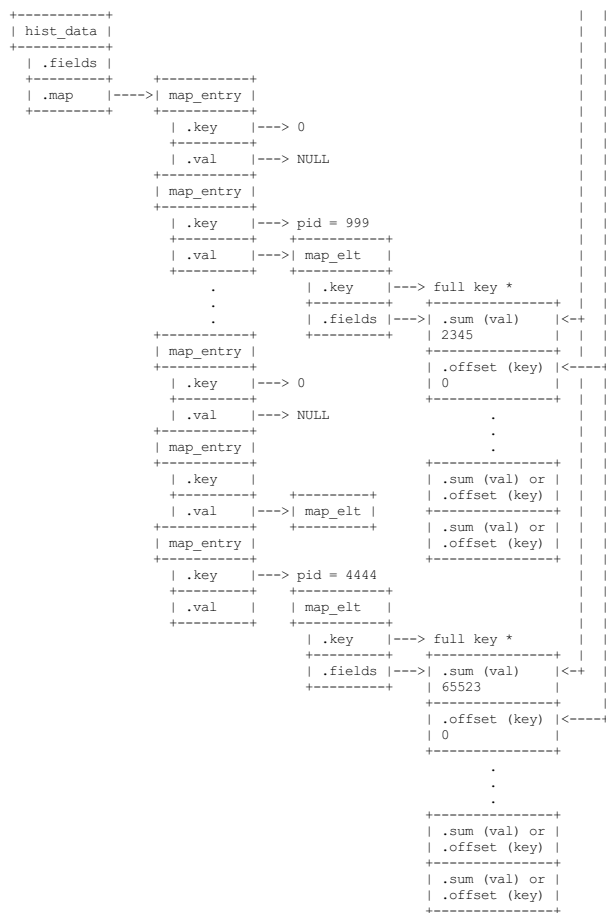


has || been claimed, the .key value contains the key's hash value and the || .val member points to a map_elt containing the full key and an entry || for each key or value in the map_elt.fields[] array. There is an || entry in the map_elt.fields[] array corresponding to each hist_field || in the histogram, and this is where the continually aggregated sums || corresponding to each histogram value are kept. ||

The diagram attempts to show the relationship between the || hist_data.fields[] and the map_elt.fields[] with the links drawn || between diagrams:



Abbreviations used in the diagrams:

```

hist_data = struct hist_trigger_data
hist_data.fields = struct hist_field
fn = hist_field_fn_t
map_entry = struct tracing_map_entry
map_elt = struct tracing_map_elt
map_elt.fields = struct tracing_map_field

```

Whenever a new event occurs and it has a hist trigger associated with it, event_hist_trigger() is called. event_hist_trigger() first deals with the key: for each subkey in the key (in the above example, there is just one subkey corresponding to pid), the hist_field that represents that subkey is retrieved from hist_data.fields[] and the hist_field_fn_t fn() associated with that field, along with the field's size and offset, is used to grab that subkey's data from the current trace record.

Once the complete key has been retrieved, it's used to look that key up in the tracing_map. If there's no tracing_map_elt associated with that key, an empty one is claimed and inserted in the map for the new key. In either case, the tracing_map_elt associated with that key is returned.

Once a tracing_map_elt available, hist_trigger_elt_update() is called. As the name implies, this updates the element, which basically means updating the element's fields. There's a tracing_map_field associated with each key and value in the histogram, and each of these correspond to the key and value hist_fields created when the histogram was created. hist_trigger_elt_update() goes through each value hist_field and, as for the keys, uses the hist_field's fn() and size and offset to grab the field's value from the current trace record. Once it has that value, it simply adds that value to that field's continually-updated tracing_map_field.sum member. Some hist_field fn(s), such as for the hitcount, don't actually grab anything from the trace record (the hitcount fn() just increments the counter sum by 1), but the idea is the same.

Once all the values have been updated, hist_trigger_elt_update() is done and returns. Note that there are also tracing_map_fields for each subkey in the key, but hist_trigger_elt_update() doesn't look at them or update anything - those exist only for sorting, which can happen later.

Basic histogram test

This is a good example to try. It produces 3 value fields and 2 key fields in the output:

```
# echo 'hist:keys=common_pid,call_site.sym:values=bytes_req,bytes_alloc,hitcount' >> events/kmem/kmalloc/trigger
```

To see the debug data, cat the kmem/kmalloc's 'hist_debug' file. It will show the trigger info of the histogram it corresponds to, along with the address of the hist_data associated with the histogram, which will become useful in later examples. It then displays the number of total hist_fields associated with the histogram along with a count of how many of those correspond to keys and how many correspond to values.

It then goes on to display details for each field, including the field's flags and the position of each field in the hist_data's fields[] array, which is useful information for verifying that things internally appear correct or not, and which again will become even more useful in further examples:

```

# cat events/kmem/kmalloc/hist_debug

# event histogram
#
# trigger info: hist:keys=common_pid,call_site.sym:vals=hitcount,bytes_req,bytes_alloc:sort=hitcount:size=2048 [active]
#

hist_data: 00000005e48c9a5

n_vals: 3
n_keys: 2
n_fields: 5

val fields:

```

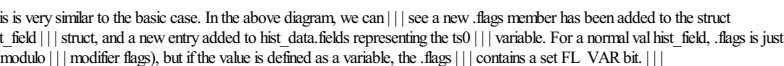
The commands below can be used to clean things up for the next test:

variables

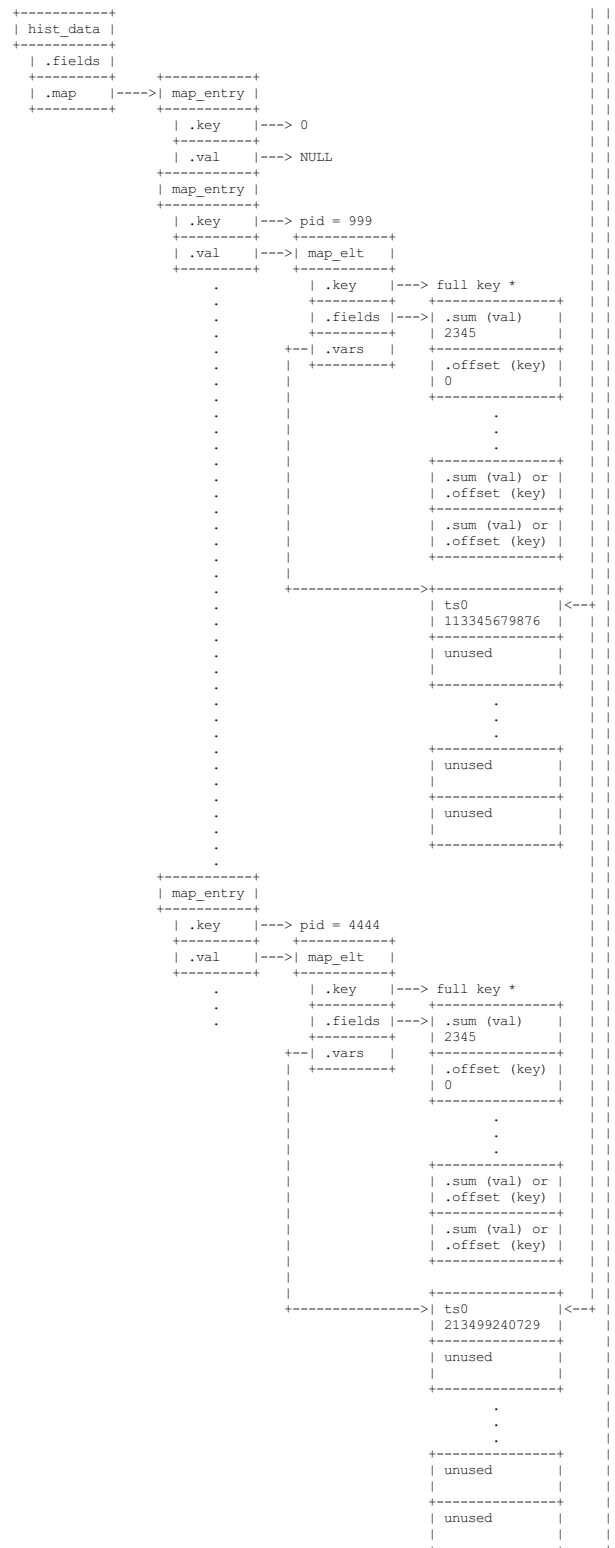
```
# echo 'hist:keys:pid:ts0=common_timestamp.usecs' >>
events/sched/sched_waking/trigger

# echo 'hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0' >>
events/sched/sched_switch/trigger
```

ned waking histogram -----:



The diagram attempts to show the relationship between the `||| hist_data.fields[]` and the `map_elt.fields[]` and `map_elt.vars[]` with `|||` the links drawn between diagrams. For each of the `map_elts`, you can `|||` see that the `.fields[]` members point to the `.sum` or `.offset` of a key `|||` or val and the `.vars[]` members point to the value of a variable. The `|||` arrows between the two diagrams show the linkages between those `|||` tracing `map` members and the field definitions in the corresponding `|||` `hist_data.fields[]` members:



Associated with the new var_ref field are a couple of new hist_field[] members, var_hist_data and var_ref_idx. For a variable reference, the var_hist_data goes with the var_idx, which together uniquely identify a particular variable on a particular histogram. The var_ref_idx is just the index into the var_ref_vals[] array that caches the values of each variable whenever a hist trigger is updated. Those resulting values are then finally accessed by other code such as trace action[] code that uses the var_ref_idx values to assign param values.

```
# echo 'hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0' >>
events/sched/sched_switch/trigger

+-----+
| hist_data |
+-----+
| .fields[] | --> val = hitcount |
+-----+
| .map | +-----+
+-----+ | .size |
+-----+ | .offset |
+-+-.var_refs[] | +-----+
+-----+ | .fn() |
+-----+ | .flags |
| $ts0 | <---+ | .var.idx |
+-----+ | .var.hist_data |
+-----+ | .var_ref_idx |
+-----+ | var = wakeup_lat |
+-----+ | .size |
| . | +-----+
| . | +-----+
+-----+ | .offset |
+-----+ | .fn() |
+-----+ | .flags & FL_VAR |
+-----+ | .var.idx |
+-----+ | .var.hist_data |
+-----+ | .var_ref_idx |
+-----+ | . |
+-----+ | . |
+-----+ | key = pid | <--- n_vals |
+-----+ | .size |
+-----+ | .offset |
+-----+ | .fn() |
+-----+ | .flags |
+-----+ | .var.idx |
+-----+ | <--- n_fields |
| unused |
+-----+
+-----+
+-----+
+-----+
+-----+
+-----+
n_keys = n_fields - n_vals

+-----+
+-----+> | var_ref = $ts0 |
+-----+ | .size |
+-----+ | .offset |
+-----+ | .fn() |
+-----+ | .flags & FL_VAR_REF |
+-----+ | .var.idx |
+-----+ | .var.hist_data |
+-----+ | .var_ref_idx |
```

```
hist_data = struct hist_trigger_data
hist_data.fields = struct hist_field
fn = hist_field_fn_t
FL KEY = HIST_FIELD_FL_KEY
FL_VAR = HIST_FIELD_FL_VAR
FL_VAR REF = HIST_FIELD_FL_VAR_REF
```

So, in order to handle an event for the `sched_switch` histogram, because we have a reference to a variable on another histogram, we need to resolve all variable references first. This is done via the `resolve_var_ref()` calls made from `event_hist_trigger()`. What this does is grabs the `var_refs[]` array from the hist data representing the `sched_switch` histogram. For each one of those, the referenced variable's `var.hist_data` along with the current key is used to look up the corresponding `tracing_map_elt` in that histogram. Once found, the referenced variable's `var.idx` is used to look up the variable's value using `tracing_map_read(var_elt, var.idx)`, which yields the value of the variable for that element, `ts0` in the case above. Note that both the `hist_fields` representing both the variable and the variable reference have the same `var.idx`, so this is straightforward.

This example creates a variable on the `sched_waking` event, `ts0`, and uses it in the `sched_switch` trigger. The `sched_switch` trigger also creates its own variable, `wakeup lat`, but nothing yet uses it:

```
# echo 'hist:keys=pid:ts0=common_timestamp.usecs' >> events/sched/sched_waking/trigger
```

```
# echo 'hist:keys=next pid:wakeup lat=common_timestamp.usecs-$ts0' >> events/sched/sched_switch/trigger
```

```
# cat events/sched/sched waking/hist debug
```

```
# event histogram
#
# trigger info: hist:keys=pid:vals=hitcount:ts0=common_timestamp.usecs:sort=hitcount:size=2048:clock=global [active]
#

hist_data: 000000009536f554

n_vals: 2
n_keys: 1
n_fields: 3

val fields:

hist_data->fields[0]:
  flags:
    VAL: HIST_FIELD_FL_HITCOUNT
    type: u64
    size: 8
    is_signed: 0

hist_data->fields[1]:
  flags:
    HIST_FIELD_FL_VAR
    var.name: ts0
    var.idx (into tracing_map_elt.vars[]): 0
    type: u64
    size: 8
    is_signed: 0

key fields:

hist_data->fields[2]:
  flags:
    HIST_FIELD_FL_KEY
    ftrace_event_field name: pid
    type: pid_t
    size: 8
    is_signed: 1
```

Moving on to the `sched_switch` trigger `hist_debug` output, in addition to the unused `wakeup_lat` variable, we see a new section displaying variable references. Variable references are displayed in a separate section because in addition to being logically separate from variables and values, they actually live in a separate `hist_data` array, `var_refs[]`.

In this example, the `sched_switch` trigger has a reference to a variable on the `sched_waking` trigger, `ts0`. Looking at the details, we can see that the `var.hist_data` value of the referenced variable matches the previously displayed `sched_waking` trigger, and the `var.idx` value matches the previously displayed `var.idx` value for that variable. Also displayed is the `var_ref_idx` value for that variable reference, which is where the value for that variable is cached for use when the trigger is invoked:

```
# cat events/sched/sched_switch/hist_debug

# event histogram
#
# trigger info: hist:keys=next_pid:vals=hitcount:wakeup_lat=common_timestamp.usecs-$ts0:sort=hitcount:size=2048:clock=global [active]
#

hist_data: 00000000f4ee8006

n_vals: 2
n_keys: 1
n_fields: 3

val fields:

hist_data->fields[0]:
  flags:
    VAL: HIST_FIELD_FL_HITCOUNT
    type: u64
    size: 8
    is_signed: 0

hist_data->fields[1]:
  flags:
    HIST_FIELD_FL_VAR
    var.name: wakeup_lat
    var.idx (into tracing_map_elt.vars[]): 0
    type: u64
    size: 0
    is_signed: 0

key fields:

hist_data->fields[2]:
  flags:
    HIST_FIELD_FL_KEY
    ftrace_event_field name: next_pid
    type: pid_t
    size: 8
    is_signed: 1

variable reference fields:

hist_data->var_refs[0]:
  flags:
    HIST_FIELD_FL_VAR_REF
    name: ts0
    var.idx (into tracing_map_elt.vars[]): 0
    var.hist_data: 000000009536f554
    var_ref_idx (into hist_data->var_refs[]): 0
    type: u64
    size: 8
    is_signed: 0
```

The commands below can be used to clean things up for the next test:

```
# echo '!hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0' >> events/sched/sched_switch/trigger
# echo '!hist:keys=pid:ts0=common_timestamp.usecs' >> events/sched/sched_waking/trigger
```

Actions and Handlers

Adding onto the previous example, we will now do something with that `wakeup_lat` variable, namely send it and another field as a synthetic event.

The `onmatch()` action below basically says that whenever we have a `sched_switch` event, if we have a matching `sched_waking` event, in this case if we have a `pid` in the `sched_waking` histogram that matches the `next_pid` field on this `sched_switch` event, we retrieve the variables specified in the `wakeup_latency()` trace action, and use them to generate a new `wakeup_latency` event into the trace stream.

Note that the way the trace handlers such as `wakeup_latency()` (which could equivalently be written `trace(wakeup_latency,$wakeup_lat,next_pid)` are implemented, the parameters specified to the trace handler must be variables. In this case, `$wakeup_lat` is obviously a variable, but `next_pid` isn't, since it's just naming a field in the `sched_switch` trace event. Since this is something that almost every `trace()` and `save()` action does, a special shortcut is implemented to allow field names to be used directly in those cases. How it works is that under the covers, a temporary variable is created for the named field, and this variable is what is actually passed to the trace handler. In the code and documentation, this type of variable is called a 'field variable'.

Fields on other trace event's histograms can be used as well. In that case we have to generate a new histogram and an unfortunately named 'synthetic_field' (the use of synthetic here has nothing to do with synthetic events) and use that special histogram field as a variable.

First, we define the wakeup latency synthetic event:

Next, the sched waking hist trigger as before:

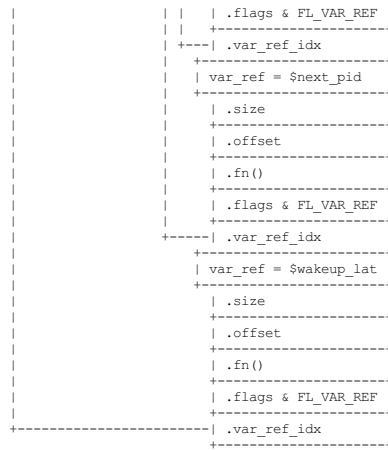
Finally, we create a hist trigger on the sched_switch event that generates a wakeup_latency() trace event. In this case we pass next_pid into the wakeup_latency synthetic event invocation, which means it will be automatically converted into a field variable:

The diagram for the `sched_switch` event is similar to previous examples but shows the additional `field_vars[]` array for `hist_data` and shows the linkages between the `field_vars` and the variables and references created to implement the field variables. The details are discussed below:

```

+-----+-----+
| hist_data |
+-----+-----+
| .fields[] | -->| val = hitcount |
+-----+-----+
| .map |
+-----+-----+
+---| .field_vars[] |
+-----+-----+
|+---| .var_refs[] |
+-----+-----+
|| +-----+
|| |
|| | var_ref_vals[] |
|| | +-----+
|| | | $ts0 | <---+
|| | +-----+
|| | | $next_pid | <+
|| | +-----+
||+>| $wakeup_lat |
||+-----+
||| |
||| | +-----+
||| | | var = wakeup_lat |
||| | +-----+
||| | | .size |
||| | +-----+
||| | | .offset |
||| | +-----+
||| | | .fn() |
||| | +-----+
||| | | .flags & FL_VAR |
||| | +-----+
||| | | .var.idx |
||| | +-----+
||| | | .var.hist_data |
||| | +-----+
||| | | .var_ref_idx |
||| | +-----+
||| |
||| | .
||| | .
||| | .
||| | .
||| | .
||| | +-----+
||| | | field_var |
||| | +-----+
||| | | | var |
||| | | +-----+
||| | | | val |
||| | | +-----+
||| | | | field_var |
||| | | +-----+
||| | | | var |
||| | | +-----+
||| | | | val |
||| | | +-----+
||| | | .
||| | | .
||| | | .
||| | +-----+
||| | | key = pid | <--- n_vals
||| | +-----+
||| | | field_var |
||| | +-----+
||| | | | .size |
||| | | +-----+
||| | | | var | --+
||| | | +-----+
||| | | | .offset |
||| | | +-----+
||| | | | val | +|
||| | | +-----+
||| | | | .fn() |
||| | | +-----+
||| | | | .flags |
||| | | +-----+
||| | | | .var.idx |
||| | | +-----+
||| | | <--- n_fields
||| | |
||| | | n_keys = n_fields - n_vals
||| | |
||| | +-----+
||| | | var = next_pid |
||| | +-----+
||| | | | .size |
||| | | +-----+
||| | | | .offset |
||| | | +-----+
||| | | | .flags & FL_VAR |
||| | | +-----+
||| | | | .var.idx |
||| | | +-----+
||| | | | .var.hist_data |
||| | | +-----+
||| | | +---| val for next_pid |
||| | | +-----+
||| | | | .size |
||| | | +-----+
||| | | | .offset |
||| | | +-----+
||| | | | .fn() |
||| | | +-----+
||| | | | .flags |
||| | | +-----+
||| | | |
||| | | +-----+
||| | |
||| | +-----+
||| | | var_ref = $ts0 |
||| | +-----+
||| | | | .size |
||| | | +-----+
||| | | | .offset |
||| | | +-----+
||| | | | .fn() |
||| | | +-----+

```



As you can see, for a field variable, two `hist_fields` are created: one representing the variable, in this case `next_pid`, and one to actually get the value of the field from the trace stream, like a normal `val` field does. These are created separately from normal variable creation and are saved in the `hist_data->field_vars[]` array. See below for how these are used. In addition, a reference `hist_field` is also created, which is needed to reference the field variables such as `$next_pid` variable in the `trace()` action.

Note that `$wakeup_lat` is also a variable reference, referencing the value of the expression `common_timestamp-$ts0`, and so also needs to have a `hist` field entry representing that reference created.

When `hist_trigger_elt_update()` is called to get the normal key and value fields, it also calls `update_field_vars()`, which goes through each `field_var` created for the histogram, and available from `hist_data->field_vars` and calls `val->fn()` to get the data from the current trace record, and then uses the `var's` `var_idx` to set the variable at the `var_idx` offset in the appropriate `tracing_map_elt's` variable at `elt->vars[var_idx]`.

Once all the variables have been updated, `resolve_var_refs()` can be called from `event_hist_trigger()`, and not only can our `$ts0` and `$next_pid` references be resolved but the `$wakeup_lat` reference as well. At this point, the `trace()` action can simply access the values assembled in the `var_ref_vals[]` array and generate the trace event.

The same process occurs for the field variables associated with the `save()` action.

Abbreviations used in the diagram

```

hist_data = struct hist_trigger data
hist_data->fields = struct hist_field
field_var = struct field_var
fn = hist_field fn_t
FL_KEY = HIST_FIELD_FL_KEY
FL_VAR = HIST_FIELD_FL_VAR
FL_VAR_REF = HIST_FIELD_FL_VAR_REF

```

trace() action field variable test

This example adds to the previous test example by finally making use of the `wakeup_lat` variable, but in addition also creates a couple of field variables that then are all passed to the `wakeup_latency()` trace action via the `onmatch()` handler.

First, we create the `wakeup_latency` synthetic event:

```
# echo 'wakeup_latency u64 lat; pid_t pid; char comm[16]' >> synthetic_events
```

Next, the `sched_waking` trigger from previous examples:

```
# echo 'hist:keys=pid:ts0=common_timestamp.usecs' >> events/sched/sched_waking/trigger
```

Finally, as in the previous test example, we calculate and assign the `wakeup_latency` using the `$ts0` reference from the `sched_waking` trigger to the `wakeup_lat` variable, and finally use it along with a couple `sched_switch` event fields, `next_pid` and `next_comm`, to generate a `wakeup_latency` trace event. The `next_pid` and `next_comm` event fields are automatically converted into field variables for this purpose:

```
# echo 'hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0:onmatch(sched.sched_waking).wakeup_latency($wakeup_lat,next_pid,next_comm)
```

The `sched_waking_hist_debug` output shows the same data as in the previous test example:

```
# cat events/sched/sched_waking/hist_debug

# event histogram
#
# trigger info: hist:keys=pid:vals=hitcount:ts0=common_timestamp.usecs:sort=hitcount:size=2048:clock=global [active]
#

hist_data: 00000000d60ff61f

n_vals: 2
n_keys: 1
n_fields: 3

val fields:

hist_data->fields[0]:
  flags:
    VAL: HIST_FIELD_FL_HITCOUNT
  type: u64
  size: 8
  is_signed: 0

hist_data->fields[1]:
  flags:
    HIST_FIELD_FL_VAR
  var.name: ts0
  var.idx (into tracing_map_elt.vars[]): 0
  type: u64
  size: 8
  is_signed: 0

key fields:

hist_data->fields[2]:
  flags:
    HIST_FIELD_FL_KEY
  ftrace_event_field name: pid
  type: pid_t
  size: 8
  is_signed: 1
```

The `sched_switch` `hist_debug` output shows the same key and value fields as in the previous test example - note that `wakeup_lat` is still in the `val` fields section, but that the new field variables are not there - although the field variables are variables, they're held separately in the `hist_data's` `field_vars[]` array. Although the field variables and the normal variables are located in separate places, you can see that the actual variable locations for those variables in the `tracing_map_elt.vars[]` do have increasing indices as expected: `wakeup_lat` takes the `var_idx = 0` slot, while the field variables for `next_pid` and `next_comm` have values `var_idx = 1`, and `var_idx = 2`. Note also that those are the same values displayed for the variable references corresponding to those variables in the variable

reference fields section. Since there are two triggers and thus two hist_data addresses, those addresses also need to be accounted for when doing the matching - you can see that the first variable refers to the 0 var.idx on the previous hist trigger (see the hist_data address associated with that trigger), while the second variable refers to the 0 var.idx on the sched_switch hist trigger, as do all the remaining variable references.

Finally, the action tracking variables section just shows the system and event name for the onmatch() handler:

```
# cat events/sched/sched_switch/hist_debug

# event histogram
# trigger info: hist:keys=next_pid:vals=hitcount:wakeup_lat=common_timestamp.usecs-$ts0:sort=hitcount:size=2048:clock=global:onmatch(sched_switch)

hist_data: 000000008f551b7

n_vals: 2
n_keys: 1
n_fields: 3

val fields:

hist_data->fields[0]:
  flags:
    VAL: HIST_FIELD_FL_HITCOUNT
  type: u64
  size: 8
  is_signed: 0

hist_data->fields[1]:
  flags:
    HIST_FIELD_FL_VAR
  var.name: wakeup_lat
  var.idx (into tracing_map_elt.vars[]): 0
  type: u64
  size: 0
  is_signed: 0

key fields:

hist_data->fields[2]:
  flags:
    HIST_FIELD_FL_KEY
  ftrace_event_field name: next_pid
  type: pid_t
  size: 8
  is_signed: 1

variable reference fields:

hist_data->var_refs[0]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: ts0
  var.idx (into tracing_map_elt.vars[]): 0
  var.hist_data: 00000000d60ff61f
  var_ref_idx (into hist_data->var_refs[]): 0
  type: u64
  size: 8
  is_signed: 0

hist_data->var_refs[1]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: wakeup_lat
  var.idx (into tracing_map_elt.vars[]): 0
  var.hist_data: 000000008f551b7
  var_ref_idx (into hist_data->var_refs[]): 1
  type: u64
  size: 0
  is_signed: 0

hist_data->var_refs[2]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: next_pid
  var.idx (into tracing_map_elt.vars[]): 1
  var.hist_data: 000000008f551b7
  var_ref_idx (into hist_data->var_refs[]): 2
  type: pid_t
  size: 4
  is_signed: 0

hist_data->var_refs[3]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: next comm
  var.idx (into tracing_map_elt.vars[]): 2
  var.hist_data: 000000008f551b7
  var_ref_idx (into hist_data->var_refs[]): 3
  type: char[16]
  size: 256
  is_signed: 0

field variables:

hist_data->field_vars[0]:

  field_vars[0].var:
    flags:
      HIST_FIELD_FL_VAR
    var.name: next_pid
    var.idx (into tracing_map_elt.vars[]): 1

  field_vars[0].val:
    ftrace_event_field name: next_pid
    type: pid_t
    size: 4
    is_signed: 1

hist_data->field_vars[1]:

  field_vars[1].var:
    flags:
      HIST_FIELD_FL_VAR
    var.name: next comm
    var.idx (into tracing_map_elt.vars[]): 2

  field_vars[1].val:
    ftrace_event_field name: next_comm
    type: char[16]
    size: 256
    is_signed: 0

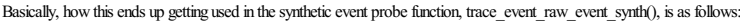
action tracking variables (for onmax()/onchange()/onmatch()):

hist_data->actions[0].match_data.event_system: sched
hist_data->actions[0].match_data.event: sched_waking
```

The commands below can be used to clean things up for the next test:

action_data and the trace() action

The values in the `var_ref_vals[]` array, however, don't necessarily follow the same ordering as the synthetic event params. To address that, struct `action_data` contains another array, `var_ref_idx[]` that maps the trace action params to the `var_ref_vals[]` values. Below is a diagram illustrating that for the `wakeup_latency()` synthetic event:



action data and the onXXX() handlers

Typically, the `onmax()` or `onchange()` handlers are used in conjunction with the `save()` and `snapshot()` actions. For example:

or:

save() action field variable test

As in previous test examples, we set up the sched_waking trigger:

In this case, however, we set up the `sched_switch` trigger to save some `sched_switch` field values whenever we hit a new maximum latency. For both the `onmax()` handler and `save()` action, variables will be created, which we can use the `hist debug` files to examine:

The sched_waking_hist debug output shows the same data as in the previous test examples:

The output of the `sched_switch` trigger shows the same `val` and `key` values as before, but also shows a couple new sections.

Finally, in the new 'save action variables' section, we can see that the 4 params to the `save()` function have resulted in 4 field variables being created for the purposes of saving the values of the named fields when the max is hit. These variables are kept in a separate `save_vars[]` array off of `hist_data`, so are displayed in a separate section:

```
# event histogram
```

```

#
# trigger info: hist:keys=next_pid:vals=hitcount:wakeup_lat=common_timestamp.usecs-$ts0:sort=hitcount:size=2048:clock=global:onmax($wakeup)
#

hist_data: 000000057bcd28d

n_vals: 2
n_keys: 1
n_fields: 3

val fields:

hist_data->fields[0]:
  flags:
    VAL: HIST_FIELD_FL_HITCOUNT
  type: u64
  size: 8
  is_signed: 0

hist_data->fields[1]:
  flags:
    HIST_FIELD_FL_VAR
  var.name: wakeup_lat
  var.idx (into tracing_map_elt.vars[]): 0
  type: u64
  size: 0
  is_signed: 0

key fields:

hist_data->fields[2]:
  flags:
    HIST_FIELD_FL_KEY
  ftrace_event_field name: next_pid
  type: pid_t
  size: 8
  is_signed: 1

variable reference fields:

hist_data->var_refs[0]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: ts0
  var.idx (into tracing_map_elt.vars[]): 0
  var.hist_data: 0000000e6290f48
  var_ref_idx (into hist_data->var_refs[]): 0
  type: u64
  size: 8
  is_signed: 0

hist_data->var_refs[1]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: wakeup_lat
  var.idx (into tracing_map_elt.vars[]): 0
  var.hist_data: 000000057bcd28d
  var_ref_idx (into hist_data->var_refs[]): 1
  type: u64
  size: 0
  is_signed: 0

action tracking variables (for onmax()/onchange()/onmatch()):

hist_data->actions[0].track_data.var_ref:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: wakeup_lat
  var.idx (into tracing_map_elt.vars[]): 0
  var.hist_data: 000000057bcd28d
  var_ref_idx (into hist_data->var_refs[]): 1
  type: u64
  size: 0
  is_signed: 0

hist_data->actions[0].track_data.track_var:
  flags:
    HIST_FIELD_FL_VAR
  var.name: __max
  var.idx (into tracing_map_elt.vars[]): 1
  type: u64
  size: 8
  is_signed: 0

save action variables (save() params):

hist_data->save_vars[0]:

  save_vars[0].var:
  flags:
    HIST_FIELD_FL_VAR
  var.name: next_comm
  var.idx (into tracing_map_elt.vars[]): 2

  save_vars[0].val:
  ftrace_event_field name: next_comm
  type: Char[16]
  size: 256
  is_signed: 0

hist_data->save_vars[1]:

  save_vars[1].var:
  flags:
    HIST_FIELD_FL_VAR
  var.name: prev_pid
  var.idx (into tracing_map_elt.vars[]): 3

  save_vars[1].val:
  ftrace_event_field name: prev_pid
  type: pid_t
  size: 4
  is_signed: 1

hist_data->save_vars[2]:

  save_vars[2].var:
  flags:
    HIST_FIELD_FL_VAR
  var.name: prev_prio
  var.idx (into tracing_map_elt.vars[]): 4

  save_vars[2].val:
  ftrace_event_field name: prev_prio
  type: int
  size: 4
  is_signed: 1

hist_data->save_vars[3]:

```

```

save_vars[3].var:
flags:
  HIST_FIELD_FL_VAR
var.name: prev_comm
var.idx (into tracing_map_elt.vars[]): 5

save_vars[3].val:
ftrace_event_field name: prev_comm
type: char[16]
size: 256
is_signed: 0

```

The commands below can be used to clean things up for the next test:

```

# echo '!hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0:onmax($wakeup_lat).save(next_comm,prev_pid,prev_prio,prev_comm)' >> events/sched/sched_waking/trigger

# echo '!hist:keys=pid:ts0=common_timestamp.usecs' >> events/sched/sched_waking/trigger

```

A couple special cases

While the above covers the basics of the histogram internals, there are a couple of special cases that should be discussed, since they tend to create even more confusion. Those are field variables on other histograms, and aliases, both described below through example tests using the hist_debug files.

Test of field variables on other histograms

This example is similar to the previous examples, but in this case, the sched_switch trigger references a hist trigger field on another event, namely the sched_waking event. In order to accomplish this, a field variable is created for the other event, but since an existing histogram can't be used, as existing histograms are immutable, a new histogram with a matching variable is created and used, and we'll see that reflected in the hist_debug output shown below.

First, we create the wakeup_latency synthetic event. Note the addition of the prio field:

```

# echo 'wakeup_latency u64 lat; pid_t pid; int prio' >> synthetic_events

```

As in previous test examples, we set up the sched_waking trigger:

```

# echo 'hist:keys=pid:ts0=common_timestamp.usecs' >> events/sched/sched_waking/trigger

```

Here we set up a hist trigger on sched_switch to send a wakeup_latency event using an onmatch handler naming the sched_waking event. Note that the third param being passed to the wakeup_latency() is prio, which is a field name that needs to have a field variable created for it. There isn't however any prio field on the sched_switch event so it would seem that it wouldn't be possible to create a field variable for it. The matching sched_waking event does have a prio field, so it should be possible to make use of it for this purpose. The problem with that is that it's not currently possible to define a new variable on an existing histogram, so it's not possible to add a new prio field variable to the existing sched_waking histogram. It is however possible to create an additional new 'matching' sched_waking histogram for the same event, meaning that it uses the same key and filters, and define the new prio field variable on that.

Here's the sched_switch trigger:

```

# echo 'hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0:onmatch(sched.sched_waking).wakeup_latency($wakeup_lat,next_pid,prio)' >> events/sched/sched_switch/trigger

```

And here's the output of the hist_debug information for the sched_waking hist trigger. Note that there are two histograms displayed in the output: the first is the normal sched_waking histogram we've seen in the previous examples, and the second is the special histogram we created to provide the prio field variable.

Looking at the second histogram below, we see a variable with the name synthetic_prio. This is the field variable created for the prio field on that sched_waking histogram

```

# cat events/sched/sched_waking/hist_debug

# event histogram
#
# trigger info: hist:keys=pid:vals=hitcount:ts0=common_timestamp.usecs:sort=hitcount:size=2048:clock=global [active]
#

hist_data: 00000000349570e4

n_vals: 2
n_keys: 1
n_fields: 3

val fields:

hist_data->fields[0]:
  flags:
    VAL: HIST_FIELD_FL_HITCOUNT
    type: u64
    size: 8
    is_signed: 0

hist_data->fields[1]:
  flags:
    HIST_FIELD_FL_VAR
    var.name: ts0
    var.idx (into tracing_map_elt.vars[]): 0
    type: u64
    size: 8
    is_signed: 0

key fields:

hist_data->fields[2]:
  flags:
    HIST_FIELD_FL_KEY
    ftrace_event_field name: pid
    type: pid_t
    size: 8
    is_signed: 1

# event histogram
#
# trigger info: hist:keys=pid:vals=hitcount:synthetic_prio=prio:sort=hitcount:size=2048 [active]
#

hist_data: 000000006920cf38

n_vals: 2
n_keys: 1
n_fields: 3

val fields:

hist_data->fields[0]:
  flags:
    VAL: HIST_FIELD_FL_HITCOUNT
    type: u64
    size: 8
    is_signed: 0

hist_data->fields[1]:
  flags:
    HIST_FIELD_FL_VAR
    ftrace_event_field name: prio
    var.name: synthetic_prio
    var.idx (into tracing_map_elt.vars[]): 0

```

```

        type: int
        size: 4
        is_signed: 1

key fields:

hist_data->fields[2]:
  flags:
    HIST_FIELD_FL_KEY
  ftrace_event_field name: pid
  type: pid_t
  size: 8
  is_signed: 1

```

Looking at the sched_switch histogram below, we can see a reference to the synthetic_prio variable on sched_waking, and looking at the associated hist_data address we see that it is indeed associated with the new histogram. Note also that the other references are to a normal variable, wakeup_lat, and to a normal field variable, next_pid, the details of which are in the field variables section:

```

# cat events/sched/sched_switch/hist_debug

# event histogram
#
# trigger info: hist:keys=next_pid:vals=hitcount:wakeup_lat=common_timestamp.usecs-$ts0:sort=hitcount:size=2048:clock=global:onmatch(sched_switch)
#

hist_data: 0000000a73b67df

n_vals: 2
n_keys: 1
n_fields: 3

val fields:

hist_data->fields[0]:
  flags:
    VAL: HIST_FIELD_FL_HITCOUNT
  type: u64
  size: 8
  is_signed: 0

hist_data->fields[1]:
  flags:
    HIST_FIELD_FL_VAR
  var.name: wakeup_lat
  var.idx (into tracing_map_elt.vars[]): 0
  type: u64
  size: 0
  is_signed: 0

key fields:

hist_data->fields[2]:
  flags:
    HIST_FIELD_FL_KEY
  ftrace_event_field name: next_pid
  type: pid_t
  size: 8
  is_signed: 1

variable reference fields:

hist_data->var_refs[0]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: ts0
  var.idx (into tracing_map_elt.vars[]): 0
  var.hist_data: 00000000349570e4
  var_ref_idx (into hist_data->var_refs[]): 0
  type: u64
  size: 8
  is_signed: 0

hist_data->var_refs[1]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: wakeup_lat
  var.idx (into tracing_map_elt.vars[]): 0
  var.hist_data: 0000000a73b67df
  var_ref_idx (into hist_data->var_refs[]): 1
  type: u64
  size: 0
  is_signed: 0

hist_data->var_refs[2]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: next_pid
  var.idx (into tracing_map_elt.vars[]): 1
  var.hist_data: 0000000a73b67df
  var_ref_idx (into hist_data->var_refs[]): 2
  type: pid_t
  size: 4
  is_signed: 0

hist_data->var_refs[3]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: synthetic_prio
  var.idx (into tracing_map_elt.vars[]): 0
  var.hist_data: 000000006920cf38
  var_ref_idx (into hist_data->var_refs[]): 3
  type: int
  size: 4
  is_signed: 1

field variables:

hist_data->field_vars[0]:

  field_vars[0].var:
  flags:
    HIST_FIELD_FL_VAR
  var.name: next_pid
  var.idx (into tracing_map_elt.vars[]): 1

  field_vars[0].val:
  ftrace_event_field name: next_pid
  type: pid_t
  size: 4
  is_signed: 1

action tracking variables (for onmax()/onchange()/onmatch()):

hist_data->actions[0].match_data.event_system: sched
hist_data->actions[0].match_data.event: sched_waking

```

The commands below can be used to clean things up for the next test:

```

# echo '!hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0:onmatch(sched.sched_waking).wakeup_latency($wakeup_lat,next_pid,prio)'

```

```
# echo '!hist:keys=pid:ts0=common_timestamp.usecs' >> events/sched/sched_waking/trigger

# echo '!wakeup_latency u64 lat; pid_t pid; int prio' >> synthetic_events
```

Alias test

This example is very similar to previous examples, but demonstrates the alias flag.

First, we create the `wakeup_latency` synthetic event:

```
# echo 'wakeup_latency u64 lat; pid_t pid; char comm[16]' >> synthetic_events
```

Next, we create a `sched_waking` trigger similar to previous examples, but in this case we save the `pid` in the `waking_pid` variable:

```
# echo 'hist:keys=pid:waking_pid=pid:ts0=common_timestamp.usecs' >> events/sched/sched_waking/trigger
```

For the `sched_switch` trigger, instead of using `$waking_pid` directly in the `wakeup_latency` synthetic event invocation, we create an alias of `$waking_pid` named `$woken_pid`, and use that in the synthetic event invocation instead:

```
# echo 'hist:keys=next_pid:woken_pid=$waking_pid:wakeup_lat=common_timestamp.usecs-$ts0:ormatch(sched.sched_waking).wakeup_latency($woken_pid)' >> events/sched/sched_switch/trigger
```

Looking at the `sched_waking` `hist_debug` output, in addition to the normal fields, we can see the `waking_pid` variable:

```
# cat events/sched/sched_waking/hist_debug

# event histogram
#
# trigger info: hist:keys=pid:vals=hitcount:waking_pid=pid,ts0=common_timestamp.usecs:sort=hitcount:size=2048:clock=global [active]
#

hist_data: 00000000a250528c

n_vals: 3
n_keys: 1
n_fields: 4

val fields:

hist_data->fields[0]:
  flags:
    VAL: HIST_FIELD_FL_HITCOUNT
    type: u64
    size: 8
    is_signed: 0

hist_data->fields[1]:
  flags:
    HIST_FIELD_FL_VAR
    ftrace_event_field name: pid
    var.name: waking_pid
    var.idx (into tracing_map_elt.vars[]): 0
    type: pid_t
    size: 4
    is_signed: 1

hist_data->fields[2]:
  flags:
    HIST_FIELD_FL_VAR
    var.name: ts0
    var.idx (into tracing_map_elt.vars[]): 1
    type: u64
    size: 8
    is_signed: 0

key fields:

hist_data->fields[3]:
  flags:
    HIST_FIELD_FL_KEY
    ftrace_event_field name: pid
    type: pid_t
    size: 8
    is_signed: 1
```

The `sched_switch` `hist_debug` output shows that a variable named `woken_pid` has been created but that it also has the `HIST_FIELD_FL_ALIAS` flag set. It also has the `HIST_FIELD_FL_VAR` flag set, which is why it appears in the `val` field section.

Despite that implementation detail, an alias variable is actually more like a variable reference; in fact it can be thought of as a reference to a reference. The implementation copies the `var_ref->fn()` from the variable reference being referenced, in this case, the `waking_pid` `fn()`, which is `hist_field_var_ref()` and makes that the `fn()` of the alias. The `hist_field_var_ref()` `fn()` requires the `var_ref_idx` of the variable reference it's using, so `waking_pid`'s `var_ref_idx` is also copied to the alias. The end result is that when the value of alias is retrieved, in the end it just does the same thing the original reference would have done and retrieves the same value from the `var_ref_vals[]` array. You can verify this in the output by noting that the `var_ref_idx` of the alias, in this case `woken_pid`, is the same as the `var_ref_idx` of the reference, `waking_pid`, in the variable reference fields section.

Additionally, once it gets that value, since it is also a variable, it then saves that value into its `var.idx`. So the `var.idx` of the `woken_pid` alias is 0, which it fills with the value from `var_ref_idx 0` when its `fn()` is called to update itself. You'll also notice that there's a `woken_pid` `var_ref` in the variable refs section. That is the reference to the `woken_pid` alias variable, and you can see that it retrieves the value from the same `var.idx` as the `woken_pid` alias, 0, and then in turn saves that value in its own `var_ref_idx` slot, 3, and the value at this position is finally what gets assigned to the `$woken_pid` slot in the trace event invocation.

```
# cat events/sched/sched_switch/hist_debug

# event histogram
#
# trigger info: hist:keys=next_pid:vals=hitcount:woken_pid=$waking_pid,wakeup_lat=common_timestamp.usecs-$ts0:sort=hitcount:size=2048:clock=global [active]
#

hist_data: 0000000055d65ed0

n_vals: 3
n_keys: 1
n_fields: 4

val fields:

hist_data->fields[0]:
  flags:
    VAL: HIST_FIELD_FL_HITCOUNT
    type: u64
    size: 8
    is_signed: 0

hist_data->fields[1]:
  flags:
    HIST_FIELD_FL_VAR
    HIST_FIELD_FL_ALIAS
    var.name: woken_pid
    var.idx (into tracing_map_elt.vars[]): 0
    var_ref_idx (into hist_data->var_refs[]): 0
    type: pid_t
    size: 4
    is_signed: 1

hist_data->fields[2]:
  flags:
    HIST_FIELD_FL_VAR
    var.name: wakeup_lat
```

```

var.idx (into tracing_map_elt.vars[]): 1
type: u64
size: 0
is_signed: 0

key fields:

hist_data->fields[3]:
  flags:
    HIST_FIELD_FL_KEY
  ftrace_event_field name: next_pid
  type: pid_t
  size: 8
  is_signed: 1

variable reference fields:

hist_data->var_refs[0]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: waking_pid
  var.idx (into tracing_map_elt.vars[]): 0
  var.hist_data: 00000000a250528c
  var_ref_idx (into hist_data->var_refs[]): 0
  type: pid_t
  size: 4
  is_signed: 1

hist_data->var_refs[1]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: ts0
  var.idx (into tracing_map_elt.vars[]): 1
  var.hist_data: 00000000a250528c
  var_ref_idx (into hist_data->var_refs[]): 1
  type: u64
  size: 8
  is_signed: 0

hist_data->var_refs[2]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: wakeup_lat
  var.idx (into tracing_map_elt.vars[]): 1
  var.hist_data: 000000005d65ed0
  var_ref_idx (into hist_data->var_refs[]): 2
  type: u64
  size: 0
  is_signed: 0

hist_data->var_refs[3]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: woken_pid
  var.idx (into tracing_map_elt.vars[]): 0
  var.hist_data: 000000005d65ed0
  var_ref_idx (into hist_data->var_refs[]): 3
  type: pid_t
  size: 4
  is_signed: 1

hist_data->var_refs[4]:
  flags:
    HIST_FIELD_FL_VAR_REF
  name: next_comm
  var.idx (into tracing_map_elt.vars[]): 2
  var.hist_data: 000000005d65ed0
  var_ref_idx (into hist_data->var_refs[]): 4
  type: char[16]
  size: 256
  is_signed: 0

field variables:

hist_data->field_vars[0]:

  field_vars[0].var:
    flags:
      HIST_FIELD_FL_VAR
    var.name: next_comm
    var.idx (into tracing_map_elt.vars[]): 2

  field_vars[0].val:
    ftrace_event_field name: next_comm
    type: char[16]
    size: 256
    is_signed: 0

action tracking variables (for onmax()/onchange()/onmatch()):

hist_data->actions[0].match_data.event_system: sched
hist_data->actions[0].match_data.event: sched_waking

```

The commands below can be used to clean things up for the next test:

```

# echo '!hist:keys=next_pid:woken_pid=$waking_pid:wakeup_lat=common_timestamp.usecs-$ts0:onmatch(sched,sched_waking).wakeup_latency($wake

# echo '!hist:keys=pid:ts0=common_timestamp.usecs' >> events/sched/sched_waking/trigger

# echo '!wakeup_latency u64 lat; pid_t pid; char comm[16]' >> synthetic_events

```