The Linux Kernel Tracepoint API

Jason Baron <jbaron@redhat.com>
William Cohen <wcohen@redhat.com>

The Linux Kernel Tracepoint API

by Jason Baron and William Cohen

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

Table of Contents

1. Introduction 1
2. IRQ 2
    - trace_irq_handler_entry 3
    - trace_irq_handler_exit 4
    - trace_softirq_entry 5
    - trace_softirq_exit 6
    - trace_softirq_raise 7
3. SIGNAL 8
    - trace_signal_generate 9
    - trace_signal_deliver 10
4. Block IO 11
    - trace_block_touch_buffer 12
    - trace_block_dirty_buffer 13
    - trace_block_rq_abort 14
    - trace_block_rq_requeue 15
    - trace_block_rq_complete 16
    - trace_block_rq_insert 17
    - trace_block_rq_issue 18
    - trace_block_bio_bounce 19
    - trace_block_bio_complete 20
    - trace_block_bio_backmerge 21
    - trace_block_bio_frontmerge 22
    - trace_block_bio_queue 23
    - trace_block_getrq 24
    - trace_block_sleeprq 25
    - trace_block_plug 26
    - trace_block_unplug 27
    - trace_block_split 28
    - trace_block_bio_remap 29
    - trace_block_rq_remap 30
5. Workqueue 31
    - trace_workqueue_queue_work 32
    - trace_workqueue_activate_work 33
    - trace_workqueue_execute_start 34
    - trace_workqueue_execute_end 35

Chapter 1. Introduction

Tracepoints are static probe points that are located in strategic points throughout the kernel. 'Probes' register/unregister with tracepoints via a callback mechanism. The 'probes' are strictly typed functions that are passed a unique set of parameters defined by each tracepoint.

From this simple callback mechanism, 'probes' can be used to profile, debug, and understand kernel behavior. There are a number of tools that provide a framework for using 'probes'. These tools include Systemtap, ftrace, and LTTng.

Tracepoints are defined in a number of header files via various macros. Thus, the purpose of this document is to provide a clear accounting of the available tracepoints. The intention is to understand not only what tracepoints are available but also to understand where future tracepoints might be added.

The API presented has functions of the form: `trace_tracepointname(function parameters)`. These are the tracepoints callbacks that are found throughout the code. Registering and unregistering probes with these callback sites is covered in the `Documentation/trace/*` directory.

Chapter 2. IRQ

Name

trace_irq_handler_entry — called immediately before the irq action handler

Synopsis

```
void trace_irq_handler_entry (int irq, struct irqaction * action);
```

Arguments

irq irq number

action pointer to struct irqaction

Description

The struct irqaction pointed to by *action* contains various information about the handler, including the device name, *action*->name, and the device id, *action*->dev_id. When used in conjunction with the irq_handler_exit tracepoint, we can figure out irq handler latencies.

Name

trace_irq_handler_exit — called immediately after the irq action handler returns

Synopsis

```
void trace_irq_handler_exit (int irq, struct irqaction * action, int
ret);
```

Arguments

irq irq number

action pointer to struct irqaction

ret return value

Description

If the *ret* value is set to IRQ_HANDLED, then we know that the corresponding *action*->handler scuccessully handled this irq. Otherwise, the irq might be a shared irq line, or the irq was not handled successfully. Can be used in conjunction with the irq_handler_entry to understand irq handler latencies.

Name

trace_softirq_entry — called immediately before the softirq handler

Synopsis

```
void trace_softirq_entry (unsigned int vec_nr);
```

Arguments

vec_nr softirq vector number

Description

When used in combination with the softirq_exit tracepoint we can determine the softirq handler routine.

Name

trace_softirq_exit — called immediately after the softirq handler returns

Synopsis

```
void trace_softirq_exit (unsigned int vec_nr);
```

Arguments

vec_nr softirq vector number

Description

When used in combination with the softirq_entry tracepoint we can determine the softirq handler routine.

Name

trace_softirq_raise — called immediately when a softirq is raised

Synopsis

```
void trace_softirq_raise (unsigned int vec_nr);
```

Arguments

vec_nr softirq vector number

Description

When used in combination with the softirq_entry tracepoint we can determine the softirq raise to run latency.

Chapter 3. SIGNAL

Name

trace_signal_generate — called when a signal is generated

Synopsis

```
void trace_signal_generate (int sig, struct siginfo * info, struct
task_struct * task, int group, int result);
```

Arguments

sig	signal number
info	pointer to struct siginfo
task	pointer to struct task_struct
group	shared or private
result	TRACE_SIGNAL_*

Description

Current process sends a 'sig' signal to 'task' process with 'info' siginfo. If 'info' is SEND_SIG_NOINFO or SEND_SIG_PRIV, 'info' is not a pointer and you can't access its field. Instead, SEND_SIG_NOINFO means that si_code is SI_USER, and SEND_SIG_PRIV means that si_code is SI_KERNEL.

Name

trace_signal_deliver — called when a signal is delivered

Synopsis

```
void trace_signal_deliver (int sig, struct siginfo * info, struct
k_sigaction * ka);
```

Arguments

sig signal number

info pointer to struct siginfo

ka pointer to struct k_sigaction

Description

A 'sig' signal is delivered to current process with 'info' siginfo, and it will be handled by 'ka'. ka->sa.sa_handler can be SIG_IGN or SIG_DFL. Note that some signals reported by signal_generate tracepoint can be lost, ignored or modified (by debugger) before hitting this tracepoint. This means, this can show which signals are actually delivered, but matching generated signals and delivered signals may not be correct.

Chapter 4. Block IO

Name

trace_block_touch_buffer — mark a buffer accessed

Synopsis

```
void trace_block_touch_buffer (struct buffer_head * bh);
```

Arguments

bh buffer_head being touched

Description

Called from `touch_buffer`.

Name

trace_block_dirty_buffer — mark a buffer dirty

Synopsis

```
void trace_block_dirty_buffer (struct buffer_head * bh);
```

Arguments

bh buffer_head being dirtied

Description

Called from `mark_buffer_dirty`.

Name

trace_block_rq_abort — abort block operation request

Synopsis

```
void trace_block_rq_abort (struct request_queue * q, struct request *
rq);
```

Arguments

q queue containing the block operation request

rq block IO operation request

Description

Called immediately after pending block IO operation request *rq* in queue *q* is aborted. The fields in the operation request *rq* can be examined to determine which device and sectors the pending operation would access.

Name

trace_block_rq_requeue — place block IO request back on a queue

Synopsis

```
void trace_block_rq_requeue (struct request_queue * q, struct request
* rq);
```

Arguments

q queue holding operation

rq block IO operation request

Description

The block operation request *rq* is being placed back into queue *q*. For some reason the request was not completed and needs to be put back in the queue.

Name

trace_block_rq_complete — block IO operation completed by device driver

Synopsis

```
void trace_block_rq_complete (struct request_queue * q, struct request
* rq, unsigned int nr_bytes);
```

Arguments

q	queue containing the block operation request
rq	block operations request
nr_bytes	number of completed bytes

Description

The block_rq_complete tracepoint event indicates that some portion of operation request has been completed by the device driver. If the *rq*->bio is NULL, then there is absolutely no additional work to do for the request. If *rq*->bio is non-NULL then there is additional work required to complete the request.

Name

trace_block_rq_insert — insert block operation request into queue

Synopsis

```
void trace_block_rq_insert (struct request_queue * q, struct request
* rq);
```

Arguments

q target queue

rq block IO operation request

Description

Called immediately before block operation request *rq* is inserted into queue *q*. The fields in the operation request *rq* struct can be examined to determine which device and sectors the pending operation would access.

Name

trace_block_rq_issue — issue pending block IO request operation to device driver

Synopsis

```
void trace_block_rq_issue (struct request_queue * q, struct request *
rq);
```

Arguments

q queue holding operation

rq block IO operation operation request

Description

Called when block operation request *rq* from queue *q* is sent to a device driver for processing.

Name

trace_block_bio_bounce — used bounce buffer when processing block operation

Synopsis

```
void trace_block_bio_bounce (struct request_queue * q, struct bio *
bio);
```

Arguments

q queue holding the block operation

bio block operation

Description

A bounce buffer was used to handle the block operation *bio* in *q*. This occurs when hardware limitations prevent a direct transfer of data between the *bio* data memory area and the IO device. Use of a bounce buffer requires extra copying of data and decreases performance.

Name

trace_block_bio_complete — completed all work on the block operation

Synopsis

```
void trace_block_bio_complete (struct request_queue * q, struct bio *
bio, int error);
```

Arguments

q	queue holding the block operation
bio	block operation completed
error	io error value

Description

This tracepoint indicates there is no further work to do on this block IO operation *bio*.

Name

trace_block_bio_backmerge — merging block operation to the end of an existing operation

Synopsis

```
void trace_block_bio_backmerge (struct request_queue * q, struct request
* rq, struct bio * bio);
```

Arguments

q queue holding operation

rq request bio is being merged into

bio new block operation to merge

Description

Merging block request *bio* to the end of an existing block request in queue *q*.

Name

trace_block_bio_frontmerge — merging block operation to the beginning of an existing operation

Synopsis

```
void trace_block_bio_frontmerge (struct request_queue * q, struct re-
quest * rq, struct bio * bio);
```

Arguments

q queue holding operation

rq request bio is being merged into

bio new block operation to merge

Description

Merging block IO operation *bio* to the beginning of an existing block operation in queue *q*.

Name

trace_block_bio_queue — putting new block IO operation in queue

Synopsis

```
void trace_block_bio_queue (struct request_queue * q, struct bio * bio);
```

Arguments

q queue holding operation

bio new block operation

Description

About to place the block IO operation *bio* into queue *q*.

Name

trace_block_getrq — get a free request entry in queue for block IO operations

Synopsis

```
void trace_block_getrq (struct request_queue * q, struct bio * bio,
int rw);
```

Arguments

q queue for operations

bio pending block IO operation

rw low bit indicates a read (`0`) or a write (`1`)

Description

A request struct for queue *q* has been allocated to handle the block IO operation *bio*.

Name

trace_block_sleeprq — waiting to get a free request entry in queue for block IO operation

Synopsis

```
void trace_block_sleeprq (struct request_queue * q, struct bio * bio,
int rw);
```

Arguments

q queue for operation

bio pending block IO operation

rw low bit indicates a read (`0`) or a write (`1`)

Description

In the case where a request struct cannot be provided for queue *q* the process needs to wait for an request struct to become available. This tracepoint event is generated each time the process goes to sleep waiting for request struct become available.

Name

trace_block_plug — keep operations requests in request queue

Synopsis

```
void trace_block_plug (struct request_queue * q);
```

Arguments

q request queue to plug

Description

Plug the request queue *q*. Do not allow block operation requests to be sent to the device driver. Instead, accumulate requests in the queue to improve throughput performance of the block device.

Name

trace_block_unplug — release of operations requests in request queue

Synopsis

```
void trace_block_unplug (struct request_queue * q, unsigned int depth,
bool explicit);
```

Arguments

q	request queue to unplug
depth	number of requests just added to the queue
explicit	whether this was an explicit unplug, or one from `schedule`

Description

Unplug request queue *q* because device driver is scheduled to work on elements in the request queue.

Name

trace_block_split — split a single bio struct into two bio structs

Synopsis

```
void trace_block_split (struct request_queue * q, struct bio * bio,
unsigned int new_sector);
```

Arguments

q	queue containing the bio
bio	block operation being split
new_sector	The starting sector for the new bio

Description

The bio request *bio* in request queue *q* needs to be split into two bio requests. The newly created *bio* request starts at *new_sector*. This split may be required due to hardware limitation such as operation crossing device boundaries in a RAID system.

Name

trace_block_bio_remap — map request for a logical device to the raw device

Synopsis

```
void trace_block_bio_remap (struct request_queue * q, struct bio * bio,
dev_t dev, sector_t from);
```

Arguments

q queue holding the operation

bio revised operation

dev device for the operation

from original sector for the operation

Description

An operation for a logical device has been mapped to the raw block device.

Name

trace_block_rq_remap — map request for a block operation request

Synopsis

```
void trace_block_rq_remap (struct request_queue * q, struct request *
rq, dev_t dev, sector_t from);
```

Arguments

q queue holding the operation

rq block IO operation request

dev device for the operation

from original sector for the operation

Description

The block operation request *rq* in *q* has been remapped. The block operation request *rq* holds the current information and *from* hold the original sector.

Chapter 5. Workqueue

Name

trace_workqueue_queue_work — called when a work gets queued

Synopsis

```
void trace_workqueue_queue_work (unsigned int req_cpu, struct
pool_workqueue * pwq, struct work_struct * work);
```

Arguments

req_cpu	the requested cpu
pwq	pointer to struct pool_workqueue
work	pointer to struct work_struct

Description

This event occurs when a work is queued immediately or once a delayed work is actually queued on a workqueue (ie: once the delay has been reached).

Name

trace_workqueue_activate_work — called when a work gets activated

Synopsis

```
void trace_workqueue_activate_work (struct work_struct * work);
```

Arguments

work pointer to struct work_struct

Description

This event occurs when a queued work is put on the active queue, which happens immediately after queueing unless *max_active* limit is reached.

Name

trace_workqueue_execute_start — called immediately before the workqueue callback

Synopsis

```
void trace_workqueue_execute_start (struct work_struct * work);
```

Arguments

work pointer to struct work_struct

Description

Allows to track workqueue execution.

Name

trace_workqueue_execute_end — called immediately after the workqueue callback

Synopsis

```
void trace_workqueue_execute_end (struct work_struct * work);
```

Arguments

work pointer to struct work_struct

Description

Allows to track workqueue execution.