Hist Triggers in Linux 4.7

08 Jun 2016

Hist triggers is a new tracing feature that recently landed in Linux 4.7-rc1. It allows the creation of custom, efficient, in-kernel histograms. Cue some screenshots!

syscalls by process name and PID

```
# echo 'hist:key=common_pid.execname' > \
      /sys/kernel/debug/tracing/events/raw_syscalls/sys_enter/trigger
[...wait some seconds...]
# cat /sys/kernel/debug/tracing/events/raw_syscalls/sys_enter/hist
# event histogram
#
# trigger info: hist:keys=common_pid.execname:vals=hitcount:sort=hitcount:size=2048 [active]
#

{ common_pid: sshd            [     28089] } hitcount:          5
{ common_pid: jps             [     27941] } hitcount:         15
[...]
{ common_pid: bash            [     32754] } hitcount:        833
{ common_pid: supervise       [      2060] } hitcount:       1824
{ common_pid: supervise       [      2062] } hitcount:       1824
{ common_pid: supervise       [      2064] } hitcount:       1824
{ common_pid: dumpsystemstats [     27909] } hitcount:       2691
{ common_pid: sshd            [     32745] } hitcount:       3761
{ common_pid: jps             [     27914] } hitcount:       3957
{ common_pid: snmpd           [      1617] } hitcount:       4854
{ common_pid: dumpsystemstats [     27940] } hitcount:       9671
{ common_pid: readproctitle   [      2054] } hitcount:      19296
{ common_pid: dumpsystemstats [     27926] } hitcount:      51386

Totals:
    Hits: 219519
    Entries: 764
    Dropped: 0
# echo '!hist:key=common_pid.execname' > \
      /sys/kernel/debug/tracing/events/raw_syscalls/sys_enter/trigger
```

The output shows that during tracing, the `dumpsystemstats` process (PID 27926) did 51,386 syscalls.

The three commands used were:

1. **echo 'hist:*config*' > *probe*/trigger**: set the histogram *config* for the given *probe*, and enable tracing
2. **cat *probe*/hist**: prints the current histogram counts
3. **echo '!hist:*config*' > *probe*/trigger**: disable tracing

If this interface seems new and bizarre to you, then you might also be new to the little-known Linux ftrace tracer (now just called "trace"), which has existed since Linux 2.6.27. This is how trace operates: echoing funny strings to /sys locations. Hist triggers is an addition to trace.

I use trace frequently, and still find the interface a bit bizarre, but it's not really a problem. I'm almost always using it via a front-end wrapper, like my perf-tools or trace-cmd.

For perf-tools I wrote syscount, which can do the same summary as above: syscalls by process. But it used older Linux capabilities, where I had to dump all syscall events to a perf.data file and post-process in user space, costing much overhead. Post Linux 4.7, I can use hist triggers, making it cheap to instrument.

syscall read() returned size and process name and PID

```
# echo 'hist:key=common_pid.execname,ret' > \
      /sys/kernel/debug/tracing/events/syscalls/sys_exit_read/trigger
# cat /sys/kernel/debug/tracing/events/syscalls/sys_exit_read/hist
[...]
{ common_pid: snmpd           [      1617], ret:          5 } hitcount:          8
{ common_pid: sshd            [     32745], ret:          1 } hitcount:          8
{ common_pid: irqbalance      [      1189], ret:       1024 } hitcount:         10
{ common_pid: supervise       [      2062], ret: 18446744073709551605 } hitcount:          14
{ common_pid: supervise       [      2064], ret: 18446744073709551605 } hitcount:          14
{ common_pid: supervise       [      2060], ret: 18446744073709551605 } hitcount:          14
{ common_pid: sshd            [     32745], ret:         36 } hitcount:         18
{ common_pid: bash            [     32754], ret:          1 } hitcount:         18
{ common_pid: readproctitle   [      2054], ret:          1 } hitcount:       1407
[...]
# echo '!hist:key=common_pid.execname,ret' > \
      /sys/kernel/debug/tracing/events/syscalls/sys_exit_read/trigger
```

So `readproctitle` is doing a lot of 1 byte reads. Note the large ret value, 18446744073709551605, which is likely -1's (errors). (Hist triggers needs a way to print these as signed decimal, not just unsigned.)

syscall total read() returned bytes by process name and PID

```
# echo 'hist:key=common_pid.execname:values=ret:sort=ret if ret >= 0' \
    > /sys/kernel/debug/tracing/events/syscalls/sys_exit_read/trigger
# cat /sys/kernel/debug/tracing/events/syscalls/sys_exit_read/hist
[...]
{ common_pid: bash            [     16608] } hitcount:          4  ret:      11722
{ common_pid: bash            [     16616] } hitcount:          4  ret:      12386
{ common_pid: bash            [     16617] } hitcount:          4  ret:      12469
{ common_pid: irqbalance      [      1189] } hitcount:         36  ret:      21702
{ common_pid: snmpd           [      1617] } hitcount:         75  ret:      22078
{ common_pid: sshd            [     32745] } hitcount:        329  ret:     165710
[...]
# echo '!hist:key=common_pid.execname:values=ret:sort=ret if ret >= 0' \
    > /sys/kernel/debug/tracing/events/syscalls/sys_exit_read/trigger
```

I'm now doing much more than just counting a custom key. Here I'm summing the ret value (return value), and also filtering to only sum positive ret values (successful reads).

kernel stacks issuing disk I/O

```
# echo 'hist:key=stacktrace' > \
      /sys/kernel/debug/tracing/events/block/block_rq_insert/trigger
# cat /sys/kernel/debug/tracing/events/block/block_rq_insert/hist
# event histogram
#
# trigger info: hist:keys=stacktrace:vals=hitcount:sort=hitcount:size=2048 [active]
#

{ stacktrace:
         blk_mq_insert_requests+0x142/0x1b0
         blk_mq_flush_plug_list+0x127/0x140
         blk_flush_plug_list+0xc7/0x220
         blk_finish_plug+0x2c/0x40
         wb_writeback+0x18b/0x2f0
         wb_workfn+0xfd/0x3c0
         process_one_work+0x153/0x3f0
         worker_thread+0x12b/0x4b0
         kthread+0xc9/0xe0
         ret_from_fork+0x1f/0x40
} hitcount:          1
{ stacktrace:
         __blk_mq_insert_request+0x9e/0xd0
         blk_mq_insert_request+0x88/0xc0
         blk_mq_requeue_work+0xd0/0x120
         process_one_work+0x153/0x3f0
         worker_thread+0x12b/0x4b0
         kthread+0xc9/0xe0
         ret_from_fork+0x1f/0x40
} hitcount:          1
{ stacktrace:
         blk_mq_insert_requests+0x142/0x1b0
         blk_mq_flush_plug_list+0x127/0x140
         blk_flush_plug_list+0xc7/0x220
         blk_finish_plug+0x2c/0x40
         __do_page_cache_readahead+0x182/0x220
         ondemand_readahead+0x135/0x260
         page_cache_sync_readahead+0x31/0x50
         generic_file_read_iter+0x4c8/0x780
         __vfs_read+0xbd/0x110
         vfs_read+0x8e/0x140
         kernel_read+0x41/0x60
         prepare_binprm+0xe6/0x200
         do_execveat_common.isra.34+0x457/0x6e0
         SyS_execve+0x3a/0x50
         do_syscall_64+0x69/0x110
         return_from_SYSCALL_64+0x0/0x6a
} hitcount:          2
{ stacktrace:
         blk_mq_insert_requests+0x142/0x1b0
         blk_mq_flush_plug_list+0x127/0x140
         blk_flush_plug_list+0xc7/0x220
         blk_finish_plug+0x2c/0x40
         __do_page_cache_readahead+0x182/0x220
         filemap_fault+0x406/0x500
         ext4_filemap_fault+0x36/0x50
         __do_fault+0x5e/0xd0
         handle_mm_fault+0xb98/0x1d20
         __do_page_fault+0x1e0/0x4d0
         do_page_fault+0x30/0x80
         page_fault+0x28/0x30
         clear_user+0x2b/0x40
         padzero+0x24/0x40
         load_elf_binary+0x8da/0x1650
         search_binary_handler+0x9e/0x1e0
} hitcount:          2
{ stacktrace:
         __blk_mq_insert_request+0x9e/0xd0
         blk_sq_make_request+0x2af/0x3d0
```

```
        generic_make_request+0xe1/0x1a0
        submit_bio+0x68/0x130
        submit_bh_wbc+0x12f/0x160
        submit_bh+0x12/0x20
        journal_submit_commit_record+0x1c7/0x1f0
        jbd2_journal_commit_transaction+0xfce/0x1860
        kjournald2+0xbb/0x230
        kthread+0xc9/0xe0
        ret_from_fork+0x1f/0x40
} hitcount:          3
{ stacktrace:
        __blk_mq_insert_request+0x9e/0xd0
        blk_mq_insert_request+0x88/0xc0
        blk_mq_requeue_work+0xf6/0x120
        process_one_work+0x153/0x3f0
        worker_thread+0x12b/0x4b0
        kthread+0xc9/0xe0
        ret_from_fork+0x1f/0x40
} hitcount:          3
{ stacktrace:
        blk_mq_insert_requests+0x142/0x1b0
        blk_mq_flush_plug_list+0x127/0x140
        blk_flush_plug_list+0xc7/0x220
        blk_finish_plug+0x2c/0x40
        jbd2_journal_commit_transaction+0xbf5/0x1860
        kjournald2+0xbb/0x230
        kthread+0xc9/0xe0
        ret_from_fork+0x1f/0x40
} hitcount:          3
{ stacktrace:
        blk_mq_insert_requests+0x142/0x1b0
        blk_mq_flush_plug_list+0x127/0x140
        blk_flush_plug_list+0xc7/0x220
        blk_finish_plug+0x2c/0x40
        generic_writepages+0x4d/0x60
        blkdev_writepages+0xe/0x10
        do_writepages+0x1e/0x30
        __filemap_fdatawrite_range+0xaa/0xf0
        filemap_fdatawrite+0x1f/0x30
        fdatawrite_one_bdev+0x16/0x20
        iterate_bdevs+0xe9/0x130
        sys_sync+0x63/0x90
        entry_SYSCALL_64_fastpath+0x1e/0xa8
} hitcount:         14
{ stacktrace:
        blk_mq_insert_requests+0x142/0x1b0
        blk_mq_flush_plug_list+0x127/0x140
        blk_flush_plug_list+0xc7/0x220
        blk_finish_plug+0x2c/0x40
        ext4_writepages+0x4db/0xce0
        do_writepages+0x1e/0x30
        __filemap_fdatawrite_range+0xaa/0xf0
        filemap_flush+0x1c/0x20
        ext4_alloc_da_blocks+0x2c/0x70
        ext4_rename+0x647/0x8a0
        ext4_rename2+0x1d/0x30
        vfs_rename+0x4aa/0x7f0
        SyS_rename+0x345/0x3a0
        entry_SYSCALL_64_fastpath+0x1e/0xa8
} hitcount:         72

Totals:
    Hits: 101
    Entries: 9
    Dropped: 0
# echo '!hist:key=stacktrace' > \
      /sys/kernel/debug/tracing/events/block/block_rq_insert/trigger
```

This is pretty useful for investigating disk I/O without an obvious reason, which can originate from an asynchronous kernel path. This is not only identifying the different paths, but doing so efficiently: using the entire stack trace as a key for the histogram, and counting it in kernel context.

Other fields can be added to that key, so one could count not just code paths to a function, but also other tracepoint fields.

user-level malloc()s by process name and PID

Now for a user-level example. I'll count which processes and PIDs are calling the libc malloc() routine:

```
# perf probe -x /lib/x86_64-linux-gnu/libc.so.6 malloc
Added new event:
  probe_libc:malloc    (on malloc in /lib/x86_64-linux-gnu/libc-2.19.so)

You can now use it in all perf tools, such as:

    perf record -e probe_libc:malloc -aR sleep 1
# echo 'hist:key=common_pid.execname' > \
      /sys/kernel/debug/tracing/events/probe_libc/malloc/trigger
# cat /sys/kernel/debug/tracing/events/probe_libc/malloc/hist
[...]
{ common_pid: chown              [      4663] } hitcount:         86
{ common_pid: chown              [      4633] } hitcount:         86
{ common_pid: chown              [      4567] } hitcount:         86
{ common_pid: chown              [      4575] } hitcount:         86
{ common_pid: chown              [      4553] } hitcount:         86
{ common_pid: chown              [      4605] } hitcount:         86
{ common_pid: chown              [      4562] } hitcount:         86
{ common_pid: chown              [      4541] } hitcount:         86
{ common_pid: ls                 [      4649] } hitcount:        160
{ common_pid: snmpd              [      1617] } hitcount:        166
{ common_pid: tar                [      4563] } hitcount:     536291
[...]
# echo '!hist:key=common_pid.execname' >
      /sys/kernel/debug/tracing/events/probe_libc/malloc/trigger
# perf probe --del probe_libc:malloc
Removed event: probe_libc:malloc
```

While tracing, the tar command called malloc() 536,291 times.

For this example, I borrowed the perf command to create the dynamic tracing probe of malloc(), and then hist triggers to instrument that probe. I didn't need to use perf: I could have set this all up via echo and /sys, but perf has better error checking.

Kernel-level dynamic tracing works too. And I can pull in arguments to functions, and their return values, and use them as keys in the histogram or sum them as values.

More info & warnings

This post demonstrates maybe one tenth of what hist triggers can do. To see more functionality, browse the official trace/events.txt documentation and search for "hist:".

This is all coming in Linux 4.7, although it is not currently enabled by default: you need to enable **CONFIG_HIST_TRIGGERS**.

WARNING: since hist triggers is all new code, so it would be wise to stress test it in the lab before any real use.

There's also the usual warnings about tracing: there will be overhead relative to the event rate multiplied by the event cost. Trace is a fast framework, and for some rough testing I was seeing 0.25 us overhead for counting kernel tracepoints, which is pretty fast. But, if you're doing millions of events per second, then 0.25 us can start to add up. User-level will be more costly.

Finally, the very last line of hist output has "Dropped: 0". If that's non-zero, then you've overflowed the default number of key slots (2048), and events will be dropped. It can be tuned by setting :size=4096 or whatever in the histogram config.

What about BPF?

Can't enhanced BPF do this too?

Yes, if you program it to. I feel like we've been waiting for advanced tracing in Linux for years, and now two busses have arrived at the same time. This overlap concern was raised and discussed on lkml, and it was eventually deemed that hist triggers was different enough to be included.

BPF can do a lot more than hist triggers, although it also requires a lot more effort. Hist triggers is a simple enhancement to trace, for some common system-wide observability. Hist triggers is also lightweight to enable in custom ways: you just need a shell, whereas BPF typically needs one (or more) compilers. I suspect there'll also be a little give and take between them for a while; for example, doing function execution counts I'm still using ftrace, since it's currently faster to initialize than BPF.

Most people won't need to know BPF or hist triggers in much detail, as I suspect most people will be using these capabilities via front end tools (either CLI or GUI). You do need to know the kinds of things that are possible (browse this post), so you can reach for the tools – or learn the interfaces – when you need them.

Thanks Tom Zanussi (Intel) for hist triggers!

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).