Event Tracing

Author: Theodore Ts'o
Updated: Li Zefan and Tom Zanussi

1. Introduction

Tracepoints (see Documentation/trace/tracepoints.rst) 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 /sys/kernel/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:

==, !=, ~

The glob (~) accepts a wild card character (*,?) and character classes ([). For example:

```
prev_comm ~ "*sh"
prev_comm ~ "sh*"
prev_comm ~ "*sh*"
prev_comm ~ "ba*sh"
```

If the field is a pointer that points into user space (for example "filename" from sys_enter_openat), then you have to append ".ustring" to the field name:

```
filename.ustring ~ "password"
```

As the kernel will have to know how to retrieve the memory that the pointer is at from user space.

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

A slightly more involved example:

```
# cd /sys/kernel/debug/tracing/events/signal/signal_generate
# 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 /sys/kernel/debug/tracing/events/signal/signal_generate
# 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.

5.2.1 Filter limitations

If a filter is placed on a string pointer `(char *)` that does not point to a string on the ring buffer, but instead points to kernel or user space memory, then, for safety reasons, at most 1024 bytes of the content is copied onto a temporary buffer to do the compare. If the copy of the memory faults (the pointer points to memory that should not be accessed), then the string compare will be treated as not matching.

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
```

5.4 PID filtering

The set_event_pid file in the same directory as the top events directory exists, will filter all events from tracing any task that does not have the PID listed in the set_event_pid file.

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

Will only trace events for the current task.

To add more PIDs without losing the PIDs already included, use '>>'.

```
# echo 123 244 1 >> set_event_pid
```

6. Event triggers

Trace events can be made to conditionally invoke trigger 'commands' which can take various forms and are described in detail below; examples would be enabling or disabling other trace events or invoking a stack trace whenever the trace event is hit. Whenever a trace event with attached triggers is invoked, the set of trigger commands associated with that event is invoked. Any given trigger can additionally have an event filter of the same form as described in section 5 (Event filtering) associated with it - the command will only be invoked if the event being invoked passes the associated filter. If no filter is associated with the trigger, it always passes.

Triggers are added to and removed from a particular event by writing trigger expressions to the 'trigger' file for the given event.

A given event can have any number of triggers associated with it, subject to any restrictions that individual commands may have in that regard.

Event triggers are implemented on top of "soft" mode, which means that whenever a trace event has one or more triggers associated with it, the event is activated even if it isn't actually enabled, but is disabled in a "soft" mode. That is, the tracepoint will be called, but just will not be traced, unless of course it's actually enabled. This scheme allows triggers to be invoked even for events that aren't enabled, and also allows the current event filter implementation to be used for conditionally invoking triggers.

The syntax for event triggers is roughly based on the syntax for set_ftrace_filter 'ftrace filter commands' (see the 'Filter commands' section of Documentation/trace/ftrace.rst), but there are major differences and the implementation isn't currently tied to it in any way, so beware about making generalizations between the two.

> **Note**
>
> Writing into trace_marker (See Documentation/trace/ftrace.rst) can also enable triggers that are written into /sys/kernel/tracing/events/ftrace/print/trigger

6.1 Expression syntax

Triggers are added by echoing the command to the 'trigger' file:

```
# echo 'command[:count] [if filter]' > trigger
```

Triggers are removed by echoing the same command but starting with '!' to the 'trigger' file:

```
# echo '!command[:count] [if filter]' > trigger
```

The [if filter] part isn't used in matching commands when removing, so leaving that off in a '!' command will accomplish the same thing as having it in.

The filter syntax is the same as that described in the 'Event filtering' section above.

For ease of use, writing to the trigger file using '>' currently just adds or removes a single trigger and there's no explicit '>>' support ('>' actually behaves like '>>') or truncation support to remove all triggers (you have to use '!' for each one added.)

6.2 Supported trigger commands

The following commands are supported:

- enable_event/disable_event

  These commands can enable or disable another trace event whenever the triggering event is hit. When these commands are registered, the other trace event is activated, but disabled in a "soft" mode. That is, the tracepoint will be called, but just will not be traced. The event tracepoint stays in this mode as long as there's a trigger in effect that can trigger it.

  For example, the following trigger causes kmalloc events to be traced when a read system call is entered, and the :1 at the end specifies that this enablement happens only once:

  ```
  # echo 'enable_event:kmem:kmalloc:1' > \
      /sys/kernel/debug/tracing/events/syscalls/sys_enter_read/trigger
  ```

  The following trigger causes kmalloc events to stop being traced when a read system call exits. This disablement happens on every read system call exit:

  ```
  # echo 'disable_event:kmem:kmalloc' > \
      /sys/kernel/debug/tracing/events/syscalls/sys_exit_read/trigger
  ```

  The format is:

  ```
  enable_event:<system>:<event>[:count]
  disable_event:<system>:<event>[:count]
  ```

  To remove the above commands:

  ```
  # echo '!enable_event:kmem:kmalloc:1' > \
      /sys/kernel/debug/tracing/events/syscalls/sys_enter_read/trigger

  # echo '!disable_event:kmem:kmalloc' > \
      /sys/kernel/debug/tracing/events/syscalls/sys_exit_read/trigger
  ```

  Note that there can be any number of enable/disable_event triggers per triggering event, but there can only be one trigger per triggered event. e.g. sys_enter_read can have triggers enabling both kmem:kmalloc and sched:sched_switch, but can't have two kmem:kmalloc versions such as kmem:kmalloc and kmem:kmalloc:1 or 'kmem:kmalloc if bytes_req == 256' and 'kmem:kmalloc if bytes_alloc == 256' (they could be combined into a single filter on kmem:kmalloc though).

- stacktrace

  This command dumps a stacktrace in the trace buffer whenever the triggering event occurs.

  For example, the following trigger dumps a stacktrace every time the kmalloc tracepoint is hit:

  ```
  # echo 'stacktrace' > \
        /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger
  ```

  The following trigger dumps a stacktrace the first 5 times a kmalloc request happens with a size >= 64K:

  ```
  # echo 'stacktrace:5 if bytes_req >= 65536' > \
        /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger
  ```

  The format is:

  ```
  stacktrace[:count]
  ```

  To remove the above commands:

  ```
  # echo '!stacktrace' > \
        /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger

  # echo '!stacktrace:5 if bytes_req >= 65536' > \
        /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger
  ```

  The latter can also be removed more simply by the following (without the filter):

  ```
  # echo '!stacktrace:5' > \
        /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger
  ```

  Note that there can be only one stacktrace trigger per triggering event.

- snapshot

  This command causes a snapshot to be triggered whenever the triggering event occurs.

  The following command creates a snapshot every time a block request queue is unplugged with a depth > 1. If you were tracing a set of events or functions at the time, the snapshot trace buffer would capture those events when the trigger event occurred:

  ```
  # echo 'snapshot if nr_rq > 1' > \
  ```

```
      /sys/kernel/debug/tracing/events/block/block_unplug/trigger
```

To only snapshot once:

```
# echo 'snapshot:1 if nr_rq > 1' > \
      /sys/kernel/debug/tracing/events/block/block_unplug/trigger
```

To remove the above commands:

```
# echo '!snapshot if nr_rq > 1' > \
      /sys/kernel/debug/tracing/events/block/block_unplug/trigger

# echo '!snapshot:1 if nr_rq > 1' > \
      /sys/kernel/debug/tracing/events/block/block_unplug/trigger
```

Note that there can be only one snapshot trigger per triggering event.

- traceon/traceoff

  These commands turn tracing on and off when the specified events are hit. The parameter determines how many times the tracing system is turned on and off. If unspecified, there is no limit.

  The following command turns tracing off the first time a block request queue is unplugged with a depth > 1. If you were tracing a set of events or functions at the time, you could then examine the trace buffer to see the sequence of events that led up to the trigger event:

```
# echo 'traceoff:1 if nr_rq > 1' > \
      /sys/kernel/debug/tracing/events/block/block_unplug/trigger
```

  To always disable tracing when nr_rq > 1:

```
# echo 'traceoff if nr_rq > 1' > \
      /sys/kernel/debug/tracing/events/block/block_unplug/trigger
```

  To remove the above commands:

```
# echo '!traceoff:1 if nr_rq > 1' > \
      /sys/kernel/debug/tracing/events/block/block_unplug/trigger

# echo '!traceoff if nr_rq > 1' > \
      /sys/kernel/debug/tracing/events/block/block_unplug/trigger
```

  Note that there can be only one traceon or traceoff trigger per triggering event.

- hist

  This command aggregates event hits into a hash table keyed on one or more trace event format fields (or stacktrace) and a set of running totals derived from one or more trace event format fields and/or event counts (hitcount).

  See Documentation/trace/histogram.rst for details and examples.

7. In-kernel trace event API

In most cases, the command-line interface to trace events is more than sufficient. Sometimes, however, applications might find the need for more complex relationships than can be expressed through a simple series of linked command-line expressions, or putting together sets of commands may be simply too cumbersome. An example might be an application that needs to 'listen' to the trace stream in order to maintain an in-kernel state machine detecting, for instance, when an illegal kernel state occurs in the scheduler.

The trace event subsystem provides an in-kernel API allowing modules or other kernel code to generate user-defined 'synthetic' events at will, which can be used to either augment the existing trace stream and/or signal that a particular important state has occurred.

A similar in-kernel API is also available for creating kprobe and kretprobe events.

Both the synthetic event and k/ret/probe event APIs are built on top of a lower-level "dynevent_cmd" event command API, which is also available for more specialized applications, or as the basis of other higher-level trace event APIs.

The API provided for these purposes is describe below and allows the following:

- dynamically creating synthetic event definitions
- dynamically creating kprobe and kretprobe event definitions
- tracing synthetic events from in-kernel code
- the low-level "dynevent_cmd" API

7.1 Dyamically creating synthetic event definitions

There are a couple ways to create a new synthetic event from a kernel module or other kernel code.

The first creates the event in one step, using synth_event_create(). In this method, the name of the event to create and an array

defining the fields is supplied to synth_event_create(). If successful, a synthetic event with that name and fields will exist following that call. For example, to create a new "schedtest" synthetic event:

```
ret = synth_event_create("schedtest", sched_fields,
                         ARRAY_SIZE(sched_fields), THIS_MODULE);
```

The sched_fields param in this example points to an array of struct synth_field_desc, each of which describes an event field by type and name:

```
static struct synth_field_desc sched_fields[] = {
      { .type = "pid_t",              .name = "next_pid_field" },
      { .type = "char[16]",           .name = "next_comm_field" },
      { .type = "u64",                .name = "ts_ns" },
      { .type = "u64",                .name = "ts_ms" },
      { .type = "unsigned int",       .name = "cpu" },
      { .type = "char[64]",           .name = "my_string_field" },
      { .type = "int",                .name = "my_int_field" },
};
```

See synth_field_size() for available types.

If field_name contains [n], the field is considered to be a static array.

If field_names contains[] (no subscript), the field is considered to be a dynamic array, which will only take as much space in the event as is required to hold the array.

Because space for an event is reserved before assigning field values to the event, using dynamic arrays implies that the piecewise in-kernel API described below can't be used with dynamic arrays. The other non-piecewise in-kernel APIs can, however, be used with dynamic arrays.

If the event is created from within a module, a pointer to the module must be passed to synth_event_create(). This will ensure that the trace buffer won't contain unreadable events when the module is removed.

At this point, the event object is ready to be used for generating new events.

In the second method, the event is created in several steps. This allows events to be created dynamically and without the need to create and populate an array of fields beforehand.

To use this method, an empty or partially empty synthetic event should first be created using synth_event_gen_cmd_start() or synth_event_gen_cmd_array_start(). For synth_event_gen_cmd_start(), the name of the event along with one or more pairs of args each pair representing a 'type field_name;' field specification should be supplied. For synth_event_gen_cmd_array_start(), the name of the event along with an array of struct synth_field_desc should be supplied. Before calling synth_event_gen_cmd_start() or synth_event_gen_cmd_array_start(), the user should create and initialize a dynevent_cmd object using synth_event_cmd_init().

For example, to create a new "schedtest" synthetic event with two fields:

```
struct dynevent_cmd cmd;
char *buf;

/* Create a buffer to hold the generated command */
buf = kzalloc(MAX_DYNEVENT_CMD_LEN, GFP_KERNEL);

/* Before generating the command, initialize the cmd object */
synth_event_cmd_init(&cmd, buf, MAX_DYNEVENT_CMD_LEN);

ret = synth_event_gen_cmd_start(&cmd, "schedtest", THIS_MODULE,
                                "pid_t", "next_pid_field",
                                "u64", "ts_ns");
```

Alternatively, using an array of struct synth_field_desc fields containing the same information:

```
ret = synth_event_gen_cmd_array_start(&cmd, "schedtest", THIS_MODULE,
                                      fields, n_fields);
```

Once the synthetic event object has been created, it can then be populated with more fields. Fields are added one by one using synth_event_add_field(), supplying the dynevent_cmd object, a field type, and a field name. For example, to add a new int field named "intfield", the following call should be made:

```
ret = synth_event_add_field(&cmd, "int", "intfield");
```

See synth_field_size() for available types. If field_name contains [n] the field is considered to be an array.

A group of fields can also be added all at once using an array of synth_field_desc with add_synth_fields(). For example, this would add just the first four sched_fields:

```
ret = synth_event_add_fields(&cmd, sched_fields, 4);
```

If you already have a string of the form 'type field_name', synth_event_add_field_str() can be used to add it as-is; it will also automatically append a ';' to the string.

Once all the fields have been added, the event should be finalized and registered by calling the synth_event_gen_cmd_end() function:

```
ret = synth_event_gen_cmd_end(&cmd);
```

At this point, the event object is ready to be used for tracing new events.

7.2 Tracing synthetic events from in-kernel code

To trace a synthetic event, there are several options. The first option is to trace the event in one call, using synth_event_trace() with a variable number of values, or synth_event_trace_array() with an array of values to be set. A second option can be used to avoid the need for a pre-formed array of values or list of arguments, via synth_event_trace_start() and synth_event_trace_end() along with synth_event_add_next_val() or synth_event_add_val() to add the values piecewise.

7.2.1 Tracing a synthetic event all at once

To trace a synthetic event all at once, the synth_event_trace() or synth_event_trace_array() functions can be used.

The synth_event_trace() function is passed the trace_event_file representing the synthetic event (which can be retrieved using trace_get_event_file() using the synthetic event name, "synthetic" as the system name, and the trace instance name (NULL if using the global trace array)), along with an variable number of u64 args, one for each synthetic event field, and the number of values being passed.

So, to trace an event corresponding to the synthetic event definition above, code like the following could be used:

```
ret = synth_event_trace(create_synth_test, 7, /* number of values */
                        444,            /* next_pid_field */
                        (u64)"clackers", /* next_comm_field */
                        1000000,        /* ts_ns */
                        1000,           /* ts_ms */
                        smp_processor_id(),/* cpu */
                        (u64)"Thneed",   /* my_string_field */
                        999);           /* my_int_field */
```

All vals should be cast to u64, and string vals are just pointers to strings, cast to u64. Strings will be copied into space reserved in the event for the string, using these pointers.

Alternatively, the synth_event_trace_array() function can be used to accomplish the same thing. It is passed the trace_event_file representing the synthetic event (which can be retrieved using trace_get_event_file() using the synthetic event name, "synthetic" as the system name, and the trace instance name (NULL if using the global trace array)), along with an array of u64, one for each synthetic event field.

To trace an event corresponding to the synthetic event definition above, code like the following could be used:

```
u64 vals[7];

vals[0] = 777;                  /* next_pid_field */
vals[1] = (u64)"tiddlywinks";   /* next_comm_field */
vals[2] = 1000000;              /* ts_ns */
vals[3] = 1000;                 /* ts_ms */
vals[4] = smp_processor_id();   /* cpu */
vals[5] = (u64)"thneed";        /* my_string_field */
vals[6] = 398;                  /* my_int_field */
```

The 'vals' array is just an array of u64, the number of which must match the number of field in the synthetic event, and which must be in the same order as the synthetic event fields.

All vals should be cast to u64, and string vals are just pointers to strings, cast to u64. Strings will be copied into space reserved in the event for the string, using these pointers.

In order to trace a synthetic event, a pointer to the trace event file is needed. The trace_get_event_file() function can be used to get it - it will find the file in the given trace instance (in this case NULL since the top trace array is being used) while at the same time preventing the instance containing it from going away:

```
schedtest_event_file = trace_get_event_file(NULL, "synthetic",
                                            "schedtest");
```

Before tracing the event, it should be enabled in some way, otherwise the synthetic event won't actually show up in the trace buffer.

To enable a synthetic event from the kernel, trace_array_set_clr_event() can be used (which is not specific to synthetic events, so does need the "synthetic" system name to be specified explicitly).

To enable the event, pass 'true' to it:

```
trace_array_set_clr_event(schedtest_event_file->tr,
                          "synthetic", "schedtest", true);
```

To disable it pass false:

```
trace_array_set_clr_event(schedtest_event_file->tr,
                          "synthetic", "schedtest", false);
```

Finally, synth_event_trace_array() can be used to actually trace the event, which should be visible in the trace buffer afterwards:

```
ret = synth_event_trace_array(schedtest_event_file, vals,
                              ARRAY_SIZE(vals));
```

To remove the synthetic event, the event should be disabled, and the trace instance should be 'put' back using trace_put_event_file():

```
trace_array_set_clr_event(schedtest_event_file->tr,
                          "synthetic", "schedtest", false);
trace_put_event_file(schedtest_event_file);
```

If those have been successful, synth_event_delete() can be called to remove the event:

```
ret = synth_event_delete("schedtest");
```

7.2.2 Tracing a synthetic event piecewise

To trace a synthetic using the piecewise method described above, the synth_event_trace_start() function is used to 'open' the synthetic event trace:

```
struct synth_event_trace_state trace_state;

ret = synth_event_trace_start(schedtest_event_file, &trace_state);
```

It's passed the trace_event_file representing the synthetic event using the same methods as described above, along with a pointer to a struct synth_event_trace_state object, which will be zeroed before use and used to maintain state between this and following calls.

Once the event has been opened, which means space for it has been reserved in the trace buffer, the individual fields can be set. There are two ways to do that, either one after another for each field in the event, which requires no lookups, or by name, which does. The tradeoff is flexibility in doing the assignments vs the cost of a lookup per field.

To assign the values one after the other without lookups, synth_event_add_next_val() should be used. Each call is passed the same synth_event_trace_state object used in the synth_event_trace_start(), along with the value to set the next field in the event. After each field is set, the 'cursor' points to the next field, which will be set by the subsequent call, continuing until all the fields have been set in order. The same sequence of calls as in the above examples using this method would be (without error-handling code):

```
/* next_pid_field */
ret = synth_event_add_next_val(777, &trace_state);

/* next_comm_field */
ret = synth_event_add_next_val((u64)"slinky", &trace_state);

/* ts_ns */
ret = synth_event_add_next_val(1000000, &trace_state);

/* ts_ms */
ret = synth_event_add_next_val(1000, &trace_state);

/* cpu */
ret = synth_event_add_next_val(smp_processor_id(), &trace_state);

/* my_string_field */
ret = synth_event_add_next_val((u64)"thneed_2.01", &trace_state);

/* my_int_field */
ret = synth_event_add_next_val(395, &trace_state);
```

To assign the values in any order, synth_event_add_val() should be used. Each call is passed the same synth_event_trace_state object used in the synth_event_trace_start(), along with the field name of the field to set and the value to set it to. The same sequence of calls as in the above examples using this method would be (without error-handling code):

```
ret = synth_event_add_val("next_pid_field", 777, &trace_state);
ret = synth_event_add_val("next_comm_field", (u64)"silly putty",
                          &trace_state);
ret = synth_event_add_val("ts_ns", 1000000, &trace_state);
ret = synth_event_add_val("ts_ms", 1000, &trace_state);
ret = synth_event_add_val("cpu", smp_processor_id(), &trace_state);
ret = synth_event_add_val("my_string_field", (u64)"thneed_9",
                          &trace_state);
ret = synth_event_add_val("my_int_field", 3999, &trace_state);
```

Note that synth_event_add_next_val() and synth_event_add_val() are incompatible if used within the same trace of an event - either one can be used but not both at the same time.

Finally, the event won't be actually traced until it's 'closed', which is done using synth_event_trace_end(), which takes only the struct synth_event_trace_state object used in the previous calls:

```
ret = synth_event_trace_end(&trace_state);
```

Note that synth_event_trace_end() must be called at the end regardless of whether any of the add calls failed (say due to a bad field

name being passed in).

7.3 Dyamically creating kprobe and kretprobe event definitions

To create a kprobe or kretprobe trace event from kernel code, the kprobe_event_gen_cmd_start() or kretprobe_event_gen_cmd_start() functions can be used.

To create a kprobe event, an empty or partially empty kprobe event should first be created using kprobe_event_gen_cmd_start(). The name of the event and the probe location should be specfied along with one or args each representing a probe field should be supplied to this function. Before calling kprobe_event_gen_cmd_start(), the user should create and initialize a dynevent_cmd object using kprobe_event_cmd_init().

For example, to create a new "schedtest" kprobe event with two fields:

```
struct dynevent_cmd cmd;
char *buf;

/* Create a buffer to hold the generated command */
buf = kzalloc(MAX_DYNEVENT_CMD_LEN, GFP_KERNEL);

/* Before generating the command, initialize the cmd object */
kprobe_event_cmd_init(&cmd, buf, MAX_DYNEVENT_CMD_LEN);

/*
 * Define the gen_kprobe_test event with the first 2 kprobe
 * fields.
 */
ret = kprobe_event_gen_cmd_start(&cmd, "gen_kprobe_test", "do_sys_open",
                                 "dfd=%ax", "filename=%dx");
```

Once the kprobe event object has been created, it can then be populated with more fields. Fields can be added using kprobe_event_add_fields(), supplying the dynevent_cmd object along with a variable arg list of probe fields. For example, to add a couple additional fields, the following call could be made:

```
ret = kprobe_event_add_fields(&cmd, "flags=%cx", "mode=+4($stack)");
```

Once all the fields have been added, the event should be finalized and registered by calling the kprobe_event_gen_cmd_end() or kretprobe_event_gen_cmd_end() functions, depending on whether a kprobe or kretprobe command was started:

```
ret = kprobe_event_gen_cmd_end(&cmd);
```

or:

```
ret = kretprobe_event_gen_cmd_end(&cmd);
```

At this point, the event object is ready to be used for tracing new events.

Similarly, a kretprobe event can be created using kretprobe_event_gen_cmd_start() with a probe name and location and additional params such as $retval:

```
ret = kretprobe_event_gen_cmd_start(&cmd, "gen_kretprobe_test",
                                    "do_sys_open", "$retval");
```

Similar to the synthetic event case, code like the following can be used to enable the newly created kprobe event:

```
gen_kprobe_test = trace_get_event_file(NULL, "kprobes", "gen_kprobe_test");

ret = trace_array_set_clr_event(gen_kprobe_test->tr,
                                "kprobes", "gen_kprobe_test", true);
```

Finally, also similar to synthetic events, the following code can be used to give the kprobe event file back and delete the event:

```
trace_put_event_file(gen_kprobe_test);

ret = kprobe_event_delete("gen_kprobe_test");
```

7.4 The "dynevent_cmd" low-level API

Both the in-kernel synthetic event and kprobe interfaces are built on top of a lower-level "dynevent_cmd" interface. This interface is meant to provide the basis for higher-level interfaces such as the synthetic and kprobe interfaces, which can be used as examples.

The basic idea is simple and amounts to providing a general-purpose layer that can be used to generate trace event commands. The generated command strings can then be passed to the command-parsing and event creation code that already exists in the trace event subsystem for creating the corresponding trace events.

In a nutshell, the way it works is that the higher-level interface code creates a struct dynevent_cmd object, then uses a couple functions, dynevent_arg_add() and dynevent_arg_pair_add() to build up a command string, which finally causes the command to be executed using the dynevent_create() function. The details of the interface are described below.

The first step in building a new command string is to create and initialize an instance of a dynevent_cmd. Here, for instance, we create a dynevent_cmd on the stack and initialize it:

```
struct dynevent_cmd cmd;
char *buf;
int ret;

buf = kzalloc(MAX_DYNEVENT_CMD_LEN, GFP_KERNEL);

dynevent_cmd_init(cmd, buf, maxlen, DYNEVENT_TYPE_FOO,
                  foo_event_run_command);
```

The dynevent_cmd initialization needs to be given a user-specified buffer and the length of the buffer (MAX_DYNEVENT_CMD_LEN can be used for this purpose - at 2k it's generally too big to be comfortably put on the stack, so is dynamically allocated), a dynevent type id, which is meant to be used to check that further API calls are for the correct command type, and a pointer to an event-specific run_command() callback that will be called to actually execute the event-specific command function.

Once that's done, the command string can by built up by successive calls to argument-adding functions.

To add a single argument, define and initialize a struct dynevent_arg or struct dynevent_arg_pair object. Here's an example of the simplest possible arg addition, which is simply to append the given string as a whitespace-separated argument to the command:

```
struct dynevent_arg arg;

dynevent_arg_init(&arg, NULL, 0);

arg.str = name;

ret = dynevent_arg_add(cmd, &arg);
```

The arg object is first initialized using dynevent_arg_init() and in this case the parameters are NULL or 0, which means there's no optional sanity-checking function or separator appended to the end of the arg.

Here's another more complicated example using an 'arg pair', which is used to create an argument that consists of a couple components added together as a unit, for example, a 'type field_name;' arg or a simple expression arg e.g. 'flags=%cx':

```
struct dynevent_arg_pair arg_pair;

dynevent_arg_pair_init(&arg_pair, dynevent_foo_check_arg_fn, 0, ';');

arg_pair.lhs = type;
arg_pair.rhs = name;

ret = dynevent_arg_pair_add(cmd, &arg_pair);
```

Again, the arg_pair is first initialized, in this case with a callback function used to check the sanity of the args (for example, that neither part of the pair is NULL), along with a character to be used to add an operator between the pair (here none) and a separator to be appended onto the end of the arg pair (here ';').

There's also a dynevent_str_add() function that can be used to simply add a string as-is, with no spaces, delimeters, or arg check.

Any number of dynevent_*_add() calls can be made to build up the string (until its length surpasses cmd->maxlen). When all the arguments have been added and the command string is complete, the only thing left to do is run the command, which happens by simply calling dynevent_create():

```
ret = dynevent_create(&cmd);
```

At that point, if the return value is 0, the dynamic event has been created and is ready to use.

See the dynevent_cmd function definitions themselves for the details of the API.