Event Histograms

Documentation written by Tom Zanussi

1. Introduction

Histogram triggers are special event triggers that can be used to aggregate trace event data into histograms. For information on trace events and event triggers, see Documentation/trace/events.rst.

2. Histogram Trigger Command

A histogram trigger command is an event trigger command that 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).

The format of a hist trigger is as follows:

```
hist:keys=<field1[,field2,...]>[:values=<field1[,field2,...]>]
  [:sort=<field1[,field2,...]>][:size=#entries][:pause][:continue]
  [:clear][:name=histname1][:<handler>.<action>] [if <filter>]
```

When a matching event is hit, an entry is added to a hash table using the key(s) and value(s) named. Keys and values correspond to fields in the event's format description. Values must correspond to numeric fields - on an event hit, the value(s) will be added to a sum kept for that field. The special string 'hitcount' can be used in place of an explicit value field - this is simply a count of event hits. If 'values' isn't specified, an implicit 'hitcount' value will be automatically created and used as the only value. Keys can be any field, or the special string 'stacktrace', which will use the event's kernel stacktrace as the key. The keywords 'keys' or 'key' can be used to specify keys, and the keywords 'values', 'vals', or 'val' can be used to specify values. Compound keys consisting of up to two fields can be specified by the 'keys' keyword. Hashing a compound key produces a unique entry in the table for each unique combination of component keys, and can be useful for providing more fine-grained summaries of event data. Additionally, sort keys consisting of up to two fields can be specified by the 'sort' keyword. If more than one field is specified, the result will be a 'sort within a sort': the first key is taken to be the primary sort key and the second the secondary key. If a hist trigger is given a name using the 'name' parameter, its histogram data will be shared with other triggers of the same name, and trigger hits will update this common data. Only triggers with 'compatible' fields can be combined in this way; triggers are 'compatible' if the fields named in the trigger share the same number and type of fields and those fields also have the same names. Note that any two events always share the compatible 'hitcount' and 'stacktrace' fields and can therefore be combined using those fields, however pointless that may be.

'hist' triggers add a 'hist' file to each event's subdirectory. Reading the 'hist' file for the event will dump the hash table in its entirety to stdout. If there are multiple hist triggers attached to an event, there will be a table for each trigger in the output. The table displayed for a named trigger will be the same as any other instance having the same name. Each printed hash table entry is a simple list of the keys and values comprising the entry; keys are printed first and are delineated by curly braces, and are followed by the set of value fields for the entry. By default, numeric fields are displayed as base-10 integers. This can be modified by appending any of the following modifiers to the field name:

.hex	display a number as a hex value
.sym	display an address as a symbol
.sym-offset	display an address as a symbol and offset
.syscall	display a syscall id as a system call name
.execname	display a common_pid as a program name
.log2	display log2 value rather than raw number
.buckets=size	display grouping of values rather than raw number
.usecs	display a common_timestamp in microseconds

Note that in general the semantics of a given field aren't interpreted when applying a modifier to it, but there are some restrictions to be aware of in this regard:

- only the 'hex' modifier can be used for values (because values are essentially sums, and the other modifiers don't make sense in that context).
- the 'execname' modifier can only be used on a 'common_pid'. The reason for this is that the execname is simply the 'comm' value saved for the 'current' process when an event was triggered, which is the same as the common_pid value saved by the event tracing code. Trying to apply that comm value to other pid values wouldn't be correct, and typically events that care save pid-specific comm fields in the event itself.

A typical usage scenario would be the following to enable a hist trigger, read its current contents, and then turn it off:

```
# echo 'hist:keys=skbaddr.hex:vals=len' > \
  /sys/kernel/debug/tracing/events/net/netif_rx/trigger

# cat /sys/kernel/debug/tracing/events/net/netif_rx/hist

# echo '!hist:keys=skbaddr.hex:vals=len' > \
  /sys/kernel/debug/tracing/events/net/netif_rx/trigger
```

The trigger file itself can be read to show the details of the currently attached hist trigger. This information is also displayed at the top of the 'hist' file when read.

By default, the size of the hash table is 2048 entries. The 'size' parameter can be used to specify more or fewer than that. The units are in terms of hashtable entries - if a run uses more entries than specified, the results will show the number of 'drops', the number of hits that were ignored. The size should be a power of 2 between 128 and 131072 (any non-power-of-2 number specified will be rounded up).

The 'sort' parameter can be used to specify a value field to sort on. The default if unspecified is 'hitcount' and the default sort order is 'ascending'. To sort in the opposite direction, append .descending' to the sort key.

The 'pause' parameter can be used to pause an existing hist trigger or to start a hist trigger but not log any events until told to do so. 'continue' or 'cont' can be used to start or restart a paused hist trigger.

The 'clear' parameter will clear the contents of a running hist trigger and leave its current paused/active state.

Note that the 'pause', 'cont', and 'clear' parameters should be applied using 'append' shell operator ('>>') if applied to an existing trigger, rather than via the '>' operator, which will cause the trigger to be removed through truncation.

- enable_hist/disable_hist

  The enable_hist and disable_hist triggers can be used to have one event conditionally start and stop another event's already-attached hist trigger. Any number of enable_hist and disable_hist triggers can be attached to a given event, allowing that event to kick off and stop aggregations on a host of other events.

  The format is very similar to the enable/disable_event triggers:

  ```
  enable_hist:<system>:<event>[:count]
  disable_hist:<system>:<event>[:count]
  ```

  Instead of enabling or disabling the tracing of the target event into the trace buffer as the enable/disable_event triggers do, the enable/disable_hist triggers enable or disable the aggregation of the target event into a hash table.

  A typical usage scenario for the enable_hist/disable_hist triggers would be to first set up a paused hist trigger on some event, followed by an enable_hist/disable_hist pair that turns the hist aggregation on and off when conditions of interest are hit:

  ```
  # echo 'hist:keys=skbaddr.hex:vals=len:pause' > \
     /sys/kernel/debug/tracing/events/net/netif_receive_skb/trigger

   # echo 'enable_hist:net:netif_receive_skb if filename==/usr/bin/wget' > \
  ```

```
  /sys/kernel/debug/tracing/events/sched/sched_process_exec/trigger

# echo 'disable_hist:net:netif_receive_skb if comm==wget' > \
  /sys/kernel/debug/tracing/events/sched/sched_process_exit/trigger
```

The above sets up an initially paused hist trigger which is unpaused and starts aggregating events when a given program is executed, and which stops aggregating when the process exits and the hist trigger is paused again.

The examples below provide a more concrete illustration of the concepts and typical usage patterns discussed above.

'special' event fields

There are a number of 'special event fields' available for use as keys or values in a hist trigger. These look like and behave as if they were actual event fields, but aren't really part of the event's field definition or format file. They are however available for any event, and can be used anywhere an actual event field could be. They are:

common_timestamp	u64	timestamp (from ring buffer) associated with the event, in nanoseconds. May be modified by .usecs to have timestamps interpreted as microseconds.
common_cpu	int	the cpu on which the event occurred.

Extended error information

For some error conditions encountered when invoking a hist trigger command, extended error information is available via the tracing/error_log file. See Error Conditions in :file:`Documentation/trace/ftrace.rst` for details.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\trace\(linux-master)(Documentation)(trace)histogram.rst`, line 201); *backlink*

Unknown interpreted text role "file".

6.2 'hist' trigger examples

The first set of examples creates aggregations using the kmalloc event. The fields that can be used for the hist trigger are listed in the kmalloc event's format file:

```
# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/format
name: kmalloc
ID: 374
format:
    field:unsigned short common_type;       offset:0;       size:2; signed:0;
    field:unsigned char common_flags;       offset:2;       size:1; signed:0;
    field:unsigned char common_preempt_count;               offset:3;       size:1; signed:0;
    field:int common_pid;                                   offset:4;       size:4; signed:1;

    field:unsigned long call_site;                          offset:8;       size:8; signed:0;
    field:const void * ptr;                                 offset:16;      size:8; signed:0;
    field:size_t bytes_req;                                 offset:24;      size:8; signed:0;
    field:size_t bytes_alloc;                               offset:32;      size:8; signed:0;
    field:gfp_t gfp_flags;                                  offset:40;      size:4; signed:0;
```

We'll start by creating a hist trigger that generates a simple table that lists the total number of bytes requested for each function in the kernel that made one or more calls to kmalloc:

```
# echo 'hist:key=call_site:val=bytes_req.buckets=32' > \
        /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger
```

This tells the tracing system to create a 'hist' trigger using the call_site field of the kmalloc event as the key for the table, which just means that each unique call_site address will have an entry created for it in the table. The 'val=bytes_req' parameter tells the hist trigger that for each unique entry (call_site) in the table, it should keep a running total of the number of bytes requested by that call_site.

We'll let it run for awhile and then dump the contents of the 'hist' file in the kmalloc event's subdirectory (for readability, a number of entries have been omitted):

```
# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/hist
# trigger info: hist:keys=call_site:vals=bytes_req:sort=hitcount:size=2048 [active]

{ call_site: 18446744072106379007 } hitcount:          1  bytes_req:        176
{ call_site: 18446744071579557049 } hitcount:          1  bytes_req:       1024
{ call_site: 18446744071580608289 } hitcount:          1  bytes_req:      16384
{ call_site: 18446744071581827654 } hitcount:          1  bytes_req:         24
{ call_site: 18446744071580700980 } hitcount:          1  bytes_req:          8
{ call_site: 18446744071579359876 } hitcount:          1  bytes_req:        152
{ call_site: 18446744071580795365 } hitcount:          3  bytes_req:        144
{ call_site: 18446744071581303129 } hitcount:          3  bytes_req:        144
{ call_site: 18446744071580713234 } hitcount:          4  bytes_req:       2560
{ call_site: 18446744071580933750 } hitcount:          4  bytes_req:        736
.
.
.
{ call_site: 18446744072106047046 } hitcount:         69  bytes_req:       5576
{ call_site: 18446744071582116407 } hitcount:         73  bytes_req:       2336
{ call_site: 18446744072106054684 } hitcount:        136  bytes_req:     140504
{ call_site: 18446744072106224230 } hitcount:        136  bytes_req:      19584
{ call_site: 18446744072106078074 } hitcount:        153  bytes_req:       2448
{ call_site: 18446744072106062406 } hitcount:        153  bytes_req:      36720
{ call_site: 18446744071582507929 } hitcount:        153  bytes_req:      37088
{ call_site: 18446744072102520590 } hitcount:        273  bytes_req:      10920
{ call_site: 18446744071582143559 } hitcount:        358  bytes_req:        716
{ call_site: 18446744072106465852 } hitcount:        417  bytes_req:      56712
{ call_site: 18446744072102523378 } hitcount:        485  bytes_req:      27160
{ call_site: 18446744072099568646 } hitcount:       1676  bytes_req:      33520

Totals:
    Hits: 4610
    Entries: 45
    Dropped: 0
```

The output displays a line for each entry, beginning with the key specified in the trigger, followed by the value(s) also specified in the trigger. At the beginning of the output is a line that displays the trigger info, which can also be displayed by reading the 'trigger' file:

```
# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger
hist:keys=call_site:vals=bytes_req:sort=hitcount:size=2048 [active]
```

At the end of the output are a few lines that display the overall totals for the run. The 'Hits' field shows the total number of times the event trigger was hit, the 'Entries' field shows the total number of used entries in the hash table, and the 'Dropped' field shows the number of hits that were dropped because the number of used entries for the run exceeded the maximum number of entries allowed for the table (normally 0, but if not a hint that you may want to increase the size of the table using the 'size' parameter).

Notice in the above output that there's an extra field, 'hitcount', which wasn't specified in the trigger. Also notice that in the trigger info output, there's a parameter, 'sort=hitcount', which wasn't specified in the trigger either. The reason for that is that every trigger implicitly keeps a count of the total number of hits attributed to a given entry, called the 'hitcount'. That hitcount information is explicitly displayed in the output, and in the absence of a user-specified sort parameter, is used as the default sort field.

The value 'hitcount' can be used in place of an explicit value in the 'values' parameter if you don't really need to have any particular field summed and are mainly interested in hit frequencies.

To turn the hist trigger off, simply call up the trigger in the command history and re-execute it with a '!' prepended:

```
# echo '!hist:key=call_site:val=bytes_req' > \
       /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger
```

Finally, notice that the call_site as displayed in the output above isn't really very useful. It's an address, but normally addresses are displayed in hex. To have a numeric field displayed as a hex value, simply append '.hex' to the field name in the trigger:

```
# echo 'hist:key=call_site.hex:val=bytes_req' > \
       /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger

# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/hist
# trigger info: hist:keys=call_site.hex:vals=bytes_req:sort=hitcount:size=2048 [active]

{ call_site: ffffffffa026b291 } hitcount:          1  bytes_req:        433
{ call_site: ffffffffa07186ff } hitcount:          1  bytes_req:        176
{ call_site: ffffffff811ae721 } hitcount:          1  bytes_req:      16384
{ call_site: ffffffff811c5134 } hitcount:          1  bytes_req:          8
{ call_site: ffffffffa04a9ebb } hitcount:          1  bytes_req:        511
{ call_site: ffffffff8122e0a6 } hitcount:          1  bytes_req:         12
{ call_site: ffffffff8107da84 } hitcount:          1  bytes_req:        152
{ call_site: ffffffff812d8246 } hitcount:          1  bytes_req:         24
{ call_site: ffffffff811dc1e5 } hitcount:          3  bytes_req:        144
{ call_site: ffffffffa02515e8 } hitcount:          3  bytes_req:        648
{ call_site: ffffffff81258159 } hitcount:          3  bytes_req:        144
{ call_site: ffffffff811c80f4 } hitcount:          4  bytes_req:        544
.
.
.
{ call_segment: ffffffffa06c7646 } hitcount:        106  bytes_req:       8024
{ call_site: ffffffffa06cb246 } hitcount:        132  bytes_req:      31680
{ call_site: ffffffffa06cef7a } hitcount:        132  bytes_req:       2112
{ call_site: ffffffff8137e399 } hitcount:        132  bytes_req:      23232
{ call_site: ffffffffa06c941c } hitcount:        185  bytes_req:     171360
{ call_site: ffffffffa06f2a66 } hitcount:        185  bytes_req:      26640
{ call_site: ffffffffa036a70e } hitcount:        265  bytes_req:      10600
{ call_site: ffffffff81325447 } hitcount:        292  bytes_req:        584
{ call_site: ffffffffa072da3c } hitcount:        446  bytes_req:      60656
{ call_site: ffffffffa036b1f2 } hitcount:        526  bytes_req:      29456
{ call_site: ffffffffa0099c06 } hitcount:       1780  bytes_req:      35600

Totals:
    Hits: 4775
    Entries: 46
    Dropped: 0
```

Even that's only marginally more useful - while hex values do look more like addresses, what users are typically more interested in when looking at text addresses are the corresponding symbols instead. To have an address displayed as symbolic value instead, simply append '.sym' or '.sym-offset' to the field name in the trigger:

```
# echo 'hist:key=call_site.sym:val=bytes_req' > \
       /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger

# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/hist
# trigger info: hist:keys=call_site.sym:vals=bytes_req:sort=hitcount:size=2048 [active]

{ call_site: [ffffffff810adcb9] syslog_print_all                             } hitcount:          1  bytes_req:       1024
{ call_site: [ffffffff8154bc62] usb_control_msg                              } hitcount:          1  bytes_req:          8
{ call_site: [ffffffffa00bf6fe] hidraw_send_report [hid]                     } hitcount:          1  bytes_req:          7
{ call_site: [ffffffff8154acbe] usb_alloc_urb                                } hitcount:          1  bytes_req:        192
{ call_site: [ffffffffa00bf1ca] hidraw_report_event [hid]                    } hitcount:          1  bytes_req:          7
{ call_site: [ffffffff811e3a25] __seq_open_private                           } hitcount:          1  bytes_req:         40
{ call_site: [ffffffff8109524a] alloc_fair_sched_group                       } hitcount:          2  bytes_req:        128
{ call_site: [ffffffff811febd5] fsnotify_alloc_group                         } hitcount:          2  bytes_req:        528
{ call_site: [ffffffff81440f58] __tty_buffer_request_room                    } hitcount:          2  bytes_req:       2624
{ call_site: [ffffffff81200ba6] inotify_new_group                            } hitcount:          2  bytes_req:         96
{ call_site: [ffffffffa05e19af] ieee80211_start_tx_ba_session [mac80211]     } hitcount:          2  bytes_req:        464
{ call_site: [ffffffff81672406] tcp_get_metrics                              } hitcount:          2  bytes_req:        304
{ call_site: [ffffffff81097ec2] alloc_rt_sched_group                         } hitcount:          2  bytes_req:        128
{ call_site: [ffffffff81089b05] sched_create_group                           } hitcount:          2  bytes_req:       1424
.
.
.
{ call_site: [ffffffffa04a580c] intel_crtc_page_flip [i915]                  } hitcount:       1185  bytes_req:     123240
{ call_site: [ffffffffa0287592] drm_mode_page_flip_ioctl [drm]               } hitcount:       1185  bytes_req:     104280
{ call_site: [ffffffffa04c4a3c] intel_plane_duplicate_state [i915]           } hitcount:       1402  bytes_req:     190672
{ call_site: [ffffffff812891ca] ext4_find_extent                             } hitcount:       1518  bytes_req:     146208
{ call_site: [ffffffffa029070e] drm_vma_node_allow [drm]                     } hitcount:       1746  bytes_req:      69840
{ call_site: [ffffffffa045e7c4] i915_gem_do_execbuffer.isra.23 [i915]        } hitcount:       2021  bytes_req:     792312
{ call_site: [ffffffffa02911f2] drm_modeset_lock_crtc [drm]                  } hitcount:       2592  bytes_req:     145152
{ call_site: [ffffffffa0489a66] intel_ring_begin [i915]                      } hitcount:       2629  bytes_req:     378576
{ call_site: [ffffffffa046041c] i915_gem_execbuffer2 [i915]                  } hitcount:       2629  bytes_req:    3783248
{ call_site: [ffffffff81325607] apparmor_file_alloc_security                 } hitcount:       5192  bytes_req:      10384
{ call_site: [ffffffffa00b7c06] hid_report_raw_event [hid]                   } hitcount:       5529  bytes_req:     110584
{ call_site: [ffffffff8131ebf7] aa_alloc_task_context                        } hitcount:      21943  bytes_req:     702176
{ call_site: [ffffffff8125847d] ext4_htree_store_dirent                      } hitcount:      55759  bytes_req:    5074265

Totals:
    Hits: 109928
    Entries: 71
    Dropped: 0
```

Because the default sort key above is 'hitcount', the above shows a the list of call_sites by increasing hitcount, so that at the bottom we see the functions that made the most kmalloc calls during the run. If instead we we wanted to see the top kmalloc callers in terms of the number of bytes requested rather than the number of calls, and we wanted the top caller to appear at the top, we can use the 'sort' parameter, along with the 'descending' modifier:

```
# echo 'hist:key=call_site.sym:val=bytes_req:sort=bytes_req.descending' > \
       /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger

# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/hist
# trigger info: hist:keys=call_site.sym:vals=bytes_req:sort=bytes_req.descending:size=2048 [active]

{ call_site: [ffffffffa046041c] i915_gem_execbuffer2 [i915]                  } hitcount:       2186  bytes_req:    3397464
{ call_site: [ffffffffa045e7c4] i915_gem_do_execbuffer.isra.23 [i915]        } hitcount:       1790  bytes_req:     712176
{ call_site: [ffffffff8125847d] ext4_htree_store_dirent                      } hitcount:       8132  bytes_req:     513135
{ call_site: [ffffffff811e2a1b] seq_buf_alloc                                } hitcount:        106  bytes_req:     440128
{ call_site: [ffffffffa0489a66] intel_ring_begin [i915]                      } hitcount:       2186  bytes_req:     314784
{ call_site: [ffffffff812891ca] ext4_find_extent                             } hitcount:       2174  bytes_req:     208992
{ call_site: [ffffffff811ae8e1] __kmalloc                                    } hitcount:          8  bytes_req:     131072
{ call_site: [ffffffffa04c4a3c] intel_plane_duplicate_state [i915]           } hitcount:        859  bytes_req:     116824
{ call_site: [ffffffffa02911f2] drm_modeset_lock_crtc [drm]                  } hitcount:       1834  bytes_req:     102704
{ call_site: [ffffffffa04a580c] intel_crtc_page_flip [i915]                  } hitcount:        972  bytes_req:     101088
{ call_site: [ffffffffa0287592] drm_mode_page_flip_ioctl [drm]               } hitcount:        972  bytes_req:      85536
{ call_site: [ffffffffa00b7c06] hid_report_raw_event [hid]                   } hitcount:       3333  bytes_req:      66664
{ call_site: [ffffffff8137e559] sg_kmalloc                                   } hitcount:        209  bytes_req:      61632
.
.
.
{ call_site: [ffffffff81095225] alloc_fair_sched_group                       } hitcount:          2  bytes_req:        128
{ call_site: [ffffffff81097ec2] alloc_rt_sched_group                         } hitcount:          2  bytes_req:        128
{ call_site: [ffffffff812d8406] copy_semundo                                 } hitcount:          2  bytes_req:         48
{ call_site: [ffffffff81200ba6] inotify_new_group                            } hitcount:          1  bytes_req:         48
{ call_site: [ffffffffa027121a] drm_getmagic [drm]                           } hitcount:          1  bytes_req:         48
{ call_site: [ffffffff811e3a25] __seq_open_private                           } hitcount:          1  bytes_req:         40
{ call_site: [ffffffff811c52f4] bprm_change_interp                           } hitcount:          2  bytes_req:         16
{ call_site: [ffffffff8154bc62] usb_control_msg                              } hitcount:          1  bytes_req:          8
{ call_site: [ffffffffa00bf1ca] hidraw_report_event [hid]                    } hitcount:          1  bytes_req:          7
```

```
{ call_site: [ffffffffa00bf6fe] hidraw_send_report [hid]                    } hitcount:          1  bytes_req:          7

Totals:
    Hits: 32133
    Entries: 81
    Dropped: 0
```

To display the offset and size information in addition to the symbol name, just use 'sym-offset' instead:

```
# echo 'hist:key=call_site.sym-offset:val=bytes_req:sort=bytes_req.descending' > \
       /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger

# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/hist
# trigger info: hist:keys=call_site.sym-offset:vals=bytes_req:sort=bytes_req.descending:size=2048 [active]

{ call_site: [ffffffffa046041c] i915_gem_execbuffer2+0x6c/0x2c0 [i915]      } hitcount:       4569  bytes_req:    3163720
{ call_site: [ffffffffa0489a66] intel_ring_begin+0xc6/0x1f0 [i915]          } hitcount:       4569  bytes_req:     657936
{ call_site: [ffffffffa045e7c4] i915_gem_do_execbuffer.isra.23+0x694/0x1020 [i915]  } hitcount:       1519  bytes_req:     472936
{ call_site: [ffffffffa045e646] i915_gem_do_execbuffer.isra.23+0x516/0x1020 [i915]  } hitcount:       3050  bytes_req:     211832
{ call_site: [ffffffff811e2a1b] seq_buf_alloc+0x1b/0x50                     } hitcount:         34  bytes_req:     148384
{ call_site: [ffffffffa04a580c] intel_crtc_page_flip+0xbc/0x870 [i915]      } hitcount:       1385  bytes_req:     144040
{ call_site: [ffffffff811ae8e1] __kmalloc+0x191/0x1b0                       } hitcount:          8  bytes_req:     131072
{ call_site: [ffffffffa0287592] drm_mode_page_flip_ioctl+0x282/0x360 [drm]  } hitcount:       1385  bytes_req:     121880
{ call_site: [ffffffffa02911f2] drm_modeset_lock_crtc+0x32/0x100 [drm]      } hitcount:       1848  bytes_req:     103488
{ call_site: [ffffffffa04c4a3c] intel_plane_duplicate_state+0x2c/0xa0 [i915]  } hitcount:        461  bytes_req:      62696
{ call_site: [ffffffffa029070e] drm_vma_node_allow+0x2e/0xd0 [drm]          } hitcount:       1541  bytes_req:      61640
{ call_site: [ffffffff815f8d7b] sk_prot_alloc+0xcb/0x1b0                    } hitcount:         57  bytes_req:      57456
.
.
.
{ call_site: [ffffffff8109524a] alloc_fair_sched_group+0x5a/0x1a0           } hitcount:          2  bytes_req:        128
{ call_site: [ffffffffa027b921] drm_vm_open_locked+0x31/0xa0 [drm]          } hitcount:          3  bytes_req:         96
{ call_site: [ffffffff8122e266] proc_self_follow_link+0x76/0xb0             } hitcount:          8  bytes_req:         96
{ call_site: [ffffffff81213e80] load_elf_binary+0x240/0x1650                } hitcount:          3  bytes_req:         84
{ call_site: [ffffffff8154bc62] usb_control_msg+0x42/0x110                  } hitcount:          1  bytes_req:          8
{ call_site: [ffffffffa00bf6fe] hidraw_send_report+0x7e/0x1a0 [hid]         } hitcount:          1  bytes_req:          7
{ call_site: [ffffffffa00bf1ca] hidraw_report_event+0x8a/0x120 [hid]        } hitcount:          1  bytes_req:          7

Totals:
    Hits: 26098
    Entries: 64
    Dropped: 0
```

We can also add multiple fields to the 'values' parameter. For example, we might want to see the total number of bytes allocated alongside bytes requested, and display the result sorted by bytes allocated in a descending order:

```
# echo 'hist:keys=call_site.sym:values=bytes_req,bytes_alloc:sort=bytes_alloc.descending' > \
       /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger

# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/hist
# trigger info: hist:keys=call_site.sym:vals=bytes_req,bytes_alloc:sort=bytes_alloc.descending:size=2048 [active]

{ call_site: [ffffffffa046041c] i915_gem_execbuffer2 [i915]                 } hitcount:       7403  bytes_req:    4084360  bytes_al
{ call_site: [ffffffff811e2a1b] seq_buf_alloc                               } hitcount:        541  bytes_req:    2213968  bytes_al
{ call_site: [ffffffffa0489a66] intel_ring_begin [i915]                     } hitcount:       7404  bytes_req:    1066176  bytes_al
{ call_site: [ffffffffa045e7c4] i915_gem_do_execbuffer.isra.23 [i915]       } hitcount:       1565  bytes_req:     557368  bytes_al
{ call_site: [ffffffff8125847d] ext4_htree_store_dirent                     } hitcount:       9557  bytes_req:     595778  bytes_al
{ call_site: [ffffffffa045e646] i915_gem_do_execbuffer.isra.23 [i915]       } hitcount:       5839  bytes_req:     430680  bytes_al
{ call_site: [ffffffffa04c4a3c] intel_plane_duplicate_state [i915]          } hitcount:       2388  bytes_req:     324768  bytes_al
{ call_site: [ffffffffa02911f2] drm_modeset_lock_crtc [drm]                 } hitcount:       3911  bytes_req:     219016  bytes_al
{ call_site: [ffffffff815f8d7b] sk_prot_alloc                               } hitcount:        235  bytes_req:     236880  bytes_al
{ call_site: [ffffffff8137e559] sg_kmalloc                                  } hitcount:        557  bytes_req:     169024  bytes_al
{ call_site: [ffffffffa00b7c06] hid_report_raw_event [hid]                  } hitcount:       9378  bytes_req:     187548  bytes_al
{ call_site: [ffffffffa04a580c] intel_crtc_page_flip [i915]                 } hitcount:       1519  bytes_req:     157976  bytes_al
.
.
.
{ call_site: [ffffffff8109bd3b] sched_autogroup_create_attach               } hitcount:          2  bytes_req:        144  bytes_al
{ call_site: [ffffffff81097ee8] alloc_rt_sched_group                        } hitcount:          2  bytes_req:        128  bytes_al
{ call_site: [ffffffff8109524a] alloc_fair_sched_group                      } hitcount:          2  bytes_req:        128  bytes_al
{ call_site: [ffffffff81095225] alloc_fair_sched_group                      } hitcount:          2  bytes_req:        128  bytes_al
{ call_site: [ffffffff81097ec2] alloc_rt_sched_group                        } hitcount:          2  bytes_req:        128  bytes_al
{ call_site: [ffffffff81213e80] load_elf_binary                             } hitcount:          3  bytes_req:         84  bytes_al
{ call_site: [ffffffff81079a2e] kthread_create_on_node                      } hitcount:          1  bytes_req:         56  bytes_al
{ call_site: [ffffffffa00bf6fe] hidraw_send_report [hid]                    } hitcount:          1  bytes_req:          7  bytes_al
{ call_site: [ffffffff8154bc62] usb_control_msg                             } hitcount:          1  bytes_req:          8  bytes_al
{ call_site: [ffffffffa00bf1ca] hidraw_report_event [hid]                   } hitcount:          1  bytes_req:          7  bytes_al

Totals:
    Hits: 66598
    Entries: 65
    Dropped: 0
```

Finally, to finish off our kmalloc example, instead of simply having the hist trigger display symbolic call_sites, we can have the hist trigger additionally display the complete set of kernel stack traces that led to each call_site. To do that, we simply use the special value 'stacktrace' for the key parameter:

```
# echo 'hist:keys=stacktrace:values=bytes_req,bytes_alloc:sort=bytes_alloc' > \
       /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger
```

The above trigger will use the kernel stack trace in effect when an event is triggered as the key for the hash table. This allows the enumeration of every kernel callpath that led up to a particular event, along with a running total of any of the event fields for that event. Here we tally bytes requested and bytes allocated for every callpath in the system that led up to a kmalloc (in this case every callpath to a kmalloc for a kernel compile):

```
# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/hist
# trigger info: hist:keys=stacktrace:vals=bytes_req,bytes_alloc:sort=bytes_alloc:size=2048 [active]

{ stacktrace:
     __kmalloc_track_caller+0x10b/0x1a0
     kmemdup+0x20/0x50
     hidraw_report_event+0x8a/0x120 [hid]
     hid_report_raw_event+0x3ea/0x440 [hid]
     hid_input_report+0x112/0x190 [hid]
     hid_irq_in+0xc2/0x260 [usbhid]
     __usb_hcd_giveback_urb+0x72/0x120
     usb_giveback_urb_bh+0x9e/0xe0
     tasklet_hi_action+0xf8/0x100
     __do_softirq+0x114/0x2c0
     irq_exit+0xa5/0xb0
     do_IRQ+0x5a/0xf0
     ret_from_intr+0x0/0x30
     cpuidle_enter+0x17/0x20
     cpu_startup_entry+0x315/0x3e0
     rest_init+0x7c/0x80
} hitcount:          3  bytes_req:         21  bytes_alloc:         24
{ stacktrace:
     __kmalloc_track_caller+0x10b/0x1a0
     kmemdup+0x20/0x50
     hidraw_report_event+0x8a/0x120 [hid]
     hid_report_raw_event+0x3ea/0x440 [hid]
     hid_input_report+0x112/0x190 [hid]
     hid_irq_in+0xc2/0x260 [usbhid]
     __usb_hcd_giveback_urb+0x72/0x120
     usb_giveback_urb_bh+0x9e/0xe0
     tasklet_hi_action+0xf8/0x100
     __do_softirq+0x114/0x2c0
     irq_exit+0xa5/0xb0
     do_IRQ+0x5a/0xf0
```

```
    ret_from_intr+0x0/0x30
} hitcount:          3  bytes_req:         21  bytes_alloc:         24
{ stacktrace:
    kmem_cache_alloc_trace+0xeb/0x150
    aa_alloc_task_context+0x27/0x40
    apparmor_cred_prepare+0x1f/0x50
    security_prepare_creds+0x16/0x20
    prepare_creds+0xdf/0x1a0
    SyS_capset+0xb5/0x200
    system_call_fastpath+0x12/0x6a
} hitcount:          1  bytes_req:         32  bytes_alloc:         32
.
.
.
{ stacktrace:
    __kmalloc+0x11b/0x1b0
    i915_gem_execbuffer2+0x6c/0x2c0 [i915]
    drm_ioctl+0x349/0x670 [drm]
    do_vfs_ioctl+0x2f0/0x4f0
    SyS_ioctl+0x81/0xa0
    system_call_fastpath+0x12/0x6a
} hitcount:      17726  bytes_req:   13944120  bytes_alloc:   19593808
{ stacktrace:
    __kmalloc+0x11b/0x1b0
    load_elf_phdrs+0x76/0xa0
    load_elf_binary+0x102/0x1650
    search_binary_handler+0x97/0x1d0
    do_execveat_common.isra.34+0x551/0x6e0
    SyS_execve+0x3a/0x50
    return_from_execve+0x0/0x23
} hitcount:      33348  bytes_req:   17152128  bytes_alloc:   20226048
{ stacktrace:
    kmem_cache_alloc_trace+0xeb/0x150
    apparmor_file_alloc_security+0x27/0x40
    security_file_alloc+0x16/0x20
    get_empty_filp+0x93/0x1c0
    path_openat+0x31/0x5f0
    do_filp_open+0x3a/0x90
    do_sys_open+0x128/0x220
    SyS_open+0x1e/0x20
    system_call_fastpath+0x12/0x6a
} hitcount:    4766422  bytes_req:    9532844  bytes_alloc:   38131376
{ stacktrace:
    __kmalloc+0x11b/0x1b0
    seq_buf_alloc+0x1b/0x50
    seq_read+0x2cc/0x370
    proc_reg_read+0x3d/0x80
    __vfs_read+0x28/0xe0
    vfs_read+0x86/0x140
    SyS_read+0x46/0xb0
    system_call_fastpath+0x12/0x6a
} hitcount:      19133  bytes_req:   78368768  bytes_alloc:   78368768

Totals:
    Hits: 6085872
    Entries: 253
    Dropped: 0
```

If you key a hist trigger on common_pid, in order for example to gather and display sorted totals for each process, you can use the special .execname modifier to display the executable names for the processes in the table rather than raw pids. The example below keeps a per-process sum of total bytes read:

```
# echo 'hist:key=common_pid.execname:val=count:sort=count.descending' > \
       /sys/kernel/debug/tracing/events/syscalls/sys_enter_read/trigger

# cat /sys/kernel/debug/tracing/events/syscalls/sys_enter_read/hist
# trigger info: hist:keys=common_pid.execname:vals=count:sort=count.descending:size=2048 [active]

{ common_pid: gnome-terminal  [      3196] } hitcount:        280  count:    1093512
{ common_pid: Xorg            [      1309] } hitcount:        525  count:     256640
{ common_pid: compiz          [      2889] } hitcount:         59  count:     254400
{ common_pid: bash            [      8710] } hitcount:          3  count:      66369
{ common_pid: dbus-daemon-lau [      8703] } hitcount:         49  count:      47739
{ common_pid: irqbalance      [      1252] } hitcount:         27  count:      27648
{ common_pid: 01ifupdown      [      8705] } hitcount:          3  count:      17216
{ common_pid: dbus-daemon     [       772] } hitcount:         10  count:      12396
{ common_pid: Socket Thread   [      8342] } hitcount:         11  count:      11264
{ common_pid: nm-dhcp-client. [      8701] } hitcount:          6  count:       7424
{ common_pid: gmain           [      1315] } hitcount:         18  count:       6336
.
.
.
{ common_pid: postgres        [      1892] } hitcount:          2  count:         32
{ common_pid: postgres        [      1891] } hitcount:          2  count:         32
{ common_pid: gmain           [      8704] } hitcount:          2  count:         32
{ common_pid: upstart-dbus-br [      2740] } hitcount:         21  count:         21
{ common_pid: nm-dispatcher.a [      8696] } hitcount:          1  count:         16
{ common_pid: indicator-datet [      2904] } hitcount:          1  count:         16
{ common_pid: gdbus           [      2998] } hitcount:          1  count:         16
{ common_pid: rtkit-daemon    [      2052] } hitcount:          1  count:          8
{ common_pid: init            [         1] } hitcount:          2  count:          2

Totals:
    Hits: 2116
    Entries: 51
    Dropped: 0
```

Similarly, if you key a hist trigger on syscall id, for example to gather and display a list of systemwide syscall hits, you can use the special .syscall modifier to display the syscall names rather than raw ids. The example below keeps a running total of syscall counts for the system during the run:

```
# echo 'hist:key=id.syscall:val=hitcount' > \
       /sys/kernel/debug/tracing/events/raw_syscalls/sys_enter/trigger

# cat /sys/kernel/debug/tracing/events/raw_syscalls/sys_enter/hist
# trigger info: hist:keys=id.syscall:vals=hitcount:sort=hitcount:size=2048 [active]

{ id: sys_fsync                    [ 74] } hitcount:          1
{ id: sys_newuname                 [ 63] } hitcount:          1
{ id: sys_prctl                    [157] } hitcount:          1
{ id: sys_statfs                   [137] } hitcount:          1
{ id: sys_symlink                  [ 88] } hitcount:          1
{ id: sys_sendmmsg                 [307] } hitcount:          1
{ id: sys_semctl                   [ 66] } hitcount:          1
{ id: sys_readlink                 [ 89] } hitcount:          3
{ id: sys_bind                     [ 49] } hitcount:          3
{ id: sys_getsockname              [ 51] } hitcount:          3
{ id: sys_unlink                   [ 87] } hitcount:          3
{ id: sys_rename                   [ 82] } hitcount:          4
{ id: unknown_syscall              [ 58] } hitcount:          4
{ id: sys_connect                  [ 42] } hitcount:          4
{ id: sys_getpid                   [ 39] } hitcount:          4
.
.
.
{ id: sys_rt_sigprocmask           [ 14] } hitcount:        952
{ id: sys_futex                    [202] } hitcount:       1534
{ id: sys_write                    [  1] } hitcount:       2689
{ id: sys_setitimer                [ 38] } hitcount:       2797
{ id: sys_read                     [  0] } hitcount:       3202
```

```
{ id: sys_select                  [ 23] } hitcount:       3773
{ id: sys_writev                  [ 20] } hitcount:       4531
{ id: sys_poll                    [  7] } hitcount:       8314
{ id: sys_recvmsg                 [ 47] } hitcount:      13738
{ id: sys_ioctl                   [ 16] } hitcount:      21843

Totals:
    Hits: 67612
    Entries: 72
    Dropped: 0
```

The syscall counts above provide a rough overall picture of system call activity on the system; we can see for example that the most popular system call on this system was the 'sys_ioctl' system call.

We can use 'compound' keys to refine that number and provide some further insight as to which processes exactly contribute to the overall ioctl count.

The command below keeps a hitcount for every unique combination of system call id and pid - the end result is essentially a table that keeps a per-pid sum of system call hits. The results are sorted using the system call id as the primary key, and the hitcount sum as the secondary key:

```
# echo 'hist:key=id.syscall,common_pid.execname:val=hitcount:sort=id,hitcount' > \
       /sys/kernel/debug/tracing/events/raw_syscalls/sys_enter/trigger

# cat /sys/kernel/debug/tracing/events/raw_syscalls/sys_enter/hist
# trigger info: hist:keys=id.syscall,common_pid.execname:vals=hitcount:sort=id.syscall,hitcount:size=2048 [active]

{ id: sys_read                    [  0], common_pid: rtkit-daemon    [      1877] } hitcount:          1
{ id: sys_read                    [  0], common_pid: gdbus           [      2976] } hitcount:          1
{ id: sys_read                    [  0], common_pid: console-kit-dae [      3400] } hitcount:          1
{ id: sys_read                    [  0], common_pid: postgres        [      1865] } hitcount:          1
{ id: sys_read                    [  0], common_pid: deja-dup-monito [      3543] } hitcount:          2
{ id: sys_read                    [  0], common_pid: NetworkManager  [       890] } hitcount:          2
{ id: sys_read                    [  0], common_pid: evolution-calen [      3048] } hitcount:          2
{ id: sys_read                    [  0], common_pid: postgres        [      1864] } hitcount:          2
{ id: sys_read                    [  0], common_pid: nm-applet       [      3022] } hitcount:          2
{ id: sys_read                    [  0], common_pid: whoopsie        [      1212] } hitcount:          2
.
.
.
{ id: sys_ioctl                   [ 16], common_pid: bash            [      8479] } hitcount:          1
{ id: sys_ioctl                   [ 16], common_pid: bash            [      3472] } hitcount:         12
{ id: sys_ioctl                   [ 16], common_pid: gnome-terminal  [      3199] } hitcount:         16
{ id: sys_ioctl                   [ 16], common_pid: Xorg            [      1267] } hitcount:       1808
{ id: sys_ioctl                   [ 16], common_pid: compiz          [      2994] } hitcount:       5580
.
.
.
{ id: sys_waitid                  [247], common_pid: upstart-dbus-br [      2690] } hitcount:          3
{ id: sys_waitid                  [247], common_pid: upstart-dbus-br [      2688] } hitcount:         16
{ id: sys_inotify_add_watch       [254], common_pid: gmain           [       975] } hitcount:          2
{ id: sys_inotify_add_watch       [254], common_pid: gmain           [      3204] } hitcount:          4
{ id: sys_inotify_add_watch       [254], common_pid: gmain           [      2888] } hitcount:          4
{ id: sys_inotify_add_watch       [254], common_pid: gmain           [      3003] } hitcount:          4
{ id: sys_inotify_add_watch       [254], common_pid: gmain           [      2873] } hitcount:          4
{ id: sys_inotify_add_watch       [254], common_pid: gmain           [      3196] } hitcount:          6
{ id: sys_openat                  [257], common_pid: java            [      2623] } hitcount:          2
{ id: sys_eventfd2                [290], common_pid: ibus-ui-gtk3    [      2760] } hitcount:          4
{ id: sys_eventfd2                [290], common_pid: compiz          [      2994] } hitcount:          6

Totals:
    Hits: 31536
    Entries: 323
    Dropped: 0
```

The above list does give us a breakdown of the ioctl syscall by pid, but it also gives us quite a bit more than that, which we don't really care about at the moment. Since we know the syscall id for sys_ioctl (16, displayed next to the sys_ioctl name), we can use that to filter out all the other syscalls:

```
# echo 'hist:key=id.syscall,common_pid.execname:val=hitcount:sort=id,hitcount if id == 16' > \
       /sys/kernel/debug/tracing/events/raw_syscalls/sys_enter/trigger

# cat /sys/kernel/debug/tracing/events/raw_syscalls/sys_enter/hist
# trigger info: hist:keys=id.syscall,common_pid.execname:vals=hitcount:sort=id.syscall,hitcount:size=2048 if id == 16 [active]

{ id: sys_ioctl                   [ 16], common_pid: gmain           [      2769] } hitcount:          1
{ id: sys_ioctl                   [ 16], common_pid: evolution-addre [      8571] } hitcount:          1
{ id: sys_ioctl                   [ 16], common_pid: gmain           [      3003] } hitcount:          1
{ id: sys_ioctl                   [ 16], common_pid: gmain           [      2781] } hitcount:          1
{ id: sys_ioctl                   [ 16], common_pid: gmain           [      2829] } hitcount:          1
{ id: sys_ioctl                   [ 16], common_pid: bash            [      8726] } hitcount:          1
{ id: sys_ioctl                   [ 16], common_pid: bash            [      8508] } hitcount:          1
{ id: sys_ioctl                   [ 16], common_pid: gmain           [      2970] } hitcount:          1
{ id: sys_ioctl                   [ 16], common_pid: gmain           [      2768] } hitcount:          1
.
.
.
{ id: sys_ioctl                   [ 16], common_pid: pool            [      8559] } hitcount:         45
{ id: sys_ioctl                   [ 16], common_pid: pool            [      8555] } hitcount:         48
{ id: sys_ioctl                   [ 16], common_pid: pool            [      8551] } hitcount:         48
{ id: sys_ioctl                   [ 16], common_pid: avahi-daemon    [       896] } hitcount:         66
{ id: sys_ioctl                   [ 16], common_pid: Xorg            [      1267] } hitcount:      26674
{ id: sys_ioctl                   [ 16], common_pid: compiz          [      2994] } hitcount:      73443

Totals:
    Hits: 101162
    Entries: 103
    Dropped: 0
```

The above output shows that 'compiz' and 'Xorg' are far and away the heaviest ioctl callers (which might lead to questions about whether they really need to be making all those calls and to possible avenues for further investigation.)

The compound key examples used a key and a sum value (hitcount) to sort the output, but we can just as easily use two keys instead. Here's an example where we use a compound key composed of the the common_pid and size event fields. Sorting with pid as the primary key and 'size' as the secondary key allows us to display an ordered summary of the recvfrom sizes, with counts, received by each process:

```
# echo 'hist:key=common_pid.execname,size:val=hitcount:sort=common_pid,size' > \
       /sys/kernel/debug/tracing/events/syscalls/sys_enter_recvfrom/trigger

# cat /sys/kernel/debug/tracing/events/syscalls/sys_enter_recvfrom/hist
# trigger info: hist:keys=common_pid.execname,size:vals=hitcount:sort=common_pid.execname,size:size=2048 [active]

{ common_pid: smbd            [       784], size:          4 } hitcount:          1
{ common_pid: dnsmasq         [      1412], size:       4096 } hitcount:        672
{ common_pid: postgres        [      1796], size:       1000 } hitcount:          6
{ common_pid: postgres        [      1867], size:       1000 } hitcount:         10
{ common_pid: bamfdaemon      [      2787], size:         28 } hitcount:          2
{ common_pid: bamfdaemon      [      2787], size:      14360 } hitcount:          1
{ common_pid: compiz          [      2994], size:          8 } hitcount:          1
{ common_pid: compiz          [      2994], size:         20 } hitcount:         11
{ common_pid: gnome-terminal  [      3199], size:          4 } hitcount:          2
{ common_pid: firefox         [      8817], size:          4 } hitcount:          1
{ common_pid: firefox         [      8817], size:          8 } hitcount:          5
{ common_pid: firefox         [      8817], size:        588 } hitcount:          2
{ common_pid: firefox         [      8817], size:        628 } hitcount:          1
{ common_pid: firefox         [      8817], size:       6944 } hitcount:          1
{ common_pid: firefox         [      8817], size:     408880 } hitcount:          2
{ common_pid: firefox         [      8822], size:          8 } hitcount:          2
```

```
{ common_pid: firefox         [      8822], size:        160 } hitcount:          2
{ common_pid: firefox         [      8822], size:        320 } hitcount:          2
{ common_pid: firefox         [      8822], size:        352 } hitcount:          1
.
.
.
{ common_pid: pool            [      8923], size:       1960 } hitcount:         10
{ common_pid: pool            [      8923], size:       2048 } hitcount:         10
{ common_pid: pool            [      8924], size:       1960 } hitcount:         10
{ common_pid: pool            [      8924], size:       2048 } hitcount:         10
{ common_pid: pool            [      8928], size:       1964 } hitcount:          4
{ common_pid: pool            [      8928], size:       1965 } hitcount:          2
{ common_pid: pool            [      8928], size:       2048 } hitcount:          6
{ common_pid: pool            [      8929], size:       1982 } hitcount:          1
{ common_pid: pool            [      8929], size:       2048 } hitcount:          1

Totals:
    Hits: 2016
    Entries: 224
    Dropped: 0
```

The above example also illustrates the fact that although a compound key is treated as a single entity for hashing purposes, the sub-keys it's composed of can be accessed independently.

The next example uses a string field as the hash key and demonstrates how you can manually pause and continue a hist trigger. In this example, we'll aggregate fork counts and don't expect a large number of entries in the hash table, so we'll drop it to a much smaller number, say 256:

```
# echo 'hist:key=child_comm:val=hitcount:size=256' > \
       /sys/kernel/debug/tracing/events/sched/sched_process_fork/trigger

# cat /sys/kernel/debug/tracing/events/sched/sched_process_fork/hist
# trigger info: hist:keys=child_comm:vals=hitcount:sort=hitcount:size=256 [active]

{ child_comm: dconf worker                       } hitcount:          1
{ child_comm: ibus-daemon                        } hitcount:          1
{ child_comm: whoopsie                           } hitcount:          1
{ child_comm: smbd                               } hitcount:          1
{ child_comm: gdbus                              } hitcount:          1
{ child_comm: kthreadd                           } hitcount:          1
{ child_comm: dconf worker                       } hitcount:          1
{ child_comm: evolution-alarm                    } hitcount:          2
{ child_comm: Socket Thread                      } hitcount:          2
{ child_comm: postgres                           } hitcount:          2
{ child_comm: bash                               } hitcount:          3
{ child_comm: compiz                             } hitcount:          3
{ child_comm: evolution-sourc                    } hitcount:          4
{ child_comm: dhclient                           } hitcount:          4
{ child_comm: pool                               } hitcount:          5
{ child_comm: nm-dispatcher.a                    } hitcount:          8
{ child_comm: firefox                            } hitcount:          8
{ child_comm: dbus-daemon                        } hitcount:          8
{ child_comm: glib-pacrunner                     } hitcount:         10
{ child_comm: evolution                          } hitcount:         23

Totals:
    Hits: 89
    Entries: 20
    Dropped: 0
```

If we want to pause the hist trigger, we can simply append :pause to the command that started the trigger. Notice that the trigger info displays as [paused]:

```
# echo 'hist:key=child_comm:val=hitcount:size=256:pause' >> \
       /sys/kernel/debug/tracing/events/sched/sched_process_fork/trigger

# cat /sys/kernel/debug/tracing/events/sched/sched_process_fork/hist
# trigger info: hist:keys=child_comm:vals=hitcount:sort=hitcount:size=256 [paused]

{ child_comm: dconf worker                       } hitcount:          1
{ child_comm: kthreadd                           } hitcount:          1
{ child_comm: dconf worker                       } hitcount:          1
{ child_comm: gdbus                              } hitcount:          1
{ child_comm: ibus-daemon                        } hitcount:          1
{ child_comm: Socket Thread                      } hitcount:          2
{ child_comm: evolution-alarm                    } hitcount:          2
{ child_comm: smbd                               } hitcount:          2
{ child_comm: bash                               } hitcount:          3
{ child_comm: whoopsie                           } hitcount:          3
{ child_comm: compiz                             } hitcount:          3
{ child_comm: evolution-sourc                    } hitcount:          4
{ child_comm: pool                               } hitcount:          5
{ child_comm: postgres                           } hitcount:          6
{ child_comm: firefox                            } hitcount:          8
{ child_comm: dhclient                           } hitcount:         10
{ child_comm: emacs                              } hitcount:         12
{ child_comm: dbus-daemon                        } hitcount:         20
{ child_comm: nm-dispatcher.a                    } hitcount:         20
{ child_comm: evolution                          } hitcount:         35
{ child_comm: glib-pacrunner                     } hitcount:         59

Totals:
    Hits: 199
    Entries: 21
    Dropped: 0
```

To manually continue having the trigger aggregate events, append :cont instead. Notice that the trigger info displays as [active] again, and the data has changed:

```
# echo 'hist:key=child_comm:val=hitcount:size=256:cont' >> \
       /sys/kernel/debug/tracing/events/sched/sched_process_fork/trigger

# cat /sys/kernel/debug/tracing/events/sched/sched_process_fork/hist
# trigger info: hist:keys=child_comm:vals=hitcount:sort=hitcount:size=256 [active]

{ child_comm: dconf worker                       } hitcount:          1
{ child_comm: dconf worker                       } hitcount:          1
{ child_comm: kthreadd                           } hitcount:          1
{ child_comm: gdbus                              } hitcount:          1
{ child_comm: ibus-daemon                        } hitcount:          1
{ child_comm: Socket Thread                      } hitcount:          2
{ child_comm: evolution-alarm                    } hitcount:          2
{ child_comm: smbd                               } hitcount:          2
{ child_comm: whoopsie                           } hitcount:          3
{ child_comm: compiz                             } hitcount:          3
{ child_comm: evolution-sourc                    } hitcount:          4
{ child_comm: bash                               } hitcount:          5
{ child_comm: pool                               } hitcount:          5
{ child_comm: postgres                           } hitcount:          6
{ child_comm: firefox                            } hitcount:          8
{ child_comm: dhclient                           } hitcount:         11
{ child_comm: emacs                              } hitcount:         12
{ child_comm: dbus-daemon                        } hitcount:         22
{ child_comm: nm-dispatcher.a                    } hitcount:         22
{ child_comm: evolution                          } hitcount:         35
{ child_comm: glib-pacrunner                     } hitcount:         59

Totals:
    Hits: 206
    Entries: 21
```

```
    Dropped: 0
```

The previous example showed how to start and stop a hist trigger by appending 'pause' and 'continue' to the hist trigger command. A hist trigger can also be started in a paused state by initially starting the trigger with ':pause' appended. This allows you to start the trigger only when you're ready to start collecting data and not before. For example, you could start the trigger in a paused state, then unpause it and do something you want to measure, then pause the trigger again when done.

Of course, doing this manually can be difficult and error-prone, but it is possible to automatically start and stop a hist trigger based on some condition, via the enable_hist and disable_hist triggers.

For example, suppose we wanted to take a look at the relative weights in terms of skb length for each callpath that leads to a netif_receive_skb event when downloading a decent-sized file using wget.

First we set up an initially paused stacktrace trigger on the netif_receive_skb event:

```
# echo 'hist:key=stacktrace:vals=len:pause' > \
       /sys/kernel/debug/tracing/events/net/netif_receive_skb/trigger
```

Next, we set up an 'enable_hist' trigger on the sched_process_exec event, with an 'if filename==/usr/bin/wget' filter. The effect of this new trigger is that it will 'unpause' the hist trigger we just set up on netif_receive_skb if and only if it sees a sched_process_exec event with a filename of '/usr/bin/wget'. When that happens, all netif_receive_skb events are aggregated into a hash table keyed on stacktrace:

```
# echo 'enable_hist:net:netif_receive_skb if filename==/usr/bin/wget' > \
       /sys/kernel/debug/tracing/events/sched/sched_process_exec/trigger
```

The aggregation continues until the netif_receive_skb is paused again, which is what the following disable_hist event does by creating a similar setup on the sched_process_exit event, using the filter 'comm==wget':

```
# echo 'disable_hist:net:netif_receive_skb if comm==wget' > \
       /sys/kernel/debug/tracing/events/sched/sched_process_exit/trigger
```

Whenever a process exits and the comm field of the disable_hist trigger filter matches 'comm==wget', the netif_receive_skb hist trigger is disabled.

The overall effect is that netif_receive_skb events are aggregated into the hash table for only the duration of the wget. Executing a wget command and then listing the 'hist' file will display the output generated by the wget command:

```
$ wget https://www.kernel.org/pub/linux/kernel/v3.x/patch-3.19.xz

# cat /sys/kernel/debug/tracing/events/net/netif_receive_skb/hist
# trigger info: hist:keys=stacktrace:vals=len:sort=hitcount:size=2048 [paused]

{ stacktrace:
     __netif_receive_skb_core+0x46d/0x990
     __netif_receive_skb+0x18/0x60
     netif_receive_skb_internal+0x23/0x90
     napi_gro_receive+0xc8/0x100
     ieee80211_deliver_skb+0xd6/0x270 [mac80211]
     ieee80211_rx_handlers+0xccf/0x22f0 [mac80211]
     ieee80211_prepare_and_rx_handle+0x4e7/0xc40 [mac80211]
     ieee80211_rx+0x31d/0x900 [mac80211]
     iwlagn_rx_reply_rx+0x3db/0x6f0 [iwldvm]
     iwl_rx_dispatch+0x8e/0xf0 [iwldvm]
     iwl_pcie_irq_handler+0xe3c/0x12f0 [iwlwifi]
     irq_thread_fn+0x20/0x50
     irq_thread+0x11f/0x150
     kthread+0xd2/0xf0
     ret_from_fork+0x42/0x70
} hitcount:         85  len:      28884
{ stacktrace:
     __netif_receive_skb_core+0x46d/0x990
     __netif_receive_skb+0x18/0x60
     netif_receive_skb_internal+0x23/0x90
     napi_gro_complete+0xa4/0xe0
     dev_gro_receive+0x23a/0x360
     napi_gro_receive+0x30/0x100
     ieee80211_deliver_skb+0xd6/0x270 [mac80211]
     ieee80211_rx_handlers+0xccf/0x22f0 [mac80211]
     ieee80211_prepare_and_rx_handle+0x4e7/0xc40 [mac80211]
     ieee80211_rx+0x31d/0x900 [mac80211]
     iwlagn_rx_reply_rx+0x3db/0x6f0 [iwldvm]
     iwl_rx_dispatch+0x8e/0xf0 [iwldvm]
     iwl_pcie_irq_handler+0xe3c/0x12f0 [iwlwifi]
     irq_thread_fn+0x20/0x50
     irq_thread+0x11f/0x150
     kthread+0xd2/0xf0
} hitcount:         98  len:     664329
{ stacktrace:
     __netif_receive_skb_core+0x46d/0x990
     __netif_receive_skb+0x18/0x60
     process_backlog+0xa8/0x150
     net_rx_action+0x15d/0x340
     __do_softirq+0x114/0x2c0
     do_softirq_own_stack+0x1c/0x30
     do_softirq+0x65/0x70
     __local_bh_enable_ip+0xb5/0xc0
     ip_finish_output+0x1f4/0x840
     ip_output+0x6b/0xc0
     ip_local_out_sk+0x31/0x40
     ip_send_skb+0x1a/0x50
     udp_send_skb+0x173/0x2a0
     udp_sendmsg+0x2bf/0x9f0
     inet_sendmsg+0x64/0xa0
     sock_sendmsg+0x3d/0x50
} hitcount:        115  len:      13030
{ stacktrace:
     __netif_receive_skb_core+0x46d/0x990
     __netif_receive_skb+0x18/0x60
     netif_receive_skb_internal+0x23/0x90
     napi_gro_complete+0xa4/0xe0
     napi_gro_flush+0x6d/0x90
     iwl_pcie_irq_handler+0x92a/0x12f0 [iwlwifi]
     irq_thread_fn+0x20/0x50
     irq_thread+0x11f/0x150
     kthread+0xd2/0xf0
     ret_from_fork+0x42/0x70
} hitcount:        934  len:    5512212

Totals:
    Hits: 1232
    Entries: 4
    Dropped: 0
```

The above shows all the netif_receive_skb callpaths and their total lengths for the duration of the wget command.

The 'clear' hist trigger param can be used to clear the hash table. Suppose we wanted to try another run of the previous example but this time also wanted to see the complete list of events that went into the histogram. In order to avoid having to set everything up again, we can just clear the histogram first:

```
# echo 'hist:key=stacktrace:vals=len:clear' >> \
       /sys/kernel/debug/tracing/events/net/netif_receive_skb/trigger
```

Just to verify that it is in fact cleared, here's what we now see in the hist file:

```
# cat /sys/kernel/debug/tracing/events/net/netif_receive_skb/hist
# trigger info: hist:keys=stacktrace:vals=len:sort=hitcount:size=2048 [paused]
```

```
Totals:
    Hits: 0
    Entries: 0
    Dropped: 0
```

Since we want to see the detailed list of every netif_receive_skb event occurring during the new run, which are in fact the same events being aggregated into the hash table, we add some additional 'enable_event' events to the triggering sched_process_exec and sched_process_exit events as such:

```
# echo 'enable_event:net:netif_receive_skb if filename==/usr/bin/wget' > \
       /sys/kernel/debug/tracing/events/sched/sched_process_exec/trigger

# echo 'disable_event:net:netif_receive_skb if comm==wget' > \
       /sys/kernel/debug/tracing/events/sched/sched_process_exit/trigger
```

If you read the trigger files for the sched_process_exec and sched_process_exit triggers, you should see two triggers for each: one enabling/disabling the hist aggregation and the other enabling/disabling the logging of events:

```
# cat /sys/kernel/debug/tracing/events/sched/sched_process_exec/trigger
enable_event:net:netif_receive_skb:unlimited if filename==/usr/bin/wget
enable_hist:net:netif_receive_skb:unlimited if filename==/usr/bin/wget

# cat /sys/kernel/debug/tracing/events/sched/sched_process_exit/trigger
enable_event:net:netif_receive_skb:unlimited if comm==wget
disable_hist:net:netif_receive_skb:unlimited if comm==wget
```

In other words, whenever either of the sched_process_exec or sched_process_exit events is hit and matches 'wget', it enables or disables both the histogram and the event log, and what you end up with is a hash table and set of events just covering the specified duration. Run the wget command again:

```
$ wget https://www.kernel.org/pub/linux/kernel/v3.x/patch-3.19.xz
```

Displaying the 'hist' file should show something similar to what you saw in the last run, but this time you should also see the individual events in the trace file:

```
# cat /sys/kernel/debug/tracing/trace

# tracer: nop
#
# entries-in-buffer/entries-written: 183/1426   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
            wget-15108 [000] ..s1 31769.606929: netif_receive_skb: dev=lo skbaddr=ffff88009c353100 len=60
            wget-15108 [000] ..s1 31769.606999: netif_receive_skb: dev=lo skbaddr=ffff88009c353200 len=60
         dnsmasq-1382  [000] ..s1 31769.677652: netif_receive_skb: dev=lo skbaddr=ffff88009c352b00 len=130
         dnsmasq-1382  [000] ..s1 31769.685917: netif_receive_skb: dev=lo skbaddr=ffff88009c352200 len=138
##### CPU 2 buffer started ####
  irq/29-iwlwifi-559   [002] ..s. 31772.031529: netif_receive_skb: dev=wlan0 skbaddr=ffff88009d433d00 len=2948
  irq/29-iwlwifi-559   [002] ..s. 31772.031572: netif_receive_skb: dev=wlan0 skbaddr=ffff88009d432200 len=1500
  irq/29-iwlwifi-559   [002] ..s. 31772.032196: netif_receive_skb: dev=wlan0 skbaddr=ffff88009d433100 len=2948
  irq/29-iwlwifi-559   [002] ..s. 31772.032761: netif_receive_skb: dev=wlan0 skbaddr=ffff88009d433000 len=2948
  irq/29-iwlwifi-559   [002] ..s. 31772.033220: netif_receive_skb: dev=wlan0 skbaddr=ffff88009d432e00 len=1500
.
.
.
```

The following example demonstrates how multiple hist triggers can be attached to a given event. This capability can be useful for creating a set of different summaries derived from the same set of events, or for comparing the effects of different filters, among other things:

```
# echo 'hist:keys=skbaddr.hex:vals=len if len < 0' >> \
       /sys/kernel/debug/tracing/events/net/netif_receive_skb/trigger
# echo 'hist:keys=skbaddr.hex:vals=len if len > 4096' >> \
       /sys/kernel/debug/tracing/events/net/netif_receive_skb/trigger
# echo 'hist:keys=skbaddr.hex:vals=len if len == 256' >> \
       /sys/kernel/debug/tracing/events/net/netif_receive_skb/trigger
# echo 'hist:keys=skbaddr.hex:vals=len' >> \
       /sys/kernel/debug/tracing/events/net/netif_receive_skb/trigger
# echo 'hist:keys=len:vals=common_preempt_count' >> \
       /sys/kernel/debug/tracing/events/net/netif_receive_skb/trigger
```

The above set of commands create four triggers differing only in their filters, along with a completely different though fairly nonsensical trigger. Note that in order to append multiple hist triggers to the same file, you should use the '>>' operator to append them ('>' will also add the new hist trigger, but will remove any existing hist triggers beforehand).

Displaying the contents of the 'hist' file for the event shows the contents of all five histograms:

```
# cat /sys/kernel/debug/tracing/events/net/netif_receive_skb/hist

# event histogram
#
# trigger info: hist:keys=len:vals=hitcount,common_preempt_count:sort=hitcount:size=2048 [active]
#

{ len:        176 } hitcount:          1  common_preempt_count:          0
{ len:        223 } hitcount:          1  common_preempt_count:          0
{ len:       4854 } hitcount:          1  common_preempt_count:          0
{ len:        395 } hitcount:          1  common_preempt_count:          0
{ len:        177 } hitcount:          1  common_preempt_count:          0
{ len:        446 } hitcount:          1  common_preempt_count:          0
{ len:       1601 } hitcount:          1  common_preempt_count:          0
.
.
.
{ len:       1280 } hitcount:         66  common_preempt_count:          0
{ len:        116 } hitcount:         81  common_preempt_count:         40
{ len:        708 } hitcount:        112  common_preempt_count:          0
{ len:         46 } hitcount:        221  common_preempt_count:          0
{ len:       1264 } hitcount:        458  common_preempt_count:          0

Totals:
    Hits: 1428
    Entries: 147
    Dropped: 0


# event histogram
#
# trigger info: hist:keys=skbaddr.hex:vals=hitcount,len:sort=hitcount:size=2048 [active]
#

{ skbaddr: ffff8800baee5e00 } hitcount:          1  len:        130
{ skbaddr: ffff88005f3d5600 } hitcount:          1  len:       1280
{ skbaddr: ffff88005f3d4900 } hitcount:          1  len:       1280
{ skbaddr: ffff88009fed6300 } hitcount:          1  len:        115
{ skbaddr: ffff88009fe0ad00 } hitcount:          1  len:        115
{ skbaddr: ffff88008cdb1900 } hitcount:          1  len:         46
{ skbaddr: ffff880064b5ef00 } hitcount:          1  len:        118
{ skbaddr: ffff880044e3c700 } hitcount:          1  len:         60
{ skbaddr: ffff880100065900 } hitcount:          1  len:         46
{ skbaddr: ffff8800d46bd500 } hitcount:          1  len:        116
{ skbaddr: ffff88005f3d5f00 } hitcount:          1  len:       1280
{ skbaddr: ffff880100064700 } hitcount:          1  len:        365
```

```
{ skbaddr: ffff8800badb6f00 } hitcount:          1  len:         60
.
.
.
{ skbaddr: ffff88009fe0be00 } hitcount:         27  len:      24677
{ skbaddr: ffff88009fe0a400 } hitcount:         27  len:      23052
{ skbaddr: ffff88009fe0b700 } hitcount:         31  len:      25589
{ skbaddr: ffff88009fe0b600 } hitcount:         32  len:      27326
{ skbaddr: ffff88006a462800 } hitcount:         68  len:      71678
{ skbaddr: ffff88006a463700 } hitcount:         70  len:      72678
{ skbaddr: ffff88006a462b00 } hitcount:         71  len:      77589
{ skbaddr: ffff88006a463600 } hitcount:         73  len:      71307
{ skbaddr: ffff88006a462200 } hitcount:         81  len:      81032

Totals:
    Hits: 1451
    Entries: 318
    Dropped: 0


# event histogram
#
# trigger info: hist:keys=skbaddr.hex:vals=hitcount,len:sort=hitcount:size=2048 if len == 256 [active]
#


Totals:
    Hits: 0
    Entries: 0
    Dropped: 0


# event histogram
#
# trigger info: hist:keys=skbaddr.hex:vals=hitcount,len:sort=hitcount:size=2048 if len > 4096 [active]
#

{ skbaddr: ffff88009fd2c300 } hitcount:          1  len:       7212
{ skbaddr: ffff8800d2bcce00 } hitcount:          1  len:       7212
{ skbaddr: ffff8800d2bcd700 } hitcount:          1  len:       7212
{ skbaddr: ffff8800d2bcda00 } hitcount:          1  len:      21492
{ skbaddr: ffff8800ae2e2d00 } hitcount:          1  len:       7212
{ skbaddr: ffff8800d2bcdb00 } hitcount:          1  len:       7212
{ skbaddr: ffff88006a4df500 } hitcount:          1  len:       4854
{ skbaddr: ffff88008ce47b00 } hitcount:          1  len:      18636
{ skbaddr: ffff8800ae2e2200 } hitcount:          1  len:      12924
{ skbaddr: ffff88005f3e1000 } hitcount:          1  len:       4356
{ skbaddr: ffff8800d2bcdc00 } hitcount:          2  len:      24420
{ skbaddr: ffff8800d2bcc200 } hitcount:          2  len:      12996

Totals:
    Hits: 14
    Entries: 12
    Dropped: 0


# event histogram
#
# trigger info: hist:keys=skbaddr.hex:vals=hitcount,len:sort=hitcount:size=2048 if len < 0 [active]
#


Totals:
    Hits: 0
    Entries: 0
    Dropped: 0
```

Named triggers can be used to have triggers share a common set of histogram data. This capability is mostly useful for combining the output of events generated by tracepoints contained inside inline functions, but names can be used in a hist trigger on any event. For example, these two triggers when hit will update the same 'len' field in the shared 'foo' histogram data:

```
# echo 'hist:name=foo:keys=skbaddr.hex:vals=len' > \
       /sys/kernel/debug/tracing/events/net/netif_receive_skb/trigger
# echo 'hist:name=foo:keys=skbaddr.hex:vals=len' > \
       /sys/kernel/debug/tracing/events/net/netif_rx/trigger
```

You can see that they're updating common histogram data by reading each event's hist files at the same time:

```
# cat /sys/kernel/debug/tracing/events/net/netif_receive_skb/hist;
  cat /sys/kernel/debug/tracing/events/net/netif_rx/hist

# event histogram
#
# trigger info: hist:name=foo:keys=skbaddr.hex:vals=hitcount,len:sort=hitcount:size=2048 [active]
#

{ skbaddr: ffff88000ad53500 } hitcount:          1  len:         46
{ skbaddr: ffff8800af5a1500 } hitcount:          1  len:         76
{ skbaddr: ffff8800d62a1900 } hitcount:          1  len:         46
{ skbaddr: ffff8800d2bccb00 } hitcount:          1  len:        468
{ skbaddr: ffff8800d3c69900 } hitcount:          1  len:         46
{ skbaddr: ffff88009ff09100 } hitcount:          1  len:         52
{ skbaddr: ffff88010f13ab00 } hitcount:          1  len:        168
{ skbaddr: ffff88006a54f400 } hitcount:          1  len:         46
{ skbaddr: ffff8800d2bcc500 } hitcount:          1  len:        260
{ skbaddr: ffff880064505000 } hitcount:          1  len:         46
{ skbaddr: ffff8800baf24e00 } hitcount:          1  len:         32
{ skbaddr: ffff88009fe0ad00 } hitcount:          1  len:         46
{ skbaddr: ffff8800d3edff00 } hitcount:          1  len:         44
{ skbaddr: ffff88009fe0b400 } hitcount:          1  len:        168
{ skbaddr: ffff8800a1c55a00 } hitcount:          1  len:         40
{ skbaddr: ffff8800d2bcd100 } hitcount:          1  len:         40
{ skbaddr: ffff880064505f00 } hitcount:          1  len:        174
{ skbaddr: ffff8800a8bff200 } hitcount:          1  len:        160
{ skbaddr: ffff880044e3cc00 } hitcount:          1  len:         76
{ skbaddr: ffff8800a8bfe700 } hitcount:          1  len:         46
{ skbaddr: ffff8800d2bcdc00 } hitcount:          1  len:         32
{ skbaddr: ffff8800a1f64800 } hitcount:          1  len:         46
{ skbaddr: ffff8800d2bcde00 } hitcount:          1  len:        988
{ skbaddr: ffff88006a5dea00 } hitcount:          1  len:         46
{ skbaddr: ffff88002e37a200 } hitcount:          1  len:         44
{ skbaddr: ffff8800a1f32c00 } hitcount:          2  len:        676
{ skbaddr: ffff88000ad52600 } hitcount:          2  len:        107
{ skbaddr: ffff8800a1f91e00 } hitcount:          2  len:         92
{ skbaddr: ffff8800af5a0200 } hitcount:          2  len:        142
{ skbaddr: ffff8800d2bcc600 } hitcount:          2  len:        220
{ skbaddr: ffff8800ba36f500 } hitcount:          2  len:         92
{ skbaddr: ffff8800d021f800 } hitcount:          2  len:         92
{ skbaddr: ffff8800a1f33600 } hitcount:          2  len:        675
{ skbaddr: ffff8800a8bfff00 } hitcount:          3  len:        138
{ skbaddr: ffff8800d62a1300 } hitcount:          3  len:        138
{ skbaddr: ffff88002e37a100 } hitcount:          4  len:        184
{ skbaddr: ffff880064504400 } hitcount:          4  len:        184
{ skbaddr: ffff8800a8bfec00 } hitcount:          4  len:        184
{ skbaddr: ffff88000ad53700 } hitcount:          5  len:        230
{ skbaddr: ffff8800d2bcdb00 } hitcount:          5  len:        196
{ skbaddr: ffff8800a1f90000 } hitcount:          6  len:        276
{ skbaddr: ffff88006a54f900 } hitcount:          6  len:        276
```

```
Totals:
    Hits: 81
    Entries: 42
    Dropped: 0
# event histogram
#
# trigger info: hist:name=foo:keys=skbaddr.hex:vals=hitcount,len:sort=hitcount:size=2048 [active]
#

{ skbaddr: ffff88000ad53500 } hitcount:          1  len:         46
{ skbaddr: ffff8800af5a1500 } hitcount:          1  len:         76
{ skbaddr: ffff8800d62a1900 } hitcount:          1  len:         46
{ skbaddr: ffff8800d2bccb00 } hitcount:          1  len:        468
{ skbaddr: ffff8800d3c69900 } hitcount:          1  len:         46
{ skbaddr: ffff88009ff09100 } hitcount:          1  len:         52
{ skbaddr: ffff88010f13ab00 } hitcount:          1  len:        168
{ skbaddr: ffff88006a54f400 } hitcount:          1  len:         46
{ skbaddr: ffff8800d2bcc500 } hitcount:          1  len:        260
{ skbaddr: ffff880064505000 } hitcount:          1  len:         46
{ skbaddr: ffff8800baf24e00 } hitcount:          1  len:         32
{ skbaddr: ffff88009fe0ad00 } hitcount:          1  len:         46
{ skbaddr: ffff8800d3edff00 } hitcount:          1  len:         44
{ skbaddr: ffff88009fe0b400 } hitcount:          1  len:        168
{ skbaddr: ffff8800a1c55a00 } hitcount:          1  len:         40
{ skbaddr: ffff8800d2bcd100 } hitcount:          1  len:         40
{ skbaddr: ffff880064505f00 } hitcount:          1  len:        174
{ skbaddr: ffff8800a8bff200 } hitcount:          1  len:        160
{ skbaddr: ffff880044e3cc00 } hitcount:          1  len:         76
{ skbaddr: ffff8800a8bfe700 } hitcount:          1  len:         46
{ skbaddr: ffff8800d2bcdc00 } hitcount:          1  len:         32
{ skbaddr: ffff8800a1f64800 } hitcount:          1  len:         46
{ skbaddr: ffff8800d2bcde00 } hitcount:          1  len:        988
{ skbaddr: ffff88006a5dea00 } hitcount:          1  len:         46
{ skbaddr: ffff88002e37a200 } hitcount:          1  len:         44
{ skbaddr: ffff8800a1f32c00 } hitcount:          2  len:        676
{ skbaddr: ffff88000ad52600 } hitcount:          2  len:        107
{ skbaddr: ffff8800a1f91e00 } hitcount:          2  len:         92
{ skbaddr: ffff8800af5a0200 } hitcount:          2  len:        142
{ skbaddr: ffff8800d2bcc600 } hitcount:          2  len:        220
{ skbaddr: ffff8800ba36f500 } hitcount:          2  len:         92
{ skbaddr: ffff8800d021f800 } hitcount:          2  len:         92
{ skbaddr: ffff8800a1f33600 } hitcount:          2  len:        675
{ skbaddr: ffff8800a8bfff00 } hitcount:          3  len:        138
{ skbaddr: ffff8800d62a1300 } hitcount:          3  len:        138
{ skbaddr: ffff88002e37a100 } hitcount:          4  len:        184
{ skbaddr: ffff880064504400 } hitcount:          4  len:        184
{ skbaddr: ffff8800a8bfec00 } hitcount:          4  len:        184
{ skbaddr: ffff88000ad53700 } hitcount:          5  len:        230
{ skbaddr: ffff8800d2bcdb00 } hitcount:          5  len:        196
{ skbaddr: ffff8800a1f90000 } hitcount:          6  len:        276
{ skbaddr: ffff88006a54f900 } hitcount:          6  len:        276

Totals:
    Hits: 81
    Entries: 42
    Dropped: 0
```

And here's an example that shows how to combine histogram data from any two events even if they don't share any 'compatible' fields other than 'hitcount' and 'stacktrace'. These commands create a couple of triggers named 'bar' using those fields:

```
# echo 'hist:name=bar:key=stacktrace:val=hitcount' > \
       /sys/kernel/debug/tracing/events/sched/sched_process_fork/trigger
# echo 'hist:name=bar:key=stacktrace:val=hitcount' > \
      /sys/kernel/debug/tracing/events/net/netif_rx/trigger
```

And displaying the output of either shows some interesting if somewhat confusing output:

```
# cat /sys/kernel/debug/tracing/events/sched/sched_process_fork/hist
# cat /sys/kernel/debug/tracing/events/net/netif_rx/hist

# event histogram
#
# trigger info: hist:name=bar:keys=stacktrace:vals=hitcount:sort=hitcount:size=2048 [active]
#

{ stacktrace:
         kernel_clone+0x18e/0x330
         kernel_thread+0x29/0x30
         kthreadd+0x154/0x1b0
         ret_from_fork+0x3f/0x70
} hitcount:          1
{ stacktrace:
         netif_rx_internal+0xb2/0xd0
         netif_rx_ni+0x20/0x70
         dev_loopback_xmit+0xaa/0xd0
         ip_mc_output+0x126/0x240
         ip_local_out_sk+0x31/0x40
         igmp_send_report+0x1e9/0x230
         igmp_timer_expire+0xe9/0x120
         call_timer_fn+0x39/0xf0
         run_timer_softirq+0x1e1/0x290
         __do_softirq+0xfd/0x290
         irq_exit+0x98/0xb0
         smp_apic_timer_interrupt+0x4a/0x60
         apic_timer_interrupt+0x6d/0x80
         cpuidle_enter+0x17/0x20
         call_cpuidle+0x3b/0x60
         cpu_startup_entry+0x22d/0x310
} hitcount:          1
{ stacktrace:
         netif_rx_internal+0xb2/0xd0
         netif_rx_ni+0x20/0x70
         dev_loopback_xmit+0xaa/0xd0
         ip_mc_output+0x17f/0x240
         ip_local_out_sk+0x31/0x40
         ip_send_skb+0x1a/0x50
         udp_send_skb+0x13e/0x270
         udp_sendmsg+0x2bf/0x980
         inet_sendmsg+0x67/0xa0
         sock_sendmsg+0x38/0x50
         SYSC_sendto+0xef/0x170
         SyS_sendto+0xe/0x10
         entry_SYSCALL_64_fastpath+0x12/0x6a
} hitcount:          2
{ stacktrace:
         netif_rx_internal+0xb2/0xd0
         netif_rx+0x1c/0x60
         loopback_xmit+0x6c/0xb0
         dev_hard_start_xmit+0x219/0x3a0
         __dev_queue_xmit+0x415/0x4f0
         dev_queue_xmit_sk+0x13/0x20
         ip_finish_output2+0x237/0x340
         ip_finish_output+0x113/0x1d0
         ip_output+0x66/0xc0
         ip_local_out_sk+0x31/0x40
         ip_send_skb+0x1a/0x50
         udp_send_skb+0x16d/0x270
         udp_sendmsg+0x2bf/0x980
         inet_sendmsg+0x67/0xa0
```

```
        sock_sendmsg+0x38/0x50
        ___sys_sendmsg+0x14e/0x270
} hitcount:         76
{ stacktrace:
        netif_rx_internal+0xb2/0xd0
        netif_rx+0x1c/0x60
        loopback_xmit+0x6c/0xb0
        dev_hard_start_xmit+0x219/0x3a0
        __dev_queue_xmit+0x415/0x4f0
        dev_queue_xmit_sk+0x13/0x20
        ip_finish_output2+0x237/0x340
        ip_finish_output+0x113/0x1d0
        ip_output+0x66/0xc0
        ip_local_out_sk+0x31/0x40
        ip_send_skb+0x1a/0x50
        udp_send_skb+0x16d/0x270
        udp_sendmsg+0x2bf/0x980
        inet_sendmsg+0x67/0xa0
        sock_sendmsg+0x38/0x50
        ___sys_sendmsg+0x269/0x270
} hitcount:         77
{ stacktrace:
        netif_rx_internal+0xb2/0xd0
        netif_rx+0x1c/0x60
        loopback_xmit+0x6c/0xb0
        dev_hard_start_xmit+0x219/0x3a0
        __dev_queue_xmit+0x415/0x4f0
        dev_queue_xmit_sk+0x13/0x20
        ip_finish_output2+0x237/0x340
        ip_finish_output+0x113/0x1d0
        ip_output+0x66/0xc0
        ip_local_out_sk+0x31/0x40
        ip_send_skb+0x1a/0x50
        udp_send_skb+0x16d/0x270
        udp_sendmsg+0x2bf/0x980
        inet_sendmsg+0x67/0xa0
        sock_sendmsg+0x38/0x50
        SYSC_sendto+0xef/0x170
} hitcount:         88
{ stacktrace:
        kernel_clone+0x18e/0x330
        SyS_clone+0x19/0x20
        entry_SYSCALL_64_fastpath+0x12/0x6a
} hitcount:        244

Totals:
    Hits: 489
    Entries: 7
    Dropped: 0
```

2.2 Inter-event hist triggers

Inter-event hist triggers are hist triggers that combine values from one or more other events and create a histogram using that data. Data from an inter-event histogram can in turn become the source for further combined histograms, thus providing a chain of related histograms, which is important for some applications.

The most important example of an inter-event quantity that can be used in this manner is latency, which is simply a difference in timestamps between two events. Although latency is the most important inter-event quantity, note that because the support is completely general across the trace event subsystem, any event field can be used in an inter-event quantity.

An example of a histogram that combines data from other histograms into a useful chain would be a 'wakeupswitch latency' histogram that combines a 'wakeup latency' histogram and a 'switch latency' histogram.

Normally, a hist trigger specification consists of a (possibly compound) key along with one or more numeric values, which are continually updated sums associated with that key. A histogram specification in this case consists of individual key and value specifications that refer to trace event fields associated with a single event type.

The inter-event hist trigger extension allows fields from multiple events to be referenced and combined into a multi-event histogram specification. In support of this overall goal, a few enabling features have been added to the hist trigger support:

- In order to compute an inter-event quantity, a value from one event needs to saved and then referenced from another event. This requires the introduction of support for histogram 'variables'.
- The computation of inter-event quantities and their combination require some minimal amount of support for applying simple expressions to variables (+ and -).
- A histogram consisting of inter-event quantities isn't logically a histogram on either event (so having the 'hist' file for either event host the histogram output doesn't really make sense). To address the idea that the histogram is associated with a combination of events, support is added allowing the creation of 'synthetic' events that are events derived from other events. These synthetic events are full-fledged events just like any other and can be used as such, as for instance to create the 'combination' histograms mentioned previously.
- A set of 'actions' can be associated with histogram entries - these can be used to generate the previously mentioned synthetic events, but can also be used for other purposes, such as for example saving context when a 'max' latency has been hit.
- Trace events don't have a 'timestamp' associated with them, but there is an implicit timestamp saved along with an event in the underlying ftrace ring buffer. This timestamp is now exposed as a a synthetic field named 'common_timestamp' which can be used in histograms as if it were any other event field; it isn't an actual field in the trace format but rather is a synthesized value that nonetheless can be used as if it were an actual field. By default it is in units of nanoseconds; appending '.usecs' to a common_timestamp field changes the units to microseconds.

A note on inter-event timestamps: If common_timestamp is used in a histogram, the trace buffer is automatically switched over to using absolute timestamps and the "global" trace clock, in order to avoid bogus timestamp differences with other clocks that aren't coherent across CPUs. This can be overridden by specifying one of the other trace clocks instead, using the "clock=XXX" hist trigger attribute, where XXX is any of the clocks listed in the tracing/trace_clock pseudo-file.

These features are described in more detail in the following sections.

2.2.1 Histogram Variables

Variables are simply named locations used for saving and retrieving values between matching events. A 'matching' event is defined as an event that has a matching key - if a variable is saved for a histogram entry corresponding to that key, any subsequent event with a matching key can access that variable.

A variable's value is normally available to any subsequent event until it is set to something else by a subsequent event. The one exception to that rule is that any variable used in an expression is essentially 'read-once' - once it's used by an expression in a subsequent event, it's reset to its 'unset' state, which means it can't be used again unless it's set again. This ensures not only that an event doesn't use an uninitialized variable in a calculation, but that that variable is used only once and not for any unrelated subsequent match.

The basic syntax for saving a variable is to simply prefix a unique variable name not corresponding to any keyword along with an '=' sign to any event field.

Either keys or values can be saved and retrieved in this way. This creates a variable named 'ts0' for a histogram entry with the key 'next_pid':

```
# echo 'hist:keys=next_pid:vals=$ts0:ts0=common_timestamp ... >> \
      event/trigger
```

The ts0 variable can be accessed by any subsequent event having the same pid as 'next_pid'.

Variable references are formed by prepending the variable name with the '$' sign. Thus for example, the ts0 variable above would be referenced as '$ts0' in expressions.

Because 'vals=' is used, the common_timestamp variable value above will also be summed as a normal histogram value would

(though for a timestamp it makes little sense).

The below shows that a key value can also be saved in the same way:

```
# echo 'hist:timer_pid=common_pid:key=timer_pid ...' >> event/trigger
```

If a variable isn't a key variable or prefixed with 'vals=', the associated event field will be saved in a variable but won't be summed as a value:

```
# echo 'hist:keys=next_pid:ts1=common_timestamp ...' >> event/trigger
```

Multiple variables can be assigned at the same time. The below would result in both ts0 and b being created as variables, with both common_timestamp and field1 additionally being summed as values:

```
# echo 'hist:keys=pid:vals=$ts0,$b:ts0=common_timestamp,b=field1 ...' >> \
      event/trigger
```

Note that variable assignments can appear either preceding or following their use. The command below behaves identically to the command above:

```
# echo 'hist:keys=pid:ts0=common_timestamp,b=field1:vals=$ts0,$b ...' >> \
      event/trigger
```

Any number of variables not bound to a 'vals=' prefix can also be assigned by simply separating them with colons. Below is the same thing but without the values being summed in the histogram:

```
# echo 'hist:keys=pid:ts0=common_timestamp:b=field1 ...' >> event/trigger
```

Variables set as above can be referenced and used in expressions on another event.

For example, here's how a latency can be calculated:

```
# echo 'hist:keys=pid,prio:ts0=common_timestamp ...' >> event1/trigger
# echo 'hist:keys=next_pid:wakeup_lat=common_timestamp-$ts0 ...' >> event2/trigger
```

In the first line above, the event's timestamp is saved into the variable ts0. In the next line, ts0 is subtracted from the second event's timestamp to produce the latency, which is then assigned into yet another variable, 'wakeup_lat'. The hist trigger below in turn makes use of the wakeup_lat variable to compute a combined latency using the same key and variable from yet another event:

```
# echo 'hist:key=pid:wakeupswitch_lat=$wakeup_lat+$switchtime_lat ...' >> event3/trigger
```

Expressions support the use of addition, subtraction, multiplication and division operators (+-*/).

Note if division by zero cannot be detected at parse time (i.e. the divisor is not a constant), the result will be -1.

Numeric constants can also be used directly in an expression:

```
# echo 'hist:keys=next_pid:timestamp_secs=common_timestamp/1000000 ...' >> event/trigger
```

or assigned to a variable and referenced in a subsequent expression:

```
# echo 'hist:keys=next_pid:us_per_sec=1000000 ...' >> event/trigger
# echo 'hist:keys=next_pid:timestamp_secs=common_timestamp/$us_per_sec ...' >> event/trigger
```

2.2.2 Synthetic Events

Synthetic events are user-defined events generated from hist trigger variables or fields associated with one or more other events. Their purpose is to provide a mechanism for displaying data spanning multiple events consistent with the existing and already familiar usage for normal events.

To define a synthetic event, the user writes a simple specification consisting of the name of the new event along with one or more variables and their types, which can be any valid field type, separated by semicolons, to the tracing/synthetic_events file.

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.

A string field can be specified using either the static notation:

```
char name[32];
```

Or the dynamic:

```
char name[];
```

The size limit for either is 256.

For instance, the following creates a new event named 'wakeup_latency' with 3 fields: lat, pid, and prio. Each of those fields is simply a variable reference to a variable on another event:

```
# echo 'wakeup_latency \
        u64 lat; \
        pid_t pid; \
        int prio' >> \
        /sys/kernel/debug/tracing/synthetic_events
```

Reading the tracing/synthetic_events file lists all the currently defined synthetic events, in this case the event defined above:

```
# cat /sys/kernel/debug/tracing/synthetic_events
  wakeup_latency u64 lat; pid_t pid; int prio
```

An existing synthetic event definition can be removed by prepending the command that defined it with a '!':

```
# echo '!wakeup_latency u64 lat pid_t pid int prio' >> \
  /sys/kernel/debug/tracing/synthetic_events
```

At this point, there isn't yet an actual 'wakeup_latency' event instantiated in the event subsystem - for this to happen, a 'hist trigger action' needs to be instantiated and bound to actual fields and variables defined on other events (see Section 2.2.3 below on how that is done using hist trigger 'onmatch' action). Once that is done, the 'wakeup_latency' synthetic event instance is created.

The new event is created under the tracing/events/synthetic/ directory and looks and behaves just like any other event:

```
# ls /sys/kernel/debug/tracing/events/synthetic/wakeup_latency
      enable  filter  format  hist  id  trigger
```

A histogram can now be defined for the new synthetic event:

```
# echo 'hist:keys=pid,prio,lat.log2:sort=lat' >> \
      /sys/kernel/debug/tracing/events/synthetic/wakeup_latency/trigger
```

The above shows the latency "lat" in a power of 2 grouping.

Like any other event, once a histogram is enabled for the event, the output can be displayed by reading the event's 'hist' file.

```
# cat /sys/kernel/debug/tracing/events/synthetic/wakeup_latency/hist

# event histogram # # trigger info: hist:keys=pid,prio,lat.log2:vals=hitcount:sort=lat.log2:size=2048 [active] #

{ pid: 2035, prio: 9, lat: ~ 2^2 } hitcount: 43 { pid: 2034, prio: 9, lat: ~ 2^2 } hitcount: 60 { pid: 2029, prio: 9, lat: ~ 2^2 } hitcount: 965 { pid: 2034, prio: 120, lat: ~ 2^2 } hitcount: 9 { pid: 2033, prio: 120, lat: ~ 2^2 } hitcount: 5 { pid: 2030, prio: 9, lat: ~ 2^2 } hitcount: 335 { pid: 2030, prio: 120, lat: ~ 2^2 } hitcount: 10 { pid: 2032, prio: 120, lat: ~ 2^2 } hitcount: 1 { pid: 2035, prio: 120, lat: ~ 2^2 } hitcount: 2 { pid: 2031, prio: 9, lat: ~ 2^2 } hitcount: 176 { pid: 2028, prio: 120, lat: ~ 2^2 } hitcount: 15 { pid: 2033, prio: 9, lat: ~ 2^2 } hitcount: 91 { pid: 2032, prio: 9, lat: ~ 2^2 } hitcount: 125 { pid: 2029, prio: 120, lat: ~ 2^2 } hitcount: 4 { pid: 2031, prio: 120, lat: ~ 2^2 } hitcount: 3 { pid: 2029, prio: 120, lat: ~ 2^3 } hitcount: 2 { pid: 2035, prio: 9, lat: ~ 2^3 } hitcount: 41 { pid: 2030, prio: 120, lat: ~ 2^3 } hitcount: 1 { pid: 2032, prio: 9, lat: ~ 2^3 } hitcount: 32 { pid: 2031, prio: 9, lat: ~ 2^3 } hitcount: 44 { pid: 2034, prio: 9, lat: ~ 2^3 } hitcount: 40
```

{ pid: 2030, prio: 9, lat: ~ 2^3 } hitcount: 29 { pid: 2033, prio: 9, lat: ~ 2^3 } hitcount: 31 { pid: 2029, prio: 9, lat: ~ 2^3 } hitcount: 31 { pid: 2028, prio: 120, lat: ~ 2^3 } hitcount: 18 { pid: 2031, prio: 120, lat: ~ 2^3 } hitcount: 2 { pid: 2028, prio: 120, lat: ~ 2^4 } hitcount: 1 { pid: 2029, prio: 9, lat: ~ 2^4 } hitcount: 4 { pid: 2031, prio: 120, lat: ~ 2^7 } hitcount: 1 { pid: 2032, prio: 120, lat: ~ 2^7 } hitcount: 1

Totals:
Hits: 2122 Entries: 30 Dropped: 0

The latency values can also be grouped linearly by a given size with the ".buckets" modifier and specify a size (in this case groups of 10).

echo 'hist:keys=pid,prio,lat.buckets=10:sort=lat' >>
/sys/kernel/debug/tracing/events/synthetic/wakeup_latency/trigger

event histogram # # trigger info: hist:keys=pid,prio,lat.buckets=10:vals=hitcount:sort=lat.buckets=10:size=2048 [active]
#

{ pid: 2067, prio: 9, lat: ~ 0-9 } hitcount: 220 { pid: 2068, prio: 9, lat: ~ 0-9 } hitcount: 157 { pid: 2070, prio: 9, lat: ~ 0-9 } hitcount: 100 { pid: 2067, prio: 120, lat: ~ 0-9 } hitcount: 6 { pid: 2065, prio: 120, lat: ~ 0-9 } hitcount: 2 { pid: 2066, prio: 120, lat: ~ 0-9 } hitcount: 2 { pid: 2069, prio: 9, lat: ~ 0-9 } hitcount: 122 { pid: 2069, prio: 120, lat: ~ 0-9 } hitcount: 8 { pid: 2070, prio: 120, lat: ~ 0-9 } hitcount: 1 { pid: 2068, prio: 120, lat: ~ 0-9 } hitcount: 7 { pid: 2066, prio: 9, lat: ~ 0-9 } hitcount: 365 { pid: 2064, prio: 120, lat: ~ 0-9 } hitcount: 35 { pid: 2065, prio: 9, lat: ~ 0-9 } hitcount: 998 { pid: 2071, prio: 9, lat: ~ 0-9 } hitcount: 85 { pid: 2065, prio: 9, lat: ~ 10-19 } hitcount: 2 { pid: 2064, prio: 120, lat: ~ 10-19 } hitcount: 2

Totals:
Hits: 2112 Entries: 16 Dropped: 0

2.2.3 Hist trigger 'handlers' and 'actions'

A hist trigger 'action' is a function that's executed (in most cases conditionally) whenever a histogram entry is added or updated.

When a histogram entry is added or updated, a hist trigger 'handler' is what decides whether the corresponding action is actually invoked or not.

Hist trigger handlers and actions are paired together in the general form:

<handler>.<action>

To specify a handler.action pair for a given event, simply specify that handler.action pair between colons in the hist trigger specification.

In theory, any handler can be combined with any action, but in practice, not every handler.action combination is currently supported; if a given handler.action combination isn't supported, the hist trigger will fail with -EINVAL;

The default 'handler.action' if none is explicitly specified is as it always has been, to simply update the set of values associated with an entry. Some applications, however, may want to perform additional actions at that point, such as generate another event, or compare and save a maximum.

The supported handlers and actions are listed below, and each is described in more detail in the following paragraphs, in the context of descriptions of some common and useful handler.action combinations.

The available handlers are:

- onmatch(matching.event) - invoke action on any addition or update
- onmax(var) - invoke action if var exceeds current max
- onchange(var) - invoke action if var changes

The available actions are:

- trace(<synthetic_event_name>,param list) - generate synthetic event
- save(field,...) - save current event fields
- snapshot() - snapshot the trace buffer

The following commonly-used handler.action pairs are available:

- onmatch(matching.event).trace(<synthetic_event_name>,param list)

  The 'onmatch(matching.event).trace(<synthetic_event_name>,param list)' hist trigger action is invoked whenever an event matches and the histogram entry would be added or updated. It causes the named synthetic event to be generated with the values given in the 'param list'. The result is the generation of a synthetic event that consists of the values contained in those variables at the time the invoking event was hit. For example, if the synthetic event name is 'wakeup_latency', a wakeup_latency event is generated using onmatch(event).trace(wakeup_latency,arg1,arg2).

  There is also an equivalent alternative form available for generating synthetic events. In this form, the synthetic event name is used as if it were a function name. For example, using the 'wakeup_latency' synthetic event name again, the wakeup_latency event would be generated by invoking it as if it were a function call, with the event field values passed in as arguments: onmatch(event).wakeup_latency(arg1,arg2). The syntax for this form is:

  onmatch(matching.event).<synthetic_event_name>(param list)

  In either case, the 'param list' consists of one or more parameters which may be either variables or fields defined on either the 'matching.event' or the target event. The variables or fields specified in the param list may be either fully-qualified or unqualified. If a variable is specified as unqualified, it must be unique between the two events. A field name used as a param can be unqualified if it refers to the target event, but must be fully qualified if it refers to the matching event. A fully-qualified name is of the form 'system.event_name.$var_name' or 'system.event_name.field'.

  The 'matching.event' specification is simply the fully qualified event name of the event that matches the target event for the onmatch() functionality, in the form 'system.event_name'. Histogram keys of both events are compared to find if events match. In case multiple histogram keys are used, they all must match in the specified order.

  Finally, the number and type of variables/fields in the 'param list' must match the number and types of the fields in the synthetic event being generated.

  As an example the below defines a simple synthetic event and uses a variable defined on the sched_wakeup_new event as a parameter when invoking the synthetic event. Here we define the synthetic event:

  ```
  # echo 'wakeup_new_test pid_t pid' >> \
        /sys/kernel/debug/tracing/synthetic_events

  # cat /sys/kernel/debug/tracing/synthetic_events
       wakeup_new_test pid_t pid
  ```

  The following hist trigger both defines the missing testpid variable and specifies an onmatch() action that generates a wakeup_new_test synthetic event whenever a sched_wakeup_new event occurs, which because of the 'if comm == "cyclictest"' filter only happens when the executable is cyclictest:

  ```
  # echo 'hist:keys=$testpid:testpid=pid:onmatch(sched.sched_wakeup_new).\
         wakeup_new_test($testpid) if comm=="cyclictest"' >> \
         /sys/kernel/debug/tracing/events/sched/sched_wakeup_new/trigger
  ```

  Or, equivalently, using the 'trace' keyword syntax:

  # echo 'hist:keys=$testpid:testpid=pid:onmatch(sched.sched_wakeup_new).
  trace(wakeup_new_test,$testpid) if comm=="cyclictest" >>
  /sys/kernel/debug/tracing/events/sched/sched_wakeup_new/trigger

  Creating and displaying a histogram based on those events is now just a matter of using the fields and new synthetic event in the tracing/events/synthetic directory, as usual:

  ```
  # echo 'hist:keys=pid:sort=pid' >> \
  ```

```
/sys/kernel/debug/tracing/events/synthetic/wakeup_new_test/trigger
```

Running 'cyclictest' should cause wakeup_new events to generate wakeup_new_test synthetic events which should result in histogram output in the wakeup_new_test event's hist file:

```
# cat /sys/kernel/debug/tracing/events/synthetic/wakeup_new_test/hist
```

A more typical usage would be to use two events to calculate a latency. The following example uses a set of hist triggers to produce a 'wakeup_latency' histogram.

First, we define a 'wakeup_latency' synthetic event:

```
# echo 'wakeup_latency u64 lat; pid_t pid; int prio' >> \
        /sys/kernel/debug/tracing/synthetic_events
```

Next, we specify that whenever we see a sched_waking event for a cyclictest thread, save the timestamp in a 'ts0' variable:

```
# echo 'hist:keys=$saved_pid:saved_pid=pid:ts0=common_timestamp.usecs \
        if comm=="cyclictest"' >> \
        /sys/kernel/debug/tracing/events/sched/sched_waking/trigger
```

Then, when the corresponding thread is actually scheduled onto the CPU by a sched_switch event (saved_pid matches next_pid), calculate the latency and use that along with another variable and an event field to generate a wakeup_latency synthetic event:

```
# echo 'hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0:\
        onmatch(sched.sched_waking).wakeup_latency($wakeup_lat,\
                $saved_pid,next_prio) if next_comm=="cyclictest"' >> \
        /sys/kernel/debug/tracing/events/sched/sched_switch/trigger
```

We also need to create a histogram on the wakeup_latency synthetic event in order to aggregate the generated synthetic event data:

```
# echo 'hist:keys=pid,prio,lat:sort=pid,lat' >> \
        /sys/kernel/debug/tracing/events/synthetic/wakeup_latency/trigger
```

Finally, once we've run cyclictest to actually generate some events, we can see the output by looking at the wakeup_latency synthetic event's hist file:

```
# cat /sys/kernel/debug/tracing/events/synthetic/wakeup_latency/hist
```

- onmax(var).save(field,.. .)

  The 'onmax(var).save(field,...)' hist trigger action is invoked whenever the value of 'var' associated with a histogram entry exceeds the current maximum contained in that variable.

  The end result is that the trace event fields specified as the onmax.save() params will be saved if 'var' exceeds the current maximum for that hist trigger entry. This allows context from the event that exhibited the new maximum to be saved for later reference. When the histogram is displayed, additional fields displaying the saved values will be printed.

  As an example the below defines a couple of hist triggers, one for sched_waking and another for sched_switch, keyed on pid. Whenever a sched_waking occurs, the timestamp is saved in the entry corresponding to the current pid, and when the scheduler switches back to that pid, the timestamp difference is calculated. If the resulting latency, stored in wakeup_lat, exceeds the current maximum latency, the values specified in the save() fields are recorded:

  ```
  # echo 'hist:keys=pid:ts0=common_timestamp.usecs \
          if comm=="cyclictest"' >> \
          /sys/kernel/debug/tracing/events/sched/sched_waking/trigger

  # echo 'hist:keys=next_pid:\
          wakeup_lat=common_timestamp.usecs-$ts0:\
          onmax($wakeup_lat).save(next_comm,prev_pid,prev_prio,prev_comm) \
          if next_comm=="cyclictest"' >> \
          /sys/kernel/debug/tracing/events/sched/sched_switch/trigger
  ```

  When the histogram is displayed, the max value and the saved values corresponding to the max are displayed following the rest of the fields:

  ```
  # cat /sys/kernel/debug/tracing/events/sched/sched_switch/hist
    { next_pid:       2255 } hitcount:        239
      common_timestamp-ts0:          0
      max:         27
      next_comm: cyclictest
      prev_pid:          0  prev_prio:        120  prev_comm: swapper/1

    { next_pid:       2256 } hitcount:       2355
      common_timestamp-ts0: 0
      max:         49  next_comm: cyclictest
      prev_pid:          0  prev_prio:        120  prev_comm: swapper/0

    Totals:
        Hits: 12970
        Entries: 2
        Dropped: 0
  ```

- onmax(var).snapshot()

  The 'onmax(var).snapshot()' hist trigger action is invoked whenever the value of 'var' associated with a histogram entry exceeds the current maximum contained in that variable.

  The end result is that a global snapshot of the trace buffer will be saved in the tracing/snapshot file if 'var' exceeds the current maximum for any hist trigger entry.

  Note that in this case the maximum is a global maximum for the current trace instance, which is the maximum across all buckets of the histogram. The key of the specific trace event that caused the global maximum and the global maximum itself are displayed, along with a message stating that a snapshot has been taken and where to find it. The user can use the key information displayed to locate the corresponding bucket in the histogram for even more detail.

  As an example the below defines a couple of hist triggers, one for sched_waking and another for sched_switch, keyed on pid. Whenever a sched_waking event occurs, the timestamp is saved in the entry corresponding to the current pid, and when the scheduler switches back to that pid, the timestamp difference is calculated. If the resulting latency, stored in wakeup_lat, exceeds the current maximum latency, a snapshot is taken. As part of the setup, all the scheduler events are also enabled, which are the events that will show up in the snapshot when it is taken at some point:

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

  # echo 'hist:keys=pid:ts0=common_timestamp.usecs
  if comm=="cyclictest" >> /sys/kernel/debug/tracing/events/sched/sched_waking/trigger
  # echo 'hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0:
  onmax($wakeup_lat).save(next_prio,next_comm,prev_pid,prev_prio,
  prev_comm):onmax($wakeup_lat).snapshot() if next_comm=="cyclictest" >>
  /sys/kernel/debug/tracing/events/sched/sched_switch/trigger

  When the histogram is displayed, for each bucket the max value and the saved values corresponding to the max are displayed following the rest of the fields.

  If a snapshot was taken, there is also a message indicating that, along with the value and event that triggered the global maximum:

  # cat /sys/kernel/debug/tracing/events/sched/sched_switch/hist

  { next_pid: 2101 } hitcount: 200
  max: 52 next_prio: 120 next_comm: cyclictest prev_pid: 0 prev_prio: 120 prev_comm:

swapper/6
{ next_pid: 2103 } hitcount: 1326
max: 572 next_prio: 19 next_comm: cyclictest prev_pid: 0 prev_prio: 120 prev_comm:
swapper/1
{ next_pid: 2102 } hitcount: 1982
max: 74 next_prio: 19 next_comm: cyclictest prev_pid: 0 prev_prio: 120 prev_comm: swapper/5

Snapshot taken (see tracing/snapshot). Details:
triggering value { onmax($wakeup_lat) }: 572 triggered by event with key: { next_pid: 2103 }
Totals:
Hits: 3508 Entries: 3 Dropped: 0

In the above case, the event that triggered the global maximum has the key with next_pid == 2103. If you look at the bucket that has 2103 as the key, you'll find the additional values save()'d along with the local maximum for that bucket, which should be the same as the global maximum (since that was the same value that triggered the global snapshot).

And finally, looking at the snapshot data should show at or near the end the event that triggered the snapshot (in this case you can verify the timestamps between the sched_waking and sched_switch events, which should match the time displayed in the global maximum):

```
# cat /sys/kernel/debug/tracing/snapshot

    <...>-2103  [005] d..3   309.873125: sched_switch: prev_comm=cyclictest prev_pid=2103 prev_prio=19 prev_state=D ==> next_co
    <idle>-0     [005] d.h3   309.873611: sched_waking: comm=cyclictest pid=2102 prio=19 target_cpu=005
    <idle>-0     [005] dNh4   309.873613: sched_wakeup: comm=cyclictest pid=2102 prio=19 target_cpu=005
    <idle>-0     [005] d..3   309.873616: sched_switch: prev_comm=swapper/5 prev_pid=0 prev_prio=120 prev_state=S ==> next_comm
    <...>-2102  [005] d..3   309.873625: sched_switch: prev_comm=cyclictest prev_pid=2102 prev_prio=19 prev_state=D ==> next_co
    <idle>-0     [005] d.h3   309.874624: sched_waking: comm=cyclictest pid=2102 prio=19 target_cpu=005
    <idle>-0     [005] dNh4   309.874626: sched_wakeup: comm=cyclictest pid=2102 prio=19 target_cpu=005
    <idle>-0     [005] dNh3   309.874628: sched_waking: comm=cyclictest pid=2103 prio=19 target_cpu=005
    <idle>-0     [005] dNh4   309.874630: sched_wakeup: comm=cyclictest pid=2103 prio=19 target_cpu=005
    <idle>-0     [005] d..3   309.874633: sched_switch: prev_comm=swapper/5 prev_pid=0 prev_prio=120 prev_state=S ==> next_comm
    <idle>-0     [004] d.h3   309.874757: sched_waking: comm=gnome-terminal- pid=1699 prio=120 target_cpu=004
    <idle>-0     [004] dNh4   309.874762: sched_wakeup: comm=gnome-terminal- pid=1699 prio=120 target_cpu=004
    <idle>-0     [004] d..3   309.874766: sched_switch: prev_comm=swapper/4 prev_pid=0 prev_prio=120 prev_state=S ==> next_comm
gnome-terminal--1699  [004] d.h2   309.874941: sched_stat_runtime: comm=gnome-terminal- pid=1699 runtime=180706 [ns] vruntime=1
    <idle>-0     [003] d.s4   309.874956: sched_waking: comm=rcu_sched pid=9 prio=120 target_cpu=007
    <idle>-0     [003] d.s5   309.874960: sched_wake_idle_without_ipi: cpu=7
    <idle>-0     [003] d.s5   309.874961: sched_wakeup: comm=rcu_sched pid=9 prio=120 target_cpu=007
    <idle>-0     [007] d..3   309.874963: sched_switch: prev_comm=swapper/7 prev_pid=0 prev_prio=120 prev_state=S ==> next_comm
 rcu_sched-9     [007] d..3   309.874973: sched_stat_runtime: comm=rcu_sched pid=9 runtime=13646 [ns] vruntime=22531430286 [ns]
 rcu_sched-9     [007] d..3   309.874978: sched_switch: prev_comm=rcu_sched prev_pid=9 prev_prio=120 prev_state=R+ ==> next_comm
    <...>-2102  [005] d..4   309.874994: sched_migrate_task: comm=cyclictest pid=2103 prio=19 orig_cpu=5 dest_cpu=1
    <...>-2102  [005] d..4   309.875185: sched_wake_idle_without_ipi: cpu=1
    <idle>-0     [001] d..3   309.875200: sched_switch: prev_comm=swapper/1 prev_pid=0 prev_prio=120 prev_state=S ==> next_comm
```

- onchange(var).save(field,.. .)

  The 'onchange(var).save(field,...)' hist trigger action is invoked whenever the value of 'var' associated with a histogram entry changes.

  The end result is that the trace event fields specified as the onchange.save() params will be saved if 'var' changes for that hist trigger entry. This allows context from the event that changed the value to be saved for later reference. When the histogram is displayed, additional fields displaying the saved values will be printed.

- onchange(var).snapshot()

  The 'onchange(var).snapshot()' hist trigger action is invoked whenever the value of 'var' associated with a histogram entry changes.

  The end result is that a global snapshot of the trace buffer will be saved in the tracing/snapshot file if 'var' changes for any hist trigger entry.

  Note that in this case the changed value is a global variable associated with current trace instance. The key of the specific trace event that caused the value to change and the global value itself are displayed, along with a message stating that a snapshot has been taken and where to find it. The user can use the key information displayed to locate the corresponding bucket in the histogram for even more detail.

  As an example the below defines a hist trigger on the tcp_probe event, keyed on dport. Whenever a tcp_probe event occurs, the cwnd field is checked against the current value stored in the $cwnd variable. If the value has changed, a snapshot is taken. As part of the setup, all the scheduler and tcp events are also enabled, which are the events that will show up in the snapshot when it is taken at some point:

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

  # echo 'hist:keys=dport:cwnd=snd_cwnd:
  onchange($cwnd).save(snd_wnd,srtt,rcv_wnd): onchange($cwnd).snapshot()' >>
  /sys/kernel/debug/tracing/events/tcp/tcp_probe/trigger

  When the histogram is displayed, for each bucket the tracked value and the saved values corresponding to that value are displayed following the rest of the fields.

  If a snapshot was taken, there is also a message indicating that, along with the value and event that triggered the snapshot:

```
# cat /sys/kernel/debug/tracing/events/tcp/tcp_probe/hist

{ dport:       1521 } hitcount:          8
  changed:         10  snd_wnd:      35456  srtt:     154262  rcv_wnd:      42112

{ dport:         80 } hitcount:         23
  changed:         10  snd_wnd:      28960  srtt:      19604  rcv_wnd:      29312

{ dport:       9001 } hitcount:        172
  changed:         10  snd_wnd:      48384  srtt:     260444  rcv_wnd:      55168

{ dport:        443 } hitcount:        211
  changed:         10  snd_wnd:      26960  srtt:      17379  rcv_wnd:      28800
```

  Snapshot taken (see tracing/snapshot). Details:

```
    triggering value { onchange($cwnd) }:         10
    triggered by event with key: { dport:         80 }

Totals:
    Hits: 414
    Entries: 4
    Dropped: 0
```

  In the above case, the event that triggered the snapshot has the key with dport == 80. If you look at the bucket that has 80 as the key, you'll find the additional values save()'d along with the changed value for that bucket, which should be the same as the global changed value (since that was the same value that triggered the global snapshot).

  And finally, looking at the snapshot data should show at or near the end the event that triggered the snapshot:

```
# cat /sys/kernel/debug/tracing/snapshot

   gnome-shell-1261  [006] dN.3    49.823113: sched_stat_runtime: comm=gnome-shell pid=1261 runtime=49347 [ns] vruntime=1835730
 kworker/u16:4-773   [003] d..3    49.823114: sched_switch: prev_comm=kworker/u16:4 prev_pid=773 prev_prio=120 prev_state=R+ ==
   gnome-shell-1261  [006] d..3    49.823114: sched_switch: prev_comm=gnome-shell prev_pid=1261 prev_prio=120 prev_state=R+ ==>
   kworker/3:2-135   [003] d..3    49.823118: sched_stat_runtime: comm=kworker/3:2 pid=135 runtime=5339 [ns] vruntime=178158003
   kworker/6:2-387   [006] d..3    49.823120: sched_stat_runtime: comm=kworker/6:2 pid=387 runtime=9594 [ns] vruntime=145896053
   kworker/6:2-387   [006] d..3    49.823122: sched_switch: prev_comm=kworker/6:2 prev_pid=387 prev_prio=120 prev_state=R+ ==> r
   kworker/3:2-135   [003] d..3    49.823123: sched_switch: prev_comm=kworker/3:2 prev_pid=135 prev_prio=120 prev_state=T ==> ne
        <idle>-0     [004] ..s7    49.823798: tcp_probe: src=10.0.0.10:54326 dest=23.215.104.193:80 mark=0x0 length=32 snd_nxt=(
```

3. User space creating a trigger

Writing into /sys/kernel/tracing/trace_marker writes into the ftrace ring buffer. This can also act like an event, by writing into the trigger file located in /sys/kernel/tracing/events/ftrace/print/

Modifying cyclictest to write into the trace_marker file before it sleeps and after it wakes up, something like this:

```
static void traceputs(char *str)
{
      /* tracemark_fd is the trace_marker file descriptor */
      if (tracemark_fd < 0)
            return;
      /* write the tracemark message */
      write(tracemark_fd, str, strlen(str));
}
```

And later add something like:

```
traceputs("start");
clock_nanosleep(...);
traceputs("end");
```

We can make a histogram from this:

```
# cd /sys/kernel/tracing
# echo 'latency u64 lat' > synthetic_events
# echo 'hist:keys=common_pid:ts0=common_timestamp.usecs if buf == "start"' > events/ftrace/print/trigger
# echo 'hist:keys=common_pid:lat=common_timestamp.usecs-$ts0:onmatch(ftrace.print).latency($lat) if buf == "end"' >> events/ftrace/print/
# echo 'hist:keys=lat,common_pid:sort=lat' > events/synthetic/latency/trigger
```

The above created a synthetic event called "latency" and two histograms against the trace_marker, one gets triggered when "start" is written into the trace_marker file and the other when "end" is written. If the pids match, then it will call the "latency" synthetic event with the calculated latency as its parameter. Finally, a histogram is added to the latency synthetic event to record the calculated latency along with the pid.

Now running cyclictest with:

```
# ./cyclictest -p80 -d0 -i250 -n -a -t --tracemark -b 1000

-p80  : run threads at priority 80
-d0   : have all threads run at the same interval
-i250 : start the interval at 250 microseconds (all threads will do this)
-n    : sleep with nanosleep
-a    : affine all threads to a separate CPU
-t    : one thread per available CPU
--tracemark : enable trace mark writing
-b 1000 : stop if any latency is greater than 1000 microseconds
```

Note, the -b 1000 is used just to make --tracemark available.

Then we can see the histogram created by this with:

```
# cat events/synthetic/latency/hist
# event histogram
#
# trigger info: hist:keys=lat,common_pid:vals=hitcount:sort=lat:size=2048 [active]
#

{ lat:        107, common_pid:       2039 } hitcount:          1
{ lat:        122, common_pid:       2041 } hitcount:          1
{ lat:        166, common_pid:       2039 } hitcount:          1
{ lat:        174, common_pid:       2039 } hitcount:          1
{ lat:        194, common_pid:       2041 } hitcount:          1
{ lat:        196, common_pid:       2036 } hitcount:          1
{ lat:        197, common_pid:       2038 } hitcount:          1
{ lat:        198, common_pid:       2039 } hitcount:          1
{ lat:        199, common_pid:       2039 } hitcount:          1
{ lat:        200, common_pid:       2041 } hitcount:          1
{ lat:        201, common_pid:       2039 } hitcount:          2
{ lat:        202, common_pid:       2038 } hitcount:          1
{ lat:        202, common_pid:       2043 } hitcount:          1
{ lat:        203, common_pid:       2039 } hitcount:          1
{ lat:        203, common_pid:       2036 } hitcount:          1
{ lat:        203, common_pid:       2041 } hitcount:          1
{ lat:        206, common_pid:       2038 } hitcount:          2
{ lat:        207, common_pid:       2039 } hitcount:          1
{ lat:        207, common_pid:       2036 } hitcount:          1
{ lat:        208, common_pid:       2040 } hitcount:          1
{ lat:        209, common_pid:       2043 } hitcount:          1
{ lat:        210, common_pid:       2039 } hitcount:          1
{ lat:        211, common_pid:       2039 } hitcount:          4
{ lat:        212, common_pid:       2043 } hitcount:          1
{ lat:        212, common_pid:       2039 } hitcount:          2
{ lat:        213, common_pid:       2039 } hitcount:          1
{ lat:        214, common_pid:       2038 } hitcount:          1
{ lat:        214, common_pid:       2039 } hitcount:          2
{ lat:        214, common_pid:       2042 } hitcount:          1
{ lat:        215, common_pid:       2039 } hitcount:          1
{ lat:        217, common_pid:       2036 } hitcount:          1
{ lat:        217, common_pid:       2040 } hitcount:          1
{ lat:        217, common_pid:       2039 } hitcount:          1
{ lat:        218, common_pid:       2039 } hitcount:          6
{ lat:        219, common_pid:       2039 } hitcount:          9
{ lat:        220, common_pid:       2039 } hitcount:         11
{ lat:        221, common_pid:       2039 } hitcount:          5
{ lat:        221, common_pid:       2042 } hitcount:          1
{ lat:        222, common_pid:       2039 } hitcount:          7
{ lat:        223, common_pid:       2036 } hitcount:          1
{ lat:        223, common_pid:       2039 } hitcount:          3
{ lat:        224, common_pid:       2039 } hitcount:          4
{ lat:        224, common_pid:       2037 } hitcount:          1
{ lat:        224, common_pid:       2036 } hitcount:          2
{ lat:        225, common_pid:       2039 } hitcount:          5
{ lat:        225, common_pid:       2042 } hitcount:          1
{ lat:        226, common_pid:       2039 } hitcount:          7
{ lat:        226, common_pid:       2036 } hitcount:          4
{ lat:        227, common_pid:       2039 } hitcount:          6
{ lat:        227, common_pid:       2036 } hitcount:         12
{ lat:        227, common_pid:       2043 } hitcount:          1
{ lat:        228, common_pid:       2039 } hitcount:          7
{ lat:        228, common_pid:       2036 } hitcount:         14
{ lat:        229, common_pid:       2039 } hitcount:          9
{ lat:        229, common_pid:       2036 } hitcount:          8
{ lat:        229, common_pid:       2038 } hitcount:          1
{ lat:        230, common_pid:       2039 } hitcount:         11
{ lat:        230, common_pid:       2036 } hitcount:          6
{ lat:        230, common_pid:       2043 } hitcount:          1
{ lat:        230, common_pid:       2042 } hitcount:          2
{ lat:        231, common_pid:       2041 } hitcount:          1
{ lat:        231, common_pid:       2036 } hitcount:          6
{ lat:        231, common_pid:       2043 } hitcount:          1
{ lat:        231, common_pid:       2039 } hitcount:          8
{ lat:        232, common_pid:       2037 } hitcount:          1
{ lat:        232, common_pid:       2039 } hitcount:          6
{ lat:        232, common_pid:       2040 } hitcount:          2
{ lat:        232, common_pid:       2036 } hitcount:          5
{ lat:        232, common_pid:       2043 } hitcount:          1
{ lat:        233, common_pid:       2036 } hitcount:          5
{ lat:        233, common_pid:       2039 } hitcount:         11
{ lat:        234, common_pid:       2039 } hitcount:          4
{ lat:        234, common_pid:       2038 } hitcount:          2
```

```
{ lat:       234, common_pid:      2043 } hitcount:         2
{ lat:       234, common_pid:      2036 } hitcount:        11
{ lat:       234, common_pid:      2040 } hitcount:         1
{ lat:       235, common_pid:      2037 } hitcount:         2
{ lat:       235, common_pid:      2036 } hitcount:         8
{ lat:       235, common_pid:      2043 } hitcount:         2
{ lat:       235, common_pid:      2039 } hitcount:         5
{ lat:       235, common_pid:      2042 } hitcount:         2
{ lat:       235, common_pid:      2040 } hitcount:         4
{ lat:       235, common_pid:      2041 } hitcount:         1
{ lat:       236, common_pid:      2036 } hitcount:         7
{ lat:       236, common_pid:      2037 } hitcount:         1
{ lat:       236, common_pid:      2041 } hitcount:         5
{ lat:       236, common_pid:      2039 } hitcount:         3
{ lat:       236, common_pid:      2043 } hitcount:         9
{ lat:       236, common_pid:      2040 } hitcount:         7
{ lat:       237, common_pid:      2037 } hitcount:         1
{ lat:       237, common_pid:      2040 } hitcount:         1
{ lat:       237, common_pid:      2036 } hitcount:         9
{ lat:       237, common_pid:      2039 } hitcount:         3
{ lat:       237, common_pid:      2043 } hitcount:         8
{ lat:       237, common_pid:      2042 } hitcount:         2
{ lat:       237, common_pid:      2041 } hitcount:         2
{ lat:       238, common_pid:      2043 } hitcount:        10
{ lat:       238, common_pid:      2040 } hitcount:         1
{ lat:       238, common_pid:      2037 } hitcount:         9
{ lat:       238, common_pid:      2038 } hitcount:         1
{ lat:       238, common_pid:      2039 } hitcount:         1
{ lat:       238, common_pid:      2042 } hitcount:         3
{ lat:       238, common_pid:      2036 } hitcount:         7
{ lat:       239, common_pid:      2041 } hitcount:         1
{ lat:       239, common_pid:      2043 } hitcount:        11
{ lat:       239, common_pid:      2037 } hitcount:        11
{ lat:       239, common_pid:      2038 } hitcount:         6
{ lat:       239, common_pid:      2036 } hitcount:         7
{ lat:       239, common_pid:      2040 } hitcount:         1
{ lat:       239, common_pid:      2042 } hitcount:         9
{ lat:       240, common_pid:      2037 } hitcount:        29
{ lat:       240, common_pid:      2043 } hitcount:        15
{ lat:       240, common_pid:      2040 } hitcount:        44
{ lat:       240, common_pid:      2039 } hitcount:         1
{ lat:       240, common_pid:      2041 } hitcount:         2
{ lat:       240, common_pid:      2038 } hitcount:         1
{ lat:       240, common_pid:      2036 } hitcount:        10
{ lat:       240, common_pid:      2042 } hitcount:        13
{ lat:       241, common_pid:      2036 } hitcount:        21
{ lat:       241, common_pid:      2041 } hitcount:        36
{ lat:       241, common_pid:      2037 } hitcount:        34
{ lat:       241, common_pid:      2042 } hitcount:        14
{ lat:       241, common_pid:      2040 } hitcount:        94
{ lat:       241, common_pid:      2039 } hitcount:        12
{ lat:       241, common_pid:      2038 } hitcount:         2
{ lat:       241, common_pid:      2043 } hitcount:        28
{ lat:       242, common_pid:      2040 } hitcount:       109
{ lat:       242, common_pid:      2041 } hitcount:       506
{ lat:       242, common_pid:      2039 } hitcount:       155
{ lat:       242, common_pid:      2042 } hitcount:        21
{ lat:       242, common_pid:      2037 } hitcount:        52
{ lat:       242, common_pid:      2043 } hitcount:        21
{ lat:       242, common_pid:      2036 } hitcount:        16
{ lat:       242, common_pid:      2038 } hitcount:       156
{ lat:       243, common_pid:      2037 } hitcount:        46
{ lat:       243, common_pid:      2039 } hitcount:        40
{ lat:       243, common_pid:      2042 } hitcount:       119
{ lat:       243, common_pid:      2041 } hitcount:       611
{ lat:       243, common_pid:      2036 } hitcount:        69
{ lat:       243, common_pid:      2038 } hitcount:       784
{ lat:       243, common_pid:      2040 } hitcount:       323
{ lat:       243, common_pid:      2043 } hitcount:        14
{ lat:       244, common_pid:      2043 } hitcount:        35
{ lat:       244, common_pid:      2042 } hitcount:       305
{ lat:       244, common_pid:      2039 } hitcount:         8
{ lat:       244, common_pid:      2040 } hitcount:      4515
{ lat:       244, common_pid:      2038 } hitcount:       371
{ lat:       244, common_pid:      2037 } hitcount:        31
{ lat:       244, common_pid:      2036 } hitcount:       114
{ lat:       244, common_pid:      2041 } hitcount:      3396
{ lat:       245, common_pid:      2036 } hitcount:       700
{ lat:       245, common_pid:      2041 } hitcount:      2772
{ lat:       245, common_pid:      2037 } hitcount:       268
{ lat:       245, common_pid:      2039 } hitcount:       472
{ lat:       245, common_pid:      2038 } hitcount:      2758
{ lat:       245, common_pid:      2042 } hitcount:      3833
{ lat:       245, common_pid:      2040 } hitcount:      3105
{ lat:       245, common_pid:      2043 } hitcount:       645
{ lat:       246, common_pid:      2038 } hitcount:      3451
{ lat:       246, common_pid:      2041 } hitcount:       142
{ lat:       246, common_pid:      2037 } hitcount:      5101
{ lat:       246, common_pid:      2040 } hitcount:        68
{ lat:       246, common_pid:      2043 } hitcount:      5099
{ lat:       246, common_pid:      2039 } hitcount:      5608
{ lat:       246, common_pid:      2042 } hitcount:      3723
{ lat:       246, common_pid:      2036 } hitcount:      4738
{ lat:       247, common_pid:      2042 } hitcount:       312
{ lat:       247, common_pid:      2043 } hitcount:      2385
{ lat:       247, common_pid:      2041 } hitcount:       452
{ lat:       247, common_pid:      2038 } hitcount:       792
{ lat:       247, common_pid:      2040 } hitcount:        78
{ lat:       247, common_pid:      2036 } hitcount:      2375
{ lat:       247, common_pid:      2039 } hitcount:      1834
{ lat:       247, common_pid:      2037 } hitcount:      2655
{ lat:       248, common_pid:      2037 } hitcount:        36
{ lat:       248, common_pid:      2042 } hitcount:        11
{ lat:       248, common_pid:      2038 } hitcount:       122
{ lat:       248, common_pid:      2036 } hitcount:       135
{ lat:       248, common_pid:      2039 } hitcount:        26
{ lat:       248, common_pid:      2041 } hitcount:       503
{ lat:       248, common_pid:      2043 } hitcount:        66
{ lat:       248, common_pid:      2040 } hitcount:        46
{ lat:       249, common_pid:      2037 } hitcount:        29
{ lat:       249, common_pid:      2038 } hitcount:         1
{ lat:       249, common_pid:      2043 } hitcount:        29
{ lat:       249, common_pid:      2039 } hitcount:         8
{ lat:       249, common_pid:      2042 } hitcount:        56
{ lat:       249, common_pid:      2040 } hitcount:        27
{ lat:       249, common_pid:      2041 } hitcount:        11
{ lat:       249, common_pid:      2036 } hitcount:        27
{ lat:       250, common_pid:      2038 } hitcount:         1
{ lat:       250, common_pid:      2036 } hitcount:        30
{ lat:       250, common_pid:      2040 } hitcount:        19
{ lat:       250, common_pid:      2043 } hitcount:        22
{ lat:       250, common_pid:      2042 } hitcount:        20
{ lat:       250, common_pid:      2041 } hitcount:         1
{ lat:       250, common_pid:      2039 } hitcount:         6
{ lat:       250, common_pid:      2037 } hitcount:        48
{ lat:       251, common_pid:      2037 } hitcount:        43
{ lat:       251, common_pid:      2039 } hitcount:         1
{ lat:       251, common_pid:      2036 } hitcount:        12
{ lat:       251, common_pid:      2042 } hitcount:         2
{ lat:       251, common_pid:      2041 } hitcount:         1
```

```
{ lat:        251, common_pid:       2043 } hitcount:         15
{ lat:        251, common_pid:       2040 } hitcount:          3
{ lat:        252, common_pid:       2040 } hitcount:          1
{ lat:        252, common_pid:       2036 } hitcount:         12
{ lat:        252, common_pid:       2037 } hitcount:         21
{ lat:        252, common_pid:       2043 } hitcount:         14
{ lat:        253, common_pid:       2037 } hitcount:         21
{ lat:        253, common_pid:       2039 } hitcount:          2
{ lat:        253, common_pid:       2036 } hitcount:          9
{ lat:        253, common_pid:       2043 } hitcount:          6
{ lat:        253, common_pid:       2040 } hitcount:          1
{ lat:        254, common_pid:       2036 } hitcount:          8
{ lat:        254, common_pid:       2043 } hitcount:          3
{ lat:        254, common_pid:       2041 } hitcount:          1
{ lat:        254, common_pid:       2042 } hitcount:          1
{ lat:        254, common_pid:       2039 } hitcount:          1
{ lat:        254, common_pid:       2037 } hitcount:         12
{ lat:        255, common_pid:       2043 } hitcount:          1
{ lat:        255, common_pid:       2037 } hitcount:          2
{ lat:        255, common_pid:       2036 } hitcount:          2
{ lat:        255, common_pid:       2039 } hitcount:          8
{ lat:        256, common_pid:       2043 } hitcount:          1
{ lat:        256, common_pid:       2036 } hitcount:          4
{ lat:        256, common_pid:       2039 } hitcount:          6
{ lat:        257, common_pid:       2039 } hitcount:          5
{ lat:        257, common_pid:       2036 } hitcount:          4
{ lat:        258, common_pid:       2039 } hitcount:          5
{ lat:        258, common_pid:       2036 } hitcount:          2
{ lat:        259, common_pid:       2036 } hitcount:          7
{ lat:        259, common_pid:       2039 } hitcount:          7
{ lat:        260, common_pid:       2036 } hitcount:          8
{ lat:        260, common_pid:       2039 } hitcount:          6
{ lat:        261, common_pid:       2036 } hitcount:          5
{ lat:        261, common_pid:       2039 } hitcount:          7
{ lat:        262, common_pid:       2039 } hitcount:          5
{ lat:        262, common_pid:       2036 } hitcount:          5
{ lat:        263, common_pid:       2039 } hitcount:          7
{ lat:        263, common_pid:       2036 } hitcount:          7
{ lat:        264, common_pid:       2039 } hitcount:          9
{ lat:        264, common_pid:       2036 } hitcount:          9
{ lat:        265, common_pid:       2036 } hitcount:          5
{ lat:        265, common_pid:       2039 } hitcount:          1
{ lat:        266, common_pid:       2036 } hitcount:          1
{ lat:        266, common_pid:       2039 } hitcount:          3
{ lat:        267, common_pid:       2036 } hitcount:          1
{ lat:        267, common_pid:       2039 } hitcount:          3
{ lat:        268, common_pid:       2036 } hitcount:          1
{ lat:        268, common_pid:       2039 } hitcount:          6
{ lat:        269, common_pid:       2036 } hitcount:          1
{ lat:        269, common_pid:       2043 } hitcount:          1
{ lat:        269, common_pid:       2039 } hitcount:          2
{ lat:        270, common_pid:       2040 } hitcount:          1
{ lat:        270, common_pid:       2039 } hitcount:          6
{ lat:        271, common_pid:       2041 } hitcount:          1
{ lat:        271, common_pid:       2039 } hitcount:          5
{ lat:        272, common_pid:       2039 } hitcount:         10
{ lat:        273, common_pid:       2039 } hitcount:          8
{ lat:        274, common_pid:       2039 } hitcount:          2
{ lat:        275, common_pid:       2039 } hitcount:          1
{ lat:        276, common_pid:       2039 } hitcount:          2
{ lat:        276, common_pid:       2037 } hitcount:          1
{ lat:        276, common_pid:       2038 } hitcount:          1
{ lat:        277, common_pid:       2039 } hitcount:          1
{ lat:        277, common_pid:       2042 } hitcount:          1
{ lat:        278, common_pid:       2039 } hitcount:          1
{ lat:        279, common_pid:       2039 } hitcount:          4
{ lat:        279, common_pid:       2043 } hitcount:          1
{ lat:        280, common_pid:       2039 } hitcount:          3
{ lat:        283, common_pid:       2036 } hitcount:          2
{ lat:        284, common_pid:       2039 } hitcount:          1
{ lat:        284, common_pid:       2043 } hitcount:          1
{ lat:        288, common_pid:       2039 } hitcount:          1
{ lat:        289, common_pid:       2039 } hitcount:          1
{ lat:        300, common_pid:       2039 } hitcount:          1
{ lat:        384, common_pid:       2039 } hitcount:          1

Totals:
    Hits: 67625
    Entries: 278
    Dropped: 0
```

Note, the writes are around the sleep, so ideally they will all be of 250 microseconds. If you are wondering how there are several that are under 250 microseconds, that is because the way cyclictest works, is if one iteration comes in late, the next one will set the timer to wake up less that 250. That is, if an iteration came in 50 microseconds late, the next wake up will be at 200 microseconds.

But this could easily be done in userspace. To make this even more interesting, we can mix the histogram between events that happened in the kernel with trace_marker:

```
# cd /sys/kernel/tracing
# echo 'latency u64 lat' > synthetic_events
# echo 'hist:keys=pid:ts0=common_timestamp.usecs' > events/sched/sched_waking/trigger
# echo 'hist:keys=common_pid:lat=common_timestamp.usecs-$ts0:onmatch(sched.sched_waking).latency($lat) if buf == "end"' > events/ftrace/p
# echo 'hist:keys=lat,common_pid:sort=lat' > events/synthetic/latency/trigger
```

The difference this time is that instead of using the trace_marker to start the latency, the sched_waking event is used, matching the common_pid for the trace_marker write with the pid that is being woken by sched_waking.

After running cyclictest again with the same parameters, we now have:

```
# cat events/synthetic/latency/hist
# event histogram
#
# trigger info: hist:keys=lat,common_pid:vals=hitcount:sort=lat:size=2048 [active]
#

{ lat:          7, common_pid:       2302 } hitcount:        640
{ lat:          7, common_pid:       2299 } hitcount:         42
{ lat:          7, common_pid:       2303 } hitcount:         18
{ lat:          7, common_pid:       2305 } hitcount:        166
{ lat:          7, common_pid:       2306 } hitcount:          1
{ lat:          7, common_pid:       2301 } hitcount:         91
{ lat:          7, common_pid:       2300 } hitcount:         17
{ lat:          8, common_pid:       2303 } hitcount:       8296
{ lat:          8, common_pid:       2304 } hitcount:       6864
{ lat:          8, common_pid:       2305 } hitcount:       9464
{ lat:          8, common_pid:       2301 } hitcount:       9213
{ lat:          8, common_pid:       2306 } hitcount:       6246
{ lat:          8, common_pid:       2302 } hitcount:       8797
{ lat:          8, common_pid:       2299 } hitcount:       8771
{ lat:          8, common_pid:       2300 } hitcount:       8119
{ lat:          9, common_pid:       2305 } hitcount:       1519
{ lat:          9, common_pid:       2299 } hitcount:       2346
{ lat:          9, common_pid:       2303 } hitcount:       2841
{ lat:          9, common_pid:       2301 } hitcount:       1846
{ lat:          9, common_pid:       2304 } hitcount:       3861
{ lat:          9, common_pid:       2302 } hitcount:       1210
{ lat:          9, common_pid:       2300 } hitcount:       2762
{ lat:          9, common_pid:       2306 } hitcount:       4247
{ lat:         10, common_pid:       2299 } hitcount:         16
{ lat:         10, common_pid:       2306 } hitcount:        333
```

```
{ lat:        10, common_pid:       2303 } hitcount:        16
{ lat:        10, common_pid:       2304 } hitcount:       168
{ lat:        10, common_pid:       2302 } hitcount:       240
{ lat:        10, common_pid:       2301 } hitcount:        28
{ lat:        10, common_pid:       2300 } hitcount:        95
{ lat:        10, common_pid:       2305 } hitcount:        18
{ lat:        11, common_pid:       2303 } hitcount:         5
{ lat:        11, common_pid:       2305 } hitcount:         8
{ lat:        11, common_pid:       2306 } hitcount:       221
{ lat:        11, common_pid:       2302 } hitcount:        76
{ lat:        11, common_pid:       2304 } hitcount:        26
{ lat:        11, common_pid:       2300 } hitcount:       125
{ lat:        11, common_pid:       2299 } hitcount:         2
{ lat:        12, common_pid:       2305 } hitcount:         3
{ lat:        12, common_pid:       2300 } hitcount:         6
{ lat:        12, common_pid:       2306 } hitcount:        90
{ lat:        12, common_pid:       2302 } hitcount:         4
{ lat:        12, common_pid:       2303 } hitcount:         1
{ lat:        12, common_pid:       2304 } hitcount:       122
{ lat:        13, common_pid:       2300 } hitcount:        12
{ lat:        13, common_pid:       2301 } hitcount:         1
{ lat:        13, common_pid:       2306 } hitcount:        32
{ lat:        13, common_pid:       2302 } hitcount:         5
{ lat:        13, common_pid:       2305 } hitcount:         1
{ lat:        13, common_pid:       2303 } hitcount:         1
{ lat:        13, common_pid:       2304 } hitcount:        61
{ lat:        14, common_pid:       2303 } hitcount:         4
{ lat:        14, common_pid:       2306 } hitcount:         5
{ lat:        14, common_pid:       2305 } hitcount:         4
{ lat:        14, common_pid:       2304 } hitcount:        62
{ lat:        14, common_pid:       2302 } hitcount:        19
{ lat:        14, common_pid:       2300 } hitcount:        33
{ lat:        14, common_pid:       2299 } hitcount:         1
{ lat:        14, common_pid:       2301 } hitcount:         4
{ lat:        15, common_pid:       2305 } hitcount:         1
{ lat:        15, common_pid:       2302 } hitcount:        25
{ lat:        15, common_pid:       2300 } hitcount:        11
{ lat:        15, common_pid:       2299 } hitcount:         5
{ lat:        15, common_pid:       2301 } hitcount:         1
{ lat:        15, common_pid:       2304 } hitcount:         8
{ lat:        15, common_pid:       2303 } hitcount:         1
{ lat:        15, common_pid:       2306 } hitcount:         6
{ lat:        16, common_pid:       2302 } hitcount:        31
{ lat:        16, common_pid:       2306 } hitcount:         3
{ lat:        16, common_pid:       2300 } hitcount:         5
{ lat:        17, common_pid:       2302 } hitcount:         6
{ lat:        17, common_pid:       2303 } hitcount:         1
{ lat:        18, common_pid:       2304 } hitcount:         1
{ lat:        18, common_pid:       2302 } hitcount:         8
{ lat:        18, common_pid:       2299 } hitcount:         1
{ lat:        18, common_pid:       2301 } hitcount:         1
{ lat:        19, common_pid:       2303 } hitcount:         4
{ lat:        19, common_pid:       2304 } hitcount:         5
{ lat:        19, common_pid:       2302 } hitcount:         4
{ lat:        19, common_pid:       2299 } hitcount:         3
{ lat:        19, common_pid:       2306 } hitcount:         1
{ lat:        19, common_pid:       2300 } hitcount:         4
{ lat:        19, common_pid:       2305 } hitcount:         5
{ lat:        20, common_pid:       2299 } hitcount:         2
{ lat:        20, common_pid:       2302 } hitcount:         3
{ lat:        20, common_pid:       2305 } hitcount:         1
{ lat:        20, common_pid:       2300 } hitcount:         2
{ lat:        20, common_pid:       2301 } hitcount:         2
{ lat:        20, common_pid:       2303 } hitcount:         3
{ lat:        21, common_pid:       2305 } hitcount:         1
{ lat:        21, common_pid:       2299 } hitcount:         5
{ lat:        21, common_pid:       2303 } hitcount:         4
{ lat:        21, common_pid:       2302 } hitcount:         7
{ lat:        21, common_pid:       2300 } hitcount:         1
{ lat:        21, common_pid:       2301 } hitcount:         5
{ lat:        21, common_pid:       2304 } hitcount:         2
{ lat:        22, common_pid:       2302 } hitcount:         5
{ lat:        22, common_pid:       2303 } hitcount:         1
{ lat:        22, common_pid:       2306 } hitcount:         3
{ lat:        22, common_pid:       2301 } hitcount:         2
{ lat:        22, common_pid:       2300 } hitcount:         1
{ lat:        22, common_pid:       2299 } hitcount:         1
{ lat:        22, common_pid:       2305 } hitcount:         1
{ lat:        22, common_pid:       2304 } hitcount:         1
{ lat:        23, common_pid:       2299 } hitcount:         1
{ lat:        23, common_pid:       2306 } hitcount:         2
{ lat:        23, common_pid:       2302 } hitcount:         6
{ lat:        24, common_pid:       2302 } hitcount:         3
{ lat:        24, common_pid:       2300 } hitcount:         1
{ lat:        24, common_pid:       2306 } hitcount:         2
{ lat:        24, common_pid:       2305 } hitcount:         1
{ lat:        24, common_pid:       2299 } hitcount:         1
{ lat:        25, common_pid:       2300 } hitcount:         1
{ lat:        25, common_pid:       2302 } hitcount:         4
{ lat:        26, common_pid:       2302 } hitcount:         2
{ lat:        27, common_pid:       2305 } hitcount:         1
{ lat:        27, common_pid:       2300 } hitcount:         1
{ lat:        27, common_pid:       2302 } hitcount:         3
{ lat:        28, common_pid:       2306 } hitcount:         1
{ lat:        28, common_pid:       2302 } hitcount:         4
{ lat:        29, common_pid:       2302 } hitcount:         1
{ lat:        29, common_pid:       2300 } hitcount:         2
{ lat:        29, common_pid:       2306 } hitcount:         1
{ lat:        29, common_pid:       2304 } hitcount:         1
{ lat:        30, common_pid:       2302 } hitcount:         4
{ lat:        31, common_pid:       2302 } hitcount:         6
{ lat:        32, common_pid:       2302 } hitcount:         1
{ lat:        33, common_pid:       2299 } hitcount:         1
{ lat:        33, common_pid:       2302 } hitcount:         3
{ lat:        34, common_pid:       2302 } hitcount:         2
{ lat:        35, common_pid:       2302 } hitcount:         1
{ lat:        35, common_pid:       2304 } hitcount:         1
{ lat:        36, common_pid:       2302 } hitcount:         4
{ lat:        37, common_pid:       2302 } hitcount:         6
{ lat:        38, common_pid:       2302 } hitcount:         2
{ lat:        39, common_pid:       2302 } hitcount:         2
{ lat:        39, common_pid:       2304 } hitcount:         1
{ lat:        40, common_pid:       2304 } hitcount:         2
{ lat:        40, common_pid:       2302 } hitcount:         5
{ lat:        41, common_pid:       2304 } hitcount:         1
{ lat:        41, common_pid:       2302 } hitcount:         8
{ lat:        42, common_pid:       2302 } hitcount:         6
{ lat:        42, common_pid:       2304 } hitcount:         1
{ lat:        43, common_pid:       2302 } hitcount:         3
{ lat:        43, common_pid:       2304 } hitcount:         4
{ lat:        44, common_pid:       2302 } hitcount:         6
{ lat:        45, common_pid:       2302 } hitcount:         5
{ lat:        46, common_pid:       2302 } hitcount:         5
{ lat:        47, common_pid:       2302 } hitcount:         7
{ lat:        48, common_pid:       2301 } hitcount:         1
{ lat:        48, common_pid:       2302 } hitcount:         9
{ lat:        49, common_pid:       2302 } hitcount:         3
{ lat:        50, common_pid:       2302 } hitcount:         1
{ lat:        50, common_pid:       2301 } hitcount:         1
{ lat:        51, common_pid:       2302 } hitcount:         2
```

```
{ lat:         51, common_pid:       2301 } hitcount:          1
{ lat:         61, common_pid:       2302 } hitcount:          1
{ lat:        110, common_pid:       2302 } hitcount:          1

Totals:
    Hits: 89565
    Entries: 158
    Dropped: 0
```

This doesn't tell us any information about how late cyclictest may have woken up, but it does show us a nice histogram of how long it took from the time that cyclictest was woken to the time it made it into user space.