotify_parent <-vfs_write
            bash-3345  [000]   729.397143: fsnotify <-vfs_write
            bash-3345  [000]   729.397146: __srcu_read_lock <-fsnotify
```

```
           bash-3345  [000]   729.397148: add_preempt_count <-__srcu_read_lock
           bash-3345  [000]   729.397151: sub_preempt_count <-__srcu_read_lock
           bash-3345  [000]   729.397153: __srcu_read_unlock <-fsnotify
           bash-3345  [000]   729.397156: add_preempt_count <-__srcu_read_unlock
(省略)
#
```

上の実行例の 7 行目以降のトレース出力部分について説明します。

先頭の 4 行はヘッダです。実行されたトレーサが function であることと各カラムの説明が出力されます。5 行目以降がトレースデータです。

タスク名が"bash"、PID は"3345"、実行された CPU は"000"、タイムスタンプは<秒>.<マイクロ秒>で表示され、実行された時間を示します。実行された関数は"__fsnotify_parent"で、"vfs_write"から呼び出されています。

function トレーサはトレースデータをリングバッファに格納するため、古いデータは上書きされる可能性があります。バッファサイズは buffer_size_kb ファイルで調整できます。

シングルスレッドトレース

PID を set_ftrace_pid に書き込むと、該当スレッドのみをトレースすることができます。

```
# echo 3008 > set_ftrace_pid
# echo function > current_tracer
# echo 1 > tracing_on
# cat trace
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
           cron-3008  [000] 19953.026060: finish_task_switch <-__schedule
           cron-3008  [000] 19953.026069: sub_preempt_count <-__schedule
           cron-3008  [000] 19953.026073: hrtimer_cancel <-do_nanosleep
           cron-3008  [000] 19953.026075: hrtimer_try_to_cancel <-hrtimer_cancel
           cron-3008  [000] 19953.026078: lock_hrtimer_base <-
hrtimer_try_to_cancel
           cron-3008  [000] 19953.026080: _raw_spin_lock_irqsave <-
lock_hrtimer_base
           cron-3008  [000] 19953.026083: __raw_spin_lock_irqsave <-
_raw_spin_lock_irqsave
           cron-3008  [000] 19953.026085: add_preempt_count <-
__raw_spin_lock_irqsave
           cron-3008  [000] 19953.026088: _raw_spin_unlock_irqrestore <-
hrtimer_try_to_cancel
           cron-3008  [000] 19953.026090: sub_preempt_count <-
_raw_spin_unlock_irqrestore
           cron-3008  [000] 19953.026100: sys_gettimeofday <-ret_fast_syscall
```

```
(省略)
# echo 0 > tracing_on
#
```

元に戻すには空文字列を set_ftrace_pid に書き込みます。

```
# cat set_ftrace_pid
3008
# echo > set_ftrace_pid
# cat set_ftrace_pid
no pid
#
```

フィルタ

カーネルのコンフィグレーションで CONFIG_DYNAMIC_FTRACE をセットした場合、トレース対象のカーネル関数をフィルタすることができます。

指定した関数のみをトレース対象にする場合は、関数名を set_ftrace_filter に書き込みます。逆に特定の関数をトレース対象から除外したい場合は、関数名を set_ftrace_notrace に書き込みます。スペースまたは改行で区切ることで複数の関数を指定することができます。

指定可能なカーネル関数名は、available_filter_functions に列挙されます。

また、関数名の指定には簡単なワイルドカードが使用できます。使用できるワイルドカードは次のものです。

ワイルドカード	説明
<match>*	前方一致
*<match>	後方一致
<match>	部分一致

これら以外の<match>*<match>のような指定はできません。上記のパターンでは指定できない場合は、available_filter_functions の内容を grep コマンドの正規表現などでフィルタし、set_ftrace_filter あるいは set_ftrace_notrace にリダイレクトすれば良いでしょう。

フィルタを解除するには、set_ftrace_filter/set_ftrace_notrace を空にします。

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

"*lock*"をトレース対象に、"*spin*"を非対象にした実行例を次に示します。

```
# echo function > current_tracer
# echo '*lock*' > set_ftrace_filter
# echo '*spin*' > set_ftrace_notrace
# echo 1 > tracing_on
# cat trace
```

```
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
            bash-3354  [000]   652.263010: __srcu_read_lock <-fsnotify
            bash-3354  [000]   652.263019: __srcu_read_unlock <-fsnotify
            bash-3354  [000]   652.263035: locks_remove_posix <-filp_close
            bash-3354  [000]   652.263039: locks_remove_flock <-fput
            bash-3354  [000]   652.263042: files_lglock_local_lock_cpu <-
file_sb_list_del
            bash-3354  [000]   652.263045: files_lglock_local_unlock_cpu <-
file_sb_list_del
            bash-3354  [000]   652.263057: vfsmount_lock_local_lock <-
mntput_no_expire
            bash-3354  [000]   652.263060: vfsmount_lock_local_unlock <-
mntput_no_expire
            bash-3354  [000]   652.263064: __rcu_read_lock <-fget_raw
            bash-3354  [000]   652.263067: __rcu_read_unlock <-fget_raw
            bash-3354  [000]   652.263070: __rcu_read_lock <-do_fcntl
            bash-3354  [000]   652.263072: __rcu_read_unlock <-do_fcntl
            bash-3354  [000]   652.263081: locks_remove_posix <-filp_close
            bash-3354  [000]   652.263146: clocksource_mmio_readl_down <-
getnstimeofday
            bash-3354  [000]   652.263186: set_current_blocked <-sigprocmask
            bash-3354  [000]   652.263190: __set_task_blocked <-
set_current_blocked
            bash-3354  [000]   652.263202: __rcu_read_lock <-tty_ioctl
            bash-3354  [000]   652.263209: __rcu_read_unlock <-tty_ioctl
            bash-3354  [000]   652.263214: set_current_blocked <-sigprocmask
(省略)
# echo 0 > tracing_on
#
```

フィルタコマンド

フィルタには 2 つのオプションコマンドがサポートされています。

コマンドは以下のフォーマットで指定します。

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

- **mod コマンド**

関数フィルタを適用するモジュールを指定することができます。パラメータに適用するモジュールを指定します。例えば、”ext3”モジュールの”write*”関数のみをトレースしたい場合、次のように指定します。

```
# echo 'write*:mod:ext3' > set_ftrace_filter
```

コマンドを削除するには、'!'を前置します。

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

- traceon/traceoff コマンド

　指定した関数を検出した際にトレースを開始または停止することができます。パラメータでトレースを開始・停止する回数を指定します。指定しなかった場合は無制限になります。

　例えば、schedule bug を検出した時に 5 回までトレースを停止するには、次のように指定します。

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

　コマンドを削除するには'!'を前置し、パラメータは無指定とします。

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

1.5.2. function_graph トレーサ

　function_graph トレーサは、関数の入り口と出口でトレースする点を除いて function トレーサと同様です。

　関数の入り口と出口でトレースするため、関数の実行時間の計測や関数コールグラフの表示が行えます。関数コールグラフは C 言語風の書式で表示されるため、カーネルの動作を視覚的に把握しやすいです。

```
# echo function_graph > current_tracer
# echo 1 > tracing_on ; echo 0 > tracing_on
# cat trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0)   3.120 us    |      sub_preempt_count();
 0) + 15.361 us   |    }
 0) + 43.201 us   |  }
 0)               |  sys_dup2() {
 0)               |    sys_dup3() {
 0)               |      _raw_spin_lock() {
(省略)
 0)   3.000 us    |              add_preempt_count();
 0)   2.881 us    |              sub_preempt_count();
 0)               |              vfsmount_lock_local_unlock() {
 0)   3.000 us    |                sub_preempt_count();
 0)   8.760 us    |              }
 0) + 37.681 us   |            }
 0) + 43.561 us   |          }
 0) ! 214.205 us  |        }
 0) ! 231.726 us  |      }
 0) ! 275.527 us  |    }
```

```
 0) ! 285.967 us  |  }
 0)               |  sys_fcntl64() {
(省略)
#
```

“CPU”は関数が実行された CPU 番号、”DURATION”は関数の実行時間、”FUNCTION CALLS”は実行された関数名が表示されます。”CPU”と”DURATION”の間に overhead を示すマークが示されます。ここには、実行時間が 10μ 秒以上かかった場合は”+”、100μ 秒以上かかった場合は”!”が表示されます。

出力項目は trace_option によってカスタマイズすることができます。

オプション名	出力項目
funcgraph-cpu nofuncgraph-cpu	CPU の表示・非表示
funcgraph-duration nofuncgraph-duration	DURATION の表示・非表示
funcgraph-overhead nofuncgraph-overhead	overhead の表示・非表示
funcgraph-proc nofuncgraph-proc	TASK/PID の表示・非表示
funcgraph-abstime nofuncgraph-abstime	TIME の表示・非表示

オプション名が”no”で始まっているものは、該当項目を非表示にします。

TASK/PID の表示を有効にした出力例を次に示します。

```
# echo function_graph > current_tracer
# echo funcgraph-proc > trace_options
# echo 1 > tracing_on ; echo 0 > tracing_on
# cat trace
# tracer: function_graph
#
# CPU  TASK/PID         DURATION                  FUNCTION CALLS
# |     |    |           |   |                     |   |   |   |
 1)   bash-3345    | + 10.921 us   |  __fsnotify_parent();
 1)   bash-3345    |               |  fsnotify() {
 1)   bash-3345    |               |    __srcu_read_lock() {
 1)   bash-3345    |   3.360 us    |      add_preempt_count();
 1)   bash-3345    |   3.240 us    |      sub_preempt_count();
 1)   bash-3345    | + 16.680 us   |    }
 1)   bash-3345    |               |    __srcu_read_unlock() {
 1)   bash-3345    |   3.121 us    |      add_preempt_count();
(省略)
```

```
#
```

TIME の表示を有効にし、DURATION を非表示にした出力例を次に示します。

```
# echo function_graph > current_tracer
# echo funcgraph-abstime > trace_options
# echo nofuncgraph-duration > trace_options
# echo 1 > tracing_on ; echo 0 > tracing_on
# cat trace
# tracer: function_graph
#
#     TIME        CPU               FUNCTION CALLS
#      |          |                 |   |   |   |
40019.445254 |   0) __fsnotify_parent();
40019.445268 |   0) fsnotify() {
40019.445273 |   0)   __srcu_read_lock() {
40019.445276 |   0)     add_preempt_count();
40019.445283 |   0)     sub_preempt_count();
40019.445288 |   0)   }
40019.445292 |   0)   __srcu_read_unlock() {
40019.445295 |   0)     add_preempt_count();
40019.445301 |   0)     sub_preempt_count();
40019.445307 |   0)   }
40019.445309 |   0) }
40019.445323 |   0) sys_dup2() {
40019.445327 |   0)   sys_dup3() {
40019.445330 |   0)     _raw_spin_lock() {
40019.445334 |   0)       __raw_spin_lock() {
(省略)
#
```

フィルタ

function_graph トレーサでは、function トレーサと同様のフィルタの他に特別なフィルタが使用できます。

指定された関数とその関数から呼び出される子関数のネストのみをトレースすることができます。起点となる関数を set_graph_function に指定します。関数名を追記することで複数の関数を起点として指定することもできます。

```
# echo sys_open > set_graph_function
# echo sys_close >> set_graph_function
# echo function_graph > current_tracer
# echo 1 > tracing_on ; echo 0 > tracing_on
# cat trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
```

```
# |     |   |                   |   |   |   |
 0) ! 623.535 us  |    }
 0) ! 636.976 us  |  }
 0)               |  sys_close() {
 0)               |    _raw_spin_lock() {
 0)               |      __raw_spin_lock() {
 0)   3.240 us    |        add_preempt_count();
 0)   9.361 us    |      }
 0) + 15.121 us   |    }
 0)               |    _raw_spin_unlock() {
 0)   3.000 us    |      sub_preempt_count();
 0)   8.880 us    |    }
 0)               |    filp_close() {
 0)   3.000 us    |      dnotify_flush();
 0)   3.840 us    |      locks_remove_posix();
 0)   3.000 us    |      fput();
 0) + 22.441 us   |    }
 0) + 63.482 us   |  }
 0)               |  sys_open() {
 0)               |    do_sys_open() {
 0)               |      getname() {
 0)               |        getname_flags() {
 0)   3.840 us    |          kmem_cache_alloc();
 0) + 10.800 us   |        }
 0) + 16.680 us   |      }
 0)               |      alloc_fd() {
 0)               |        _raw_spin_lock() {
 0)               |          __raw_spin_lock() {
 0)   3.120 us    |            add_preempt_count();
 0)   8.760 us    |          }
```

元に戻したいときは set_graph_function を空にします。

```
# echo > set_graph_function
# cat set_graph_function
#### all functions enabled ####
#
```

1.5.3. irqsoff トレーサ

irqsoff トレーサは、割り込み禁止時間をトレースして、最大待ち時間を記録します。

最大値は tracing_max_latency に保持されます。"0"を tracing_max_latency に書き込むことでリセットできます。

このようなレイテンシの測定の場合、トレースオプションの latency-format を使用して trace ファイルのフォーマットを変更することで遅延の原因を分析しやすくなります。

```
# echo irqsoff > current_tracer
# echo latency-format > trace_options
# echo 0 > tracing_max_latency
# echo 1 > tracing_on
# ls -ltr
(省略)
# echo 0 > tracing_on
# cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.1.1
# --------------------------------------------------------------------
# latency: 1501 us, #674/674, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#    -----------------
#    | task: mmcqd/1-533 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: __irq_svc
#  => ended at:   __irq_svc
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
 mmcqd/1-533     0d...    1us+: __irq_svc
 mmcqd/1-533     0d...    3us+: asm_do_IRQ <-__irq_svc
 mmcqd/1-533     0d...    6us+: handle_IRQ <-asm_do_IRQ
 mmcqd/1-533     0d...    8us+: irq_enter <-handle_IRQ
 mmcqd/1-533     0d...   10us+: rcu_irq_enter <-irq_enter
 mmcqd/1-533     0d...   12us+: rcu_exit_nohz <-rcu_irq_enter
 mmcqd/1-533     0d...   15us+: idle_cpu <-irq_enter
(省略)
 mmcqd/1-533     0d..1 1495us+: idle_cpu <-irq_exit
 mmcqd/1-533     0d..1 1497us+: sub_preempt_count <-irq_exit
 mmcqd/1-533     0d... 1500us+: __irq_svc
 mmcqd/1-533     0d... 1505us+: trace_hardirqs_on <-__irq_svc
 mmcqd/1-533     0d... 1576us : <stack trace>
#
```

上の例では、最大の待ち時間は 1501μ秒、そのトレースデータのレコード番号は 674、記録されたレコードの総数が 674 個、タスクが実行された CPU は 0 番、タスク名は"mmcqd/1"、PID は 533、割り込みが禁止された関数(started at)と割り込みが許可された関数

(ended at)は共に"__irq_svc"、となっています。

ヘッダに続いてトレースデータが出力されます。タスク名、PID、各種状態を示すマーク、トレース開始からの相対時間、実行時間の程度を示すマーク、実行された関数名が表示されます。

状態を示すマーク部は次のように表示されます。

CPU#	タスクが実行された CPU の番号	
irqs-off	d	割り込みが禁止されている
	.	それ以外
need-resched	N	need_resched フラグがセットされている
	.	それ以外
hardirq/softirq	H	softirq 中に発生した hardirq
	h	hardirq の発生
	s	softirq の発生
	.	通常のコンテキスト
preempt-depth	プリエンプト禁止のレベル	

実行時間の程度を示すマークには、実行時間が preempt_mark_thresh(デフォルトは 100μ 秒)以上の場合に'!'、1μ 秒以上の場合に'+'が表示されます。

1.5.4. preemptoff トレーサ

preemptoff トレーサは、プリエンプト禁止時間をトレースし、irqsoff トレーサと同様に最大待ち時間を記録します。

```
# echo preemptoff > current_tracer
# echo latency-format > trace_options
# echo 0 > tracing_max_latency
# echo 1 > tracing_on
# ls -ltr
(省略)
# echo 0 > tracing_on
# cat trace
# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.1.1
# --------------------------------------------------------------------
# latency: 1494 us, #667/667, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#    -----------------
#    | task: mmcqd/1-533 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
```

```
#  => started at: irq_enter
#  => ended at:   irq_exit
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
 mmcqd/1-533     0d.h.    1us+: irq_enter
 mmcqd/1-533     0d.h.    4us+: generic_handle_irq <-handle_IRQ
 mmcqd/1-533     0d.h.    7us+: handle_fasteoi_irq <-generic_handle_irq
 mmcqd/1-533     0d.h.    9us+: _raw_spin_lock <-handle_fasteoi_irq
 mmcqd/1-533     0d.h.   11us+: __raw_spin_lock <-_raw_spin_lock
(省略)
 mmcqd/1-533     0d.h. 1469us+: add_preempt_count <-__raw_spin_lock
 mmcqd/1-533     0d.h1 1472us+: gic_eoi_irq <-handle_fasteoi_irq
 mmcqd/1-533     0d.h1 1474us+: _raw_spin_unlock <-handle_fasteoi_irq
 mmcqd/1-533     0d.h1 1477us+: sub_preempt_count <-_raw_spin_unlock
 mmcqd/1-533     0d.h. 1479us+: irq_exit <-handle_IRQ
 mmcqd/1-533     0d.h. 1481us+: sub_preempt_count <-irq_exit
 mmcqd/1-533     0d..1 1484us+: rcu_irq_exit <-irq_exit
 mmcqd/1-533     0d..1 1486us+: rcu_enter_nohz <-rcu_irq_exit
 mmcqd/1-533     0d..1 1489us+: idle_cpu <-irq_exit
 mmcqd/1-533     0d..1 1491us+: sub_preempt_count <-irq_exit
 mmcqd/1-533     0d..1 1493us+: irq_exit
 mmcqd/1-533     0d..1 1497us+: trace_preempt_on <-irq_exit
 mmcqd/1-533     0d..1 1575us : <stack trace>
#
```

1.5.5. preemptirqsoff トレーサ

preemptirqsoff トレーサは、割り込みまたはプリエンプトの禁止時間をトレースして、最大待ち時間を記録します。

```
# echo preemptirqsoff > current_tracer
# echo latency-format > trace_options
# echo 0 > tracing_max_latency
# echo 1 > tracing_on
# ls -ltr
(省略)
# echo 0 > tracing_on
# cat trace
```

```
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.1.1
# --------------------------------------------------------------------
# latency: 1507 us, #674/674, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#    -----------------
#    | task: mmcqd/1-533 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: __irq_svc
#  => ended at:   irq_exit
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
 mmcqd/1-533     0d...    1us+: __irq_svc
 mmcqd/1-533     0d...    3us+: asm_do_IRQ <-__irq_svc
 mmcqd/1-533     0d...    6us+: handle_IRQ <-asm_do_IRQ
 mmcqd/1-533     0d...    8us+: irq_enter <-handle_IRQ
 mmcqd/1-533     0d...   10us+: rcu_irq_enter <-irq_enter
 mmcqd/1-533     0d...   12us+: rcu_exit_nohz <-rcu_irq_enter
 mmcqd/1-533     0d...   15us+: idle_cpu <-irq_enter
 mmcqd/1-533     0d...   18us+: add_preempt_count <-irq_enter
 mmcqd/1-533     0d.h.   21us+: generic_handle_irq <-handle_IRQ
(省略)
 mmcqd/1-533     0d.h1 1487us+: _raw_spin_unlock <-handle_fasteoi_irq
 mmcqd/1-533     0d.h1 1489us+: sub_preempt_count <-_raw_spin_unlock
 mmcqd/1-533     0d.h. 1491us+: irq_exit <-handle_IRQ
 mmcqd/1-533     0d.h. 1494us+: sub_preempt_count <-irq_exit
 mmcqd/1-533     0d..1 1496us+: rcu_irq_exit <-irq_exit
 mmcqd/1-533     0d..1 1498us+: rcu_enter_nohz <-rcu_irq_exit
 mmcqd/1-533     0d..1 1501us+: idle_cpu <-irq_exit
 mmcqd/1-533     0d..1 1503us+: sub_preempt_count <-irq_exit
 mmcqd/1-533     0d..1 1506us+: irq_exit
 mmcqd/1-533     0d..1 1510us!: trace_preempt_on <-irq_exit
 mmcqd/1-533     0d..1 1613us : <stack trace>
#
```

1.5.6. wakeup トレーサ

リアルタイム環境において、起床されたタスクが実際に実行されるまでの時間をトレースし

て、最大待ち時間を記録します。このトレーサはRTタスクのみを対象としていることに注意が必要です。

```
# echo wakeup > current_tracer
e# echo latency-format > trace_options
# echo 0 > tracing_max_latency
# echo 1 > tracing_on
# chrt -f 5 sleep 1
# echo 0 > tracing_on
# cat trace
# tracer: wakeup
#
# wakeup latency trace v1.1.5 on 3.1.1
# --------------------------------------------------------------------
# latency: 582 us, #199/199, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#    -----------------
#    | task: sleep-12593 (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    3us+:      0:120:R   + [001] 12593: 94:R sleep
  <idle>-0       1d.h4    9us+: 0
  <idle>-0       1d.h3   12us+: check_preempt_curr <-ttwu_do_wakeup
  <idle>-0       1d.h3   15us+: resched_task <-check_preempt_curr
  <idle>-0       1dNh3   19us+: task_woken_rt <-ttwu_do_wakeup
  <idle>-0       1dNh3   23us+: _raw_spin_unlock <-try_to_wake_up
(省略)
  <idle>-0       1.N.1  555us+: add_preempt_count <-__raw_spin_lock_irqsave
  <idle>-0       1dN.2  558us+: put_prev_task_idle <-__schedule
  <idle>-0       1dN.2  561us+: pick_next_task_stop <-__schedule
  <idle>-0       1dN.2  564us+: pick_next_task_rt <-__schedule
  <idle>-0       1d..2  568us+: _raw_spin_unlock_irq <-__schedule
  <idle>-0       1...2  570us+: sub_preempt_count <-_raw_spin_unlock_irq
  <idle>-0       1...2  573us+: __schedule
  <idle>-0       1...2  576us :      0:120:R ==> [001] 12593: 94:R sleep
#
```

1.6. トレースオプション

トレース結果の出力内容をオプションによってカスタマイズすることができます。ここでは、使用できるオプションについて説明します。

オプションを有効にするには、オプション名を trace_options ファイルに書き込みます。無効にするにはオプション名の前に"no"を付けて書き込みます。

- print-parent

  トレースされた関数に呼び出し元関数の表示を追加します。

  - noprint-parent

```
           bash-3330  [001] 36042.036183: __fsnotify_parent
```

  - print-parent

```
           bash-3330  [001] 36042.036183: __fsnotify_parent <-vfs_write
```

- sym-offset

  関数名にオフセットの表示を追加します。

  - nosym-offset

```
           bash-3330  [001] 36042.036183: __fsnotify_parent
```

  - sym-offset

```
           bash-3330  [001] 36042.036183: __fsnotify_parent+0x14/0xe0
```

- sym-addr

  関数名にアドレスの表示を追加します。

  - nosym-addr

```
           bash-3330  [001] 36042.036183: __fsnotify_parent
```

  - sym-addr

```
           bash-3330  [001] 36042.036183: __fsnotify_parent <c0103b7c>
```

- verbose

  latency-format オプションが有効なときに冗長な表示を行います。

  - noverbose

```
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.1.1
# --------------------------------------------------------------------
# latency: 1255 us, #521/521, 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: __irq_svc
#  => ended at:   __irq_svc
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>-0       0d..1    1us+: __irq_svc
```

- verbose

```
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.1.1
# --------------------------------------------------------------------
# latency: 1255 us, #521/521, 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: __irq_svc
#  => ended at:   __irq_svc
#
#
          <idle>     0   0 1 00000001 00000000 [8997200d9] 0.001ms (+0.003ms):
__irq_svc
```

- **raw**

  生のトレースデータを表示します。

- noraw

```
            bash-3330  [001] 36042.036183: __fsnotify_parent
```

- raw

```
3330 1 36042036182545 c0103b7c c00cb750
```

- **hex**

  raw と同様に、16 進数表記で表示します。

- nohex

```
            bash-3330  [001] 36042.036183: __fsnotify_parent
```

- hex

```
00000d02 00000001 000020c7b0c37e11 c0103b7c c00cb750
```

- bin

  raw と同様に、バイナリフォーマットで出力します。

- block

  wakeup トレーサをブロッキングモードにします。

- stacktrace

  トレース時にコールスタックを記録するようにトレーサの動作を変更します。

- userstacktrace

  ユーザスペーススレッドのスタックトレースを記録するようにトレーサの動作を変更します。

- sym-userobj

  userstacktrace が有効な時、アドレスが属するオブジェクトを探し、オブジェクト名と相対アドレスを表示します。オブジェクトの検索は trace または trace_pipe の読み込み時に実行されます。

- latency-format

  レイテンシに関する追加情報が表示されます。

  - nolatency-format

```
# tracer: irqsoff
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
          <idle>-0     [000] 36934.123737: __irq_svc
```

  - latency-format

```
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.1.1
# --------------------------------------------------------------------
# latency: 1255 us, #521/521, 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: __irq_svc
#  => ended at:   __irq_svc
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
```

```
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>-0       0d..1    1us+: __irq_svc
```

- overwrite

  トレースバッファが溢れた時の動作を選択します。

  "1"(デフォルト)の場合、最も古いレコードは破棄され上書きされます。

  "0"の場合、新しいイベントは記録されなくなります。