events.txt
Event Tracing

Documentation written by Theodore Ts'o
Updated by Li Zefan and Tom Zanussi

1. Introduction

Tracepoints (see Documentation/trace/tracepoints.txt) can be used
without creating custom kernel modules to register probe functions
using the event tracing infrastructure.

Not all tracepoints can be traced using the event tracing system;
the kernel developer must provide code snippets which define how the
tracing information is saved into the tracing buffer, and how the
tracing information should be printed.

2. Using Event Tracing

2.1 Via the 'set_event' interface

The events which are available for tracing can be found in the file
/sys/kernel/debug/tracing/available_events.

To enable a particular event, such as 'sched_wakeup', simply echo it
to /sys/kernel/debug/tracing/set_event. For example:

```
# echo sched_wakeup >> /sys/kernel/debug/tracing/set_event
```

[Note: '>>' is necessary, otherwise it will firstly disable
  all the events.]

To disable an event, echo the event name to the set_event file prefixed
with an exclamation point:

```
# echo '!sched_wakeup' >> /sys/kernel/debug/tracing/set_event
```

To disable all events, echo an empty line to the set_event file:

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

To enable all events, echo '*:*' or '*:' to the set_event file:

```
# echo *:* > /sys/kernel/debug/tracing/set_event
```

The events are organized into subsystems, such as ext4, irq, sched,
etc., and a full event name looks like this: <subsystem>:<event>. The
subsystem name is optional, but it is displayed in the available_events
file. All of the events in a subsystem can be specified via the syntax
"<subsystem>:*"; for example, to enable all irq events, you can use the
command:

```
# echo 'irq:*' > /sys/kernel/debug/tracing/set_event
```

2.2 Via the 'enable' toggle

The events available are also listed in /sys/kernel/debug/tracing/events/
hierarchy
of directories.

To enable event 'sched_wakeup':

```
# echo 1 > /sys/kernel/debug/tracing/events/sched/sched_wakeup/enable
```

To disable it:

```
# echo 0 > /sys/kernel/debug/tracing/events/sched/sched_wakeup/enable
```

To enable all events in sched subsystem:

```
# echo 1 > /sys/kernel/debug/tracing/events/sched/enable
```

To enable all events:

```
# echo 1 > /sys/kernel/debug/tracing/events/enable
```

When reading one of these enable files, there are four results:

```
 0 - all events this file affects are disabled
 1 - all events this file affects are enabled
 X - there is a mixture of events enabled and disabled
 ? - this file does not affect any event
```

2.3 Boot option

In order to facilitate early boot debugging, use boot option:

```
trace_event=[event-list]
```

event-list is a comma separated list of events. See section 2.1 for event
format.

3. Defining an event-enabled tracepoint
=======================================

See The example provided in samples/trace_events

4. Event formats
================

Each trace event has a 'format' file associated with it that contains
a description of each field in a logged event. This information can
be used to parse the binary trace stream, and is also the place to
find the field names that can be used in event filters (see section 5).

It also displays the format string that will be used to print the
event in text mode, along with the event name and ID used for
profiling.

Every event has a set of 'common' fields associated with it; these are
the fields prefixed with 'common_'. The other fields vary between
events and correspond to the fields defined in the TRACE_EVENT
definition for that event.

Each field in the format has the form:

     field:field-type field-name; offset:N; size:N;

where offset is the offset of the field in the trace record and size
is the size of the data item, in bytes.

For example, here's the information displayed for the 'sched_wakeup'
event:

```
# cat /debug/tracing/events/sched/sched_wakeup/format

name: sched_wakeup
ID: 60
format:
        field:unsigned short common_type;       offset:0;       size:2;
        field:unsigned char common_flags;       offset:2;       size:1;
        field:unsigned char common_preempt_count;       offset:3;       size:1;
        field:int common_pid;   offset:4;       size:4;
        field:int common_tgid;  offset:8;       size:4;

        field:char comm[TASK_COMM_LEN]; offset:12;      size:16;
        field:pid_t pid;        offset:28;      size:4;
        field:int prio; offset:32;      size:4;
        field:int success;      offset:36;      size:4;
        field:int cpu;  offset:40;      size:4;

print fmt: "task %s:%d [%d] success=%d [%03d]", REC->comm, REC->pid,
           REC->prio, REC->success, REC->cpu
```

This event contains 10 fields, the first 5 common and the remaining 5
event-specific. All the fields for this event are numeric, except for
'comm' which is a string, a distinction important for event filtering.

5. Event filtering

Trace events can be filtered in the kernel by associating boolean
'filter expressions' with them. As soon as an event is logged into
the trace buffer, its fields are checked against the filter expression
associated with that event type. An event with field values that
'match' the filter will appear in the trace output, and an event whose
values don't match will be discarded. An event with no filter
associated with it matches everything, and is the default when no
filter has been set for an event.

5.1 Expression syntax

A filter expression consists of one or more 'predicates' that can be

combined using the logical operators '&&' and '||'. A predicate is simply a clause that compares the value of a field contained within a logged event with a constant value and returns either 0 or 1 depending on whether the field value matched (1) or didn't match (0):

field-name relational-operator value

Parentheses can be used to provide arbitrary logical groupings and double-quotes can be used to prevent the shell from interpreting operators as shell metacharacters.

The field-names available for use in filters can be found in the 'format' files for trace events (see section 4).

The relational-operators depend on the type of the field being tested:

The operators available for numeric fields are:

==, !=, <, <=, >, >=

And for string fields they are:

==, !=

Currently, only exact string matches are supported.

Currently, the maximum number of predicates in a filter is 16.

5.2 Setting filters

A filter for an individual event is set by writing a filter expression to the 'filter' file for the given event.

For example:

```
# cd /debug/tracing/events/sched/sched_wakeup
# echo "common_preempt_count > 4" > filter
```

A slightly more involved example:

```
# cd /debug/tracing/events/sched/sched_signal_send
# echo "((sig >= 10 && sig < 15) || sig == 17) && comm != bash" > filter
```

If there is an error in the expression, you'll get an 'Invalid argument' error when setting it, and the erroneous string along with an error message can be seen by looking at the filter e.g.:

```
# cd /debug/tracing/events/sched/sched_signal_send
# echo "((sig >= 10 && sig < 15) || dsig == 17) && comm != bash" > filter
-bash: echo: write error: Invalid argument
# cat filter
((sig >= 10 && sig < 15) || dsig == 17) && comm != bash
^
parse_error: Field not found
```

Currently the caret ('^') for an error always appears at the beginning of
the filter string; the error message should still be useful though
even without more accurate position info.

5.3 Clearing filters

To clear the filter for an event, write a '0' to the event's filter
file.

To clear the filters for all events in a subsystem, write a '0' to the
subsystem's filter file.

5.3 Subsystem filters

For convenience, filters for every event in a subsystem can be set or
cleared as a group by writing a filter expression into the filter file
at the root of the subsystem. Note however, that if a filter for any
event within the subsystem lacks a field specified in the subsystem
filter, or if the filter can't be applied for any other reason, the
filter for that event will retain its previous setting. This can
result in an unintended mixture of filters which could lead to
confusing (to the user who might think different filters are in
effect) trace output. Only filters that reference just the common
fields can be guaranteed to propagate successfully to all events.

Here are a few subsystem filter examples that also illustrate the
above points:

Clear the filters on all events in the sched subsystem:

```
# cd /sys/kernel/debug/tracing/events/sched
# echo 0 > filter
# cat sched_switch/filter
none
# cat sched_wakeup/filter
none
```

Set a filter using only common fields for all events in the sched
subsystem (all events end up with the same filter):

```
# cd /sys/kernel/debug/tracing/events/sched
# echo common_pid == 0 > filter
# cat sched_switch/filter
common_pid == 0
# cat sched_wakeup/filter
common_pid == 0
```

Attempt to set a filter using a non-common field for all events in the
sched subsystem (all events but those that have a prev_pid field retain
their old filters):

```
# cd /sys/kernel/debug/tracing/events/sched
# echo prev_pid == 0 > filter
# cat sched_switch/filter
```

```
prev_pid == 0
# cat sched_wakeup/filter
common_pid == 0
```