

The Linux Kernel Tracepoint API

Jason Baron

`jbaron@redhat.com`

The Linux Kernel Tracepoint API

by Jason Baron

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.....	3
trace_irq_handler_entry	3
trace_irq_handler_exit	4
trace_softirq_entry	4
trace_softirq_exit	5
3. SIGNAL	7
trace_signal_generate.....	7
trace_signal_deliver	8
trace_signal_overflow_fail.....	9
trace_signal_lose_info	10

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

trace_irq_handler_entry

LINUX

Kernel Hackers Manual March 2012

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.

trace_irq_handler_exit

LINUX

Kernel Hackers Manual March 2012

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* successfully 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.

trace_softirq_entry

LINUX

Kernel Hackers Manual March 2012

Name

`trace_softirq_entry` — called immediately before the softirq handler

Synopsis

```
void trace_softirq_entry (struct softirq_action * h, struct  
softirq_action * vec);
```

Arguments

h

pointer to struct `softirq_action`

vec

pointer to first struct `softirq_action` in `softirq_vec` array

Description

The *h* parameter, contains a pointer to the struct `softirq_action` which has a pointer to the action handler that is called. By subtracting the *vec* pointer from the *h* pointer, we can determine the softirq number. Also, when used in combination with the `softirq_exit` tracepoint we can determine the softirq latency.

trace_softirq_exit

LINUX

Kernel Hackers Manual March 2012

Name

`trace_softirq_exit` — called immediately after the `softirq` handler returns

Synopsis

```
void trace_softirq_exit (struct softirq_action * h, struct  
softirq_action * vec);
```

Arguments

h

pointer to struct `softirq_action`

vec

pointer to first struct `softirq_action` in `softirq_vec` array

Description

The *h* parameter contains a pointer to the struct `softirq_action` that has handled the `softirq`. By subtracting the *vec* pointer from the *h* pointer, we can determine the `softirq` number. Also, when used in combination with the `softirq_entry` tracepoint we can determine the `softirq` latency.

Chapter 3. SIGNAL

trace_signal_generate

LINUX

Kernel Hackers Manual March 2012

Name

`trace_signal_generate` — called when a signal is generated

Synopsis

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

Arguments

sig

signal number

info

pointer to struct siginfo

task

pointer to struct task_struct

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.

trace_signal_deliver

LINUX

Kernel Hackers Manual March 2012

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.

trace_signal_overflow_fail

LINUX

Kernel Hackers Manual March 2012

Name

`trace_signal_overflow_fail` — called when signal queue is overflow

Synopsis

```
void trace_signal_overflow_fail (int sig, int group, struct  
siginfo * info);
```

Arguments

sig

signal number

group

signal to process group or not (bool)

info

pointer to struct siginfo

Description

Kernel fails to generate 'sig' signal with 'info' siginfo, because siginfo queue is overflow, and the signal is dropped. 'group' is not 0 if the signal will be sent to a process group. 'sig' is always one of RT signals.

trace_signal_lose_info

LINUX

Kernel Hackers Manual March 2012

Name

`trace_signal_lose_info` — called when siginfo is lost

Synopsis

```
void trace_signal_lose_info (int sig, int group, struct  
siginfo * info);
```

Arguments

sig

signal number

group

signal to process group or not (bool)

info

pointer to struct siginfo

Description

Kernel generates 'sig' signal but loses 'info' siginfo, because siginfo queue is overflow. 'group' is not 0 if the signal will be sent to a process group. 'sig' is always one of non-RT signals.

