Linux Trace Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

FUNCTION TRACER DESIGN

Author
Mike Frysinger

Caution: This document is out of date. Some of the description below doesn't match current implementation now.

1.1 Introduction

Here we will cover the architecture pieces that the common function tracing code relies on for proper functioning. Things are broken down into increasing complexity so that you can start simple and at least get basic functionality.

Note that this focuses on architecture implementation details only. If you want more explanation of a feature in terms of common code, review the common ftrace.txt file.

Ideally, everyone who wishes to retain performance while supporting tracing in their kernel should make it all the way to dynamic ftrace support.

1.2 Prerequisites

Ftrace relies on these features being implemented:

- STACKTRACE_SUPPORT - implement save_stack_trace()
- TRACE_IRQFLAGS_SUPPORT - implement include/asm/irqflags.h

1.3 HAVE_FUNCTION_TRACER

You will need to implement the mcount and the ftrace_stub functions.

The exact mcount symbol name will depend on your toolchain. Some call it "mcount", "_mcount", or even "__mcount". You can probably figure it out by running something like:

```
$ echo 'main(){}' | gcc -x c -S -o - - -pg | grep mcount
        call    mcount
```

We'll make the assumption below that the symbol is "mcount" just to keep things nice and simple in the examples.

Keep in mind that the ABI that is in effect inside of the mcount function is *highly* architecture/toolchain specific. We cannot help you in this regard, sorry. Dig up some old documentation and/or find someone more familiar than you to bang ideas off of. Typically, register usage (argument/scratch/etc…) is a major issue at this point, especially in relation to the location of the mcount call (before/after function prologue). You might also want to look at how glibc has implemented the mcount function for your architecture. It might be (semi-)relevant.

The mcount function should check the function pointer ftrace_trace_function to see if it is set to ftrace_stub. If it is, there is nothing for you to do, so return immediately. If it isn't, then call that function in the same way the mcount function normally calls __mcount_internal – the first argument is the "frompc" while the second argument is the "selfpc" (adjusted to remove the size of the mcount call that is embedded in the function).

For example, if the function foo() calls bar(), when the bar() function calls mcount(), the arguments mcount() will pass to the tracer are:

- "frompc" - the address bar() will use to return to foo()
- "selfpc" - the address bar() (with mcount() size adjustment)

Also keep in mind that this mcount function will be called *a lot*, so optimizing for the default case of no tracer will help the smooth running of your system when tracing is disabled. So the start of the mcount function is typically the bare minimum with checking things before returning. That also means the code flow should usually be kept linear (i.e. no branching in the nop case). This is of course an optimization and not a hard requirement.

Here is some pseudo code that should help (these functions should actually be implemented in assembly):

```
void ftrace_stub(void)
{
        return;
}

void mcount(void)
{
        /* save any bare state needed in order to do initial␣
```

(continues on next page)

(continued from previous page)

```
↪checking */

        extern void (*ftrace_trace_function)(unsigned long,␣
↪unsigned long);
        if (ftrace_trace_function != ftrace_stub)
                goto do_trace;

        /* restore any bare state */

        return;

do_trace:

        /* save all state needed by the ABI (see paragraph above) */

        unsigned long frompc = ...;
        unsigned long selfpc = <return address> - MCOUNT_INSN_SIZE;
        ftrace_trace_function(frompc, selfpc);

        /* restore all state needed by the ABI */
}
```

Don' t forget to export mcount for modules !

```
extern void mcount(void);
EXPORT_SYMBOL(mcount);
```

1.4 HAVE_FUNCTION_GRAPH_TRACER

Deep breath …time to do some real work. Here you will need to update the mcount function to check ftrace graph function pointers, as well as implement some functions to save (hijack) and restore the return address.

The mcount function should check the function pointers ftrace_graph_return (compare to ftrace_stub) and ftrace_graph_entry (compare to ftrace_graph_entry_stub). If either of those is not set to the relevant stub function, call the arch-specific function ftrace_graph_caller which in turn calls the arch-specific function prepare_ftrace_return. Neither of these function names is strictly required, but you should use them anyway to stay consistent across the architecture ports – easier to compare & contrast things.

The arguments to prepare_ftrace_return are slightly different than what are passed to ftrace_trace_function. The second argument "selfpc" is the same, but the first argument should be a pointer to the "frompc". Typically this is located on the stack. This allows the function to hijack the return address temporarily to have it point to the arch-specific function return_to_handler. That function will simply call the common ftrace_return_to_handler function and that will return the original return address with which you can return to the original call site.

Here is the updated mcount pseudo code:

```
void mcount(void)
{
...
        if (ftrace_trace_function != ftrace_stub)
                goto do_trace;

+#ifdef CONFIG_FUNCTION_GRAPH_TRACER
+       extern void (*ftrace_graph_return)(...);
+       extern void (*ftrace_graph_entry)(...);
+       if (ftrace_graph_return != ftrace_stub ||
+           ftrace_graph_entry != ftrace_graph_entry_stub)
+               ftrace_graph_caller();
+#endif

        /* restore any bare state */
...
```

Here is the pseudo code for the new ftrace_graph_caller assembly function:

```
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
void ftrace_graph_caller(void)
{
        /* save all state needed by the ABI */

        unsigned long *frompc = &...;
        unsigned long selfpc = <return address> - MCOUNT_INSN_SIZE;
        /* passing frame pointer up is optional -- see below */
        prepare_ftrace_return(frompc, selfpc, frame_pointer);

        /* restore all state needed by the ABI */
}
#endif
```

For information on how to implement prepare_ftrace_return(), simply look at the x86 version (the frame pointer passing is optional; see the next section for more information). The only architecture-specific piece in it is the setup of the fault recovery table (the asm(…) code). The rest should be the same across architectures.

Here is the pseudo code for the new return_to_handler assembly function. Note that the ABI that applies here is different from what applies to the mcount code. Since you are returning from a function (after the epilogue), you might be able to skimp on things saved/restored (usually just registers used to pass return values).

```
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
void return_to_handler(void)
{
        /* save all state needed by the ABI (see paragraph above) */

        void (*original_return_point)(void) = ftrace_return_to_
↪handler();

```

(continues on next page)

(continued from previous page)

```
        /* restore all state needed by the ABI */

        /* this is usually either a return or a jump */
        original_return_point();
}
#endif
```

1.5 HAVE_FUNCTION_GRAPH_FP_TEST

An arch may pass in a unique value (frame pointer) to both the entering and exiting of a function. On exit, the value is compared and if it does not match, then it will panic the kernel. This is largely a sanity check for bad code generation with gcc. If gcc for your port sanely updates the frame pointer under different optimization levels, then ignore this option.

However, adding support for it isn't terribly difficult. In your assembly code that calls prepare_ftrace_return(), pass the frame pointer as the 3rd argument. Then in the C version of that function, do what the x86 port does and pass it along to ftrace_push_return_trace() instead of a stub value of 0.

Similarly, when you call ftrace_return_to_handler(), pass it the frame pointer.

1.6 HAVE_FUNCTION_GRAPH_RET_ADDR_PTR

An arch may pass in a pointer to the return address on the stack. This prevents potential stack unwinding issues where the unwinder gets out of sync with ret_stack and the wrong addresses are reported by ftrace_graph_ret_addr().

Adding support for it is easy: just define the macro in asm/ftrace.h and pass the return address pointer as the 'retp' argument to ftrace_push_return_trace().

1.7 HAVE_SYSCALL_TRACEPOINTS

You need very few things to get the syscalls tracing in an arch.

- Support HAVE_ARCH_TRACEHOOK (see arch/Kconfig).
- Have a NR_syscalls variable in <asm/unistd.h> that provides the number of syscalls supported by the arch.
- Support the TIF_SYSCALL_TRACEPOINT thread flags.
- Put the trace_sys_enter() and trace_sys_exit() tracepoints calls from ptrace in the ptrace syscalls tracing path.
- If the system call table on this arch is more complicated than a simple array of addresses of the system calls, implement an arch_syscall_addr to return the address of a given system call.

- If the symbol names of the system calls do not match the function names on this arch, define ARCH_HAS_SYSCALL_MATCH_SYM_NAME in asm/ftrace.h and implement arch_syscall_match_sym_name with the appropriate logic to return true if the function name corresponds with the symbol name.
- Tag this arch as HAVE_SYSCALL_TRACEPOINTS.

1.8 HAVE_FTRACE_MCOUNT_RECORD

See scripts/recordmcount.pl for more info. Just fill in the arch-specific details for how to locate the addresses of mcount call sites via objdump. This option doesn't make much sense without also implementing dynamic ftrace.

1.9 HAVE_DYNAMIC_FTRACE

You will first need HAVE_FTRACE_MCOUNT_RECORD and HAVE_FUNCTION_TRACER, so scroll your reader back up if you got over eager.

Once those are out of the way, you will need to implement:

- **asm/ftrace.h:**
  - MCOUNT_ADDR
  - ftrace_call_adjust()
  - struct dyn_arch_ftrace{}
- **asm code:**
  - mcount() (new stub)
  - ftrace_caller()
  - ftrace_call()
  - ftrace_stub()
- **C code:**
  - ftrace_dyn_arch_init()
  - ftrace_make_nop()
  - ftrace_make_call()
  - ftrace_update_ftrace_func()

First you will need to fill out some arch details in your asm/ftrace.h.

Define MCOUNT_ADDR as the address of your mcount symbol similar to:

```
#define MCOUNT_ADDR ((unsigned long)mcount)
```

Since no one else will have a decl for that function, you will need to:

```
extern void mcount(void);
```

You will also need the helper function ftrace_call_adjust(). Most people will be able to stub it out like so:

```
static inline unsigned long ftrace_call_adjust(unsigned long addr)
{
        return addr;
}
```

<details to be filled>

Lastly you will need the custom dyn_arch_ftrace structure. If you need some extra state when runtime patching arbitrary call sites, this is the place. For now though, create an empty struct:

```
struct dyn_arch_ftrace {
        /* No extra data needed */
};
```

With the header out of the way, we can fill out the assembly code. While we did already create a mcount() function earlier, dynamic ftrace only wants a stub function. This is because the mcount() will only be used during boot and then all references to it will be patched out never to return. Instead, the guts of the old mcount() will be used to create a new ftrace_caller() function. Because the two are hard to merge, it will most likely be a lot easier to have two separate definitions split up by #ifdefs. Same goes for the ftrace_stub() as that will now be inlined in ftrace_caller().

Before we get confused anymore, let's check out some pseudo code so you can implement your own stuff in assembly:

```
void mcount(void)
{
        return;
}

void ftrace_caller(void)
{
        /* save all state needed by the ABI (see paragraph above) */

        unsigned long frompc = ...;
        unsigned long selfpc = <return address> - MCOUNT_INSN_SIZE;

ftrace_call:
        ftrace_stub(frompc, selfpc);

        /* restore all state needed by the ABI */

ftrace_stub:
        return;
}
```

This might look a little odd at first, but keep in mind that we will be runtime patching multiple things. First, only functions that we actually want to trace will be patched to call ftrace_caller(). Second, since we only have one tracer active at a time, we will patch the ftrace_caller() function itself to call the specific tracer in question. That is the point of the ftrace_call label.

With that in mind, let's move on to the C code that will actually be doing the runtime patching. You'll need a little knowledge of your arch's opcodes in order to make it through the next section.

Every arch has an init callback function. If you need to do something early on to initialize some state, this is the time to do that. Otherwise, this simple function below should be sufficient for most people:

```
int __init ftrace_dyn_arch_init(void)
{
        return 0;
}
```

There are two functions that are used to do runtime patching of arbitrary functions. The first is used to turn the mcount call site into a nop (which is what helps us retain runtime performance when not tracing). The second is used to turn the mcount call site into a call to an arbitrary location (but typically that is ftracer_caller()). See the general function definition in linux/ftrace.h for the functions:

```
ftrace_make_nop()
ftrace_make_call()
```

The rec->ip value is the address of the mcount call site that was collected by the scripts/recordmcount.pl during build time.

The last function is used to do runtime patching of the active tracer. This will be modifying the assembly code at the location of the ftrace_call symbol inside of the ftrace_caller() function. So you should have sufficient padding at that location to support the new function calls you'll be inserting. Some people will be using a "call" type instruction while others will be using a "branch" type instruction. Specifically, the function is:

```
ftrace_update_ftrace_func()
```

1.10 HAVE_DYNAMIC_FTRACE + HAVE_FUNCTION_GRAPH_TRACER

The function grapher needs a few tweaks in order to work with dynamic ftrace. Basically, you will need to:

- **update:**
  - ftrace_caller()
  - ftrace_graph_call()
  - ftrace_graph_caller()
- **implement:**

  - ftrace_enable_ftrace_graph_caller()
  - ftrace_disable_ftrace_graph_caller()

<details to be filled>

Quick notes:

- add a nop stub after the ftrace_call location named ftrace_graph_call; stub needs to be large enough to support a call to ftrace_graph_caller()
- update ftrace_graph_caller() to work with being called by the new ftrace_caller() since some semantics may have changed
- ftrace_enable_ftrace_graph_caller() will runtime patch the ftrace_graph_call location with a call to ftrace_graph_caller()
- ftrace_disable_ftrace_graph_caller() will runtime patch the ftrace_graph_call location with nops

CHAPTER

TWO

NOTES ON ANALYSING BEHAVIOUR USING EVENTS AND TRACEPOINTS

Author
Mel Gorman (PCL information heavily based on email from Ingo Molnar)

2.1 1. Introduction

Tracepoints (see *Using the Linux Kernel Tracepoints*) can be used without creating custom kernel modules to register probe functions using the event tracing infrastructure.

Simplistically, tracepoints represent important events that can be taken in conjunction with other tracepoints to build a "Big Picture" of what is going on within the system. There are a large number of methods for gathering and interpreting these events. Lacking any current Best Practises, this document describes some of the methods that can be used.

This document assumes that debugfs is mounted on /sys/kernel/debug and that the appropriate tracing options have been configured into the kernel. It is assumed that the PCL tool tools/perf has been installed and is in your path.

2.2 2. Listing Available Events

2.2.1 2.1 Standard Utilities

All possible events are visible from /sys/kernel/debug/tracing/events. Simply calling:

```
$ find /sys/kernel/debug/tracing/events -type d
```

will give a fair indication of the number of events available.

2.2.2 2.2 PCL (Performance Counters for Linux)

Discovery and enumeration of all counters and events, including tracepoints, are available with the perf tool. Getting a list of available events is a simple case of:

```
$ perf list 2>&1 | grep Tracepoint
ext4:ext4_free_inode                     [Tracepoint event]
ext4:ext4_request_inode                  [Tracepoint event]
ext4:ext4_allocate_inode                 [Tracepoint event]
ext4:ext4_write_begin                    [Tracepoint event]
ext4:ext4_ordered_write_end              [Tracepoint event]
[ .... remaining output snipped .... ]
```

2.3 3. Enabling Events

2.3.1 3.1 System-Wide Event Enabling

See *Event Tracing* for a proper description on how events can be enabled system-wide. A short example of enabling all events related to page allocation would look something like:

```
$ for i in `find /sys/kernel/debug/tracing/events -name "enable" | grep mm_`; do echo 1 > $i; done
```

2.3.2 3.2 System-Wide Event Enabling with SystemTap

In SystemTap, tracepoints are accessible using the kernel.trace() function call. The following is an example that reports every 5 seconds what processes were allocating the pages.

```
global page_allocs

probe kernel.trace("mm_page_alloc") {
      page_allocs[execname()]++
}

function print_count() {
      printf ("%-25s %-s\n", "#Pages Allocated", "Process Name")
      foreach (proc in page_allocs-)
              printf("%-25d %s\n", page_allocs[proc], proc)
      printf ("\n")
      delete page_allocs
}

probe timer.s(5) {
        print_count()
}
```

2.3.3 3.3 System-Wide Event Enabling with PCL

By specifying the -a switch and analysing sleep, the system-wide events for a duration of time can be examined.

```
$ perf stat -a \
       -e kmem:mm_page_alloc -e kmem:mm_page_free \
       -e kmem:mm_page_free_batched \
       sleep 10
Performance counter stats for 'sleep 10':

          9630  kmem:mm_page_alloc
          2143  kmem:mm_page_free
          7424  kmem:mm_page_free_batched

  10.002577764  seconds time elapsed
```

Similarly, one could execute a shell and exit it as desired to get a report at that point.

2.3.4 3.4 Local Event Enabling

ftrace - Function Tracer describes how to enable events on a per-thread basis using set_ftrace_pid.

2.3.5 3.5 Local Event Enablement with PCL

Events can be activated and tracked for the duration of a process on a local basis using PCL such as follows.

```
$ perf stat -e kmem:mm_page_alloc -e kmem:mm_page_free \
               -e kmem:mm_page_free_batched ./hackbench 10
Time: 0.909

  Performance counter stats for './hackbench 10':

        17803  kmem:mm_page_alloc
        12398  kmem:mm_page_free
         4827  kmem:mm_page_free_batched

  0.973913387  seconds time elapsed
```

2.4 4. Event Filtering

ftrace - Function Tracer covers in-depth how to filter events in ftrace. Obviously using grep and awk of trace_pipe is an option as well as any script reading trace_pipe.

2.5 5. Analysing Event Variances with PCL

Any workload can exhibit variances between runs and it can be important to know what the standard deviation is. By and large, this is left to the performance analyst to do it by hand. In the event that the discrete event occurrences are useful to the performance analyst, then perf can be used.

```
$ perf stat --repeat 5 -e kmem:mm_page_alloc -e kmem:mm_page_free
                      -e kmem:mm_page_free_batched ./hackbench 10
Time: 0.890
Time: 0.895
Time: 0.915
Time: 1.001
Time: 0.899

 Performance counter stats for './hackbench 10' (5 runs):

       16630  kmem:mm_page_alloc         ( +-   3.542% )
       11486  kmem:mm_page_free          ( +-   4.771% )
        4730  kmem:mm_page_free_batched  ( +-   2.325% )

  0.982653002  seconds time elapsed   ( +-   1.448% )
```

In the event that some higher-level event is required that depends on some aggregation of discrete events, then a script would need to be developed.

Using –repeat, it is also possible to view how events are fluctuating over time on a system-wide basis using -a and sleep.

```
$ perf stat -e kmem:mm_page_alloc -e kmem:mm_page_free \
              -e kmem:mm_page_free_batched \
              -a --repeat 10 \
              sleep 1
Performance counter stats for 'sleep 1' (10 runs):

         1066  kmem:mm_page_alloc         ( +-  26.148% )
          182  kmem:mm_page_free          ( +-   5.464% )
          890  kmem:mm_page_free_batched  ( +-  30.079% )

  1.002251757  seconds time elapsed   ( +-   0.005% )
```

2.6 6. Higher-Level Analysis with Helper Scripts

When events are enabled the events that are triggering can be read from /sys/kernel/debug/tracing/trace_pipe in human-readable format although binary options exist as well. By post-processing the output, further information can be gathered on-line as appropriate. Examples of post-processing might include

- Reading information from /proc for the PID that triggered the event
- Deriving a higher-level event from a series of lower-level events.
- Calculating latencies between two events

Documentation/trace/postprocess/trace-pagealloc-postprocess.pl is an example script that can read trace_pipe from STDIN or a copy of a trace. When used on-line, it can be interrupted once to generate a report without exiting and twice to exit.

Simplistically, the script just reads STDIN and counts up events but it also can do more such as

- Derive high-level events from many low-level events. If a number of pages are freed to the main allocator from the per-CPU lists, it recognises that as one per-CPU drain even though there is no specific tracepoint for that event
- It can aggregate based on PID or individual process number
- In the event memory is getting externally fragmented, it reports on whether the fragmentation event was severe or moderate.
- When receiving an event about a PID, it can record who the parent was so that if large numbers of events are coming from very short-lived processes, the parent process responsible for creating all the helpers can be identified

2.7 7. Lower-Level Analysis with PCL

There may also be a requirement to identify what functions within a program were generating events within the kernel. To begin this sort of analysis, the data must be recorded. At the time of writing, this required root:

```
$ perf record -c 1 \
      -e kmem:mm_page_alloc -e kmem:mm_page_free \
      -e kmem:mm_page_free_batched \
      ./hackbench 10
Time: 0.894
[ perf record: Captured and wrote 0.733 MB perf.data (~32010␣
↪samples) ]
```

Note the use of '-c 1' to set the event period to sample. The default sample period is quite high to minimise overhead but the information collected can be very coarse as a result.

This record outputted a file called perf.data which can be analysed using perf report.

```
$ perf report
# Samples: 30922
#
# Overhead    Command                     Shared Object
# ........  .........  ................................
#
    87.27%  hackbench  [vdso]
     6.85%  hackbench  /lib/i686/cmov/libc-2.9.so
     2.62%  hackbench  /lib/ld-2.9.so
     1.52%       perf  [vdso]
     1.22%  hackbench  ./hackbench
     0.48%  hackbench  [kernel]
     0.02%       perf  /lib/i686/cmov/libc-2.9.so
     0.01%       perf  /usr/bin/perf
     0.01%       perf  /lib/ld-2.9.so
     0.00%  hackbench  /lib/i686/cmov/libpthread-2.9.so
#
# (For more details, try: perf report --sort comm,dso,symbol)
#
```

According to this, the vast majority of events triggered on events within the VDSO. With simple binaries, this will often be the case so let's take a slightly different example. In the course of writing this, it was noticed that X was generating an insane amount of page allocations so let's look at it:

```
$ perf record -c 1 -f \
              -e kmem:mm_page_alloc -e kmem:mm_page_free \
              -e kmem:mm_page_free_batched \
              -p `pidof X`
```

This was interrupted after a few seconds and

```
$ perf report
# Samples: 27666
#
# Overhead  Command                            Shared Object
# ........  .......  .......................................
#
    51.95%     Xorg  [vdso]
    47.95%     Xorg  /opt/gfx-test/lib/libpixman-1.so.0.13.1
     0.09%     Xorg  /lib/i686/cmov/libc-2.9.so
     0.01%     Xorg  [kernel]
#
# (For more details, try: perf report --sort comm,dso,symbol)
#
```

So, almost half of the events are occurring in a library. To get an idea which symbol:

```
$ perf report --sort comm,dso,symbol
# Samples: 27666
#
```

(continues on next page)

(continued from previous page)

```
# Overhead  Command                                 Shared Object  Symbol
# ........  .......  ....................................  ......
#
    51.95%     Xorg  [vdso]                                [.]␣
→0x000000ffffe424
    47.93%     Xorg  /opt/gfx-test/lib/libpixman-1.so.0.13.1  [.]␣
→pixmanFillsse2
     0.09%     Xorg  /lib/i686/cmov/libc-2.9.so            [.] _
→int_malloc
     0.01%     Xorg  /opt/gfx-test/lib/libpixman-1.so.0.13.1  [.]␣
→pixman_region32_copy_f
     0.01%     Xorg  [kernel]                              [k]␣
→read_hpet
     0.01%     Xorg  /opt/gfx-test/lib/libpixman-1.so.0.13.1  [.]␣
→get_fast_path
     0.00%     Xorg  [kernel]                              [k]␣
→ftrace_trace_userstack
```

To see where within the function pixmanFillsse2 things are going wrong:

```
$ perf annotate pixmanFillsse2
[ ... ]
  0.00 :         34eeb:       0f 18 08                prefetcht0 (
→%eax)
       :      }
       :
       :      extern __inline void __attribute__((__gnu_inline__, __
→always_inline__, _
       :      _mm_store_si128 (__m128i *__P, __m128i __B) :      {
       :        *__P = __B;
 12.40 :         34eee:       66 0f 7f 80 40 ff ff    movdqa %xmm0,-
→0xc0(%eax)
  0.00 :         34ef5:       ff
 12.40 :         34ef6:       66 0f 7f 80 50 ff ff    movdqa %xmm0,-
→0xb0(%eax)
  0.00 :         34efd:       ff
 12.39 :         34efe:       66 0f 7f 80 60 ff ff    movdqa %xmm0,-
→0xa0(%eax)
  0.00 :         34f05:       ff
 12.67 :         34f06:       66 0f 7f 80 70 ff ff    movdqa %xmm0,-
→0x90(%eax)
  0.00 :         34f0d:       ff
 12.58 :         34f0e:       66 0f 7f 40 80          movdqa %xmm0,-
→0x80(%eax)
 12.31 :         34f13:       66 0f 7f 40 90          movdqa %xmm0,-
→0x70(%eax)
 12.40 :         34f18:       66 0f 7f 40 a0          movdqa %xmm0,-
→0x60(%eax)
 12.31 :         34f1d:       66 0f 7f 40 b0          movdqa %xmm0,-
→0x50(%eax)
```

At a glance, it looks like the time is being spent copying pixmaps to the card. Further investigation would be needed to determine why pixmaps are being copied around so much but a starting point would be to take an ancient build of libpixmap out of the library path where it was totally forgotten about from months ago!

CHAPTER

THREE

FTRACE - FUNCTION TRACER

Author
Steven Rostedt <srostedt@redhat.com>

Original Reviewers
Elias Oltmanns, Randy Dunlap, Andrew Morton, John Kacur, and David Teigland.

- Written for: 2.6.28-rc2
- Updated for: 3.10
- Updated for: 4.13 - Copyright 2017 VMware Inc. Steven Rostedt
- Converted to rst format - Changbin Du <changbin.du@intel.com>

3.1 Introduction

Ftrace is an internal tracer designed to help out developers and designers of systems to find what is going on inside the kernel. It can be used for debugging or analyzing latencies and performance issues that take place outside of user-space.

Although ftrace is typically considered the function tracer, it is really a framework of several assorted tracing utilities. There' s latency tracing to examine what occurs between interrupts disabled and enabled, as well as for preemption and from a time a task is woken to the task is actually scheduled in.

One of the most common uses of ftrace is the event tracing. Throughout the kernel is hundreds of static event points that can be enabled via the tracefs file system to see what is going on in certain parts of the kernel.

See events.rst for more information.

3.2 Implementation Details

See *Function Tracer Design* for details for arch porters and such.

3.3 The File System

Ftrace uses the tracefs file system to hold the control files as well as the files to display output.

When tracefs is configured into the kernel (which selecting any ftrace option will do) the directory /sys/kernel/tracing will be created. To mount this directory, you can add to your /etc/fstab file:

```
tracefs       /sys/kernel/tracing       tracefs defaults        0     0
```

Or you can mount it at run time with:

```
mount -t tracefs nodev /sys/kernel/tracing
```

For quicker access to that directory you may want to make a soft link to it:

```
ln -s /sys/kernel/tracing /tracing
```

> **Attention:** Before 4.1, all ftrace tracing control files were within the debugfs file system, which is typically located at /sys/kernel/debug/tracing. For backward compatibility, when mounting the debugfs file system, the tracefs file system will be automatically mounted at:
>
> /sys/kernel/debug/tracing
>
> All files located in the tracefs file system will be located in that debugfs file system directory as well.

> **Attention:** Any selected ftrace option will also create the tracefs file system. The rest of the document will assume that you are in the ftrace directory (cd /sys/kernel/tracing) and will only concentrate on the files within that directory and not distract from the content with the extended "/sys/kernel/tracing" path name.

That's it! (assuming that you have ftrace configured into your kernel)

After mounting tracefs you will have access to the control and output files of ftrace. Here is a list of some of the key files:

Note: all time values are in microseconds.

current_tracer:

This is used to set or display the current tracer that is configured. Changing the current tracer clears the ring buffer content as well as the "snapshot" buffer.

available_tracers:

This holds the different types of tracers that have been compiled into the kernel. The tracers listed here can be configured by echoing their name into current_tracer.

tracing_on:

This sets or displays whether writing to the trace ring buffer is enabled. Echo 0 into this file to disable the tracer or 1 to enable it. Note, this only disables writing to the ring buffer, the tracing overhead may still be occurring.

The kernel function tracing_off() can be used within the kernel to disable writing to the ring buffer, which will set this file to "0". User space can re-enable tracing by echoing "1" into the file.

Note, the function and event trigger "traceoff" will also set this file to zero and stop tracing. Which can also be re-enabled by user space using this file.

trace:

This file holds the output of the trace in a human readable format (described below). Opening this file for writing with the O_TRUNC flag clears the ring buffer content. Note, this file is not a consumer. If tracing is off (no tracer running, or tracing_on is zero), it will produce the same output each time it is read. When tracing is on, it may produce inconsistent results as it tries to read the entire buffer without consuming it.

trace_pipe:

The output is the same as the "trace" file but this file is meant to be streamed with live tracing. Reads from this file will block until new data is retrieved. Unlike the "trace" file, this file is a consumer. This means reading from this file causes sequential reads to display more current data. Once data is read from this file, it is consumed, and will not be read again with a sequential read. The "trace" file is static, and if the tracer is not adding more data, it will display the same information every time it is read.

trace_options:

This file lets the user control the amount of data that is displayed in one of the above output files. Options also exist to modify how a tracer or events work (stack traces, timestamps, etc).

options:

> This is a directory that has a file for every available trace option (also in trace_options). Options may also be set or cleared by writing a "1" or "0" respectively into the corresponding file with the option name.

tracing_max_latency:

> Some of the tracers record the max latency. For example, the maximum time that interrupts are disabled. The maximum time is saved in this file. The max trace will also be stored, and displayed by "trace". A new max trace will only be recorded if the latency is greater than the value in this file (in microseconds).
>
> By echoing in a time into this file, no latency will be recorded unless it is greater than the time in this file.

tracing_thresh:

> Some latency tracers will record a trace whenever the latency is greater than the number in this file. Only active when the file contains a number greater than 0. (in microseconds)

buffer_size_kb:

> This sets or displays the number of kilobytes each CPU buffer holds. By default, the trace buffers are the same size for each CPU. The displayed number is the size of the CPU buffer and not total size of all buffers. The trace buffers are allocated in pages (blocks of memory that the kernel uses for allocation, usually 4 KB in size). A few extra pages may be allocated to accommodate buffer management meta-data. If the last page allocated has room for more bytes than requested, the rest of the page will be used, making the actual allocation bigger than requested or shown. (Note, the size may not be a multiple of the page size due to buffer management meta-data.)
>
> Buffer sizes for individual CPUs may vary (see "per_cpu/cpu0/buffer_size_kb" below), and if they do this file will show "X".

buffer_total_size_kb:

> This displays the total combined size of all the trace buffers.

free_buffer:

> If a process is performing tracing, and the ring buffer should be shrunk "freed" when the process is finished, even if it were to be killed by a signal, this file can be used for that purpose. On close of this file, the ring buffer will be resized to its minimum size. Having a process that is tracing also open this file, when the process

exits its file descriptor for this file will be closed, and in doing so, the ring buffer will be "freed".

It may also stop tracing if disable_on_free option is set.

tracing_cpumask:

This is a mask that lets the user only trace on specified CPUs. The format is a hex string representing the CPUs.

set_ftrace_filter:

When dynamic ftrace is configured in (see the section below "dynamic ftrace"), the code is dynamically modified (code text rewrite) to disable calling of the function profiler (mcount). This lets tracing be configured in with practically no overhead in performance. This also has a side effect of enabling or disabling specific functions to be traced. Echoing names of functions into this file will limit the trace to only those functions. This influences the tracers "function" and "function_graph" and thus also function profiling (see "function_profile_enabled").

The functions listed in "available_filter_functions" are what can be written into this file.

This interface also allows for commands to be used. See the "Filter commands" section for more details.

As a speed up, since processing strings can be quite expensive and requires a check of all functions registered to tracing, instead an index can be written into this file. A number (starting with "1") written will instead select the same corresponding at the line position of the "available_filter_functions" file.

set_ftrace_notrace:

This has an effect opposite to that of set_ftrace_filter. Any function that is added here will not be traced. If a function exists in both set_ftrace_filter and set_ftrace_notrace, the function will _not_ be traced.

set_ftrace_pid:

Have the function tracer only trace the threads whose PID are listed in this file.

If the "function-fork" option is set, then when a task whose PID is listed in this file forks, the child's PID will automatically be added to this file, and the child will be traced by the function tracer as well. This option will also cause PIDs of tasks that exit to be removed from the file.

set_ftrace_notrace_pid:

Have the function tracer ignore threads whose PID are listed in this file.

If the "function-fork" option is set, then when a task whose PID is listed in this file forks, the child's PID will automatically be added to this file, and the child will not be traced by the function tracer as well. This option will also cause PIDs of tasks that exit to be removed from the file.

If a PID is in both this file and "set_ftrace_pid", then this file takes precedence, and the thread will not be traced.

set_event_pid:

Have the events only trace a task with a PID listed in this file. Note, sched_switch and sched_wake_up will also trace events listed in this file.

To have the PIDs of children of tasks with their PID in this file added on fork, enable the "event-fork" option. That option will also cause the PIDs of tasks to be removed from this file when the task exits.

set_event_notrace_pid:

Have the events not trace a task with a PID listed in this file. Note, sched_switch and sched_wakeup will trace threads not listed in this file, even if a thread's PID is in the file if the sched_switch or sched_wakeup events also trace a thread that should be traced.

To have the PIDs of children of tasks with their PID in this file added on fork, enable the "event-fork" option. That option will also cause the PIDs of tasks to be removed from this file when the task exits.

set_graph_function:

Functions listed in this file will cause the function graph tracer to only trace these functions and the functions that they call. (See the section "dynamic ftrace"for more details). Note, set_ftrace_filter and set_ftrace_notrace still affects what functions are being traced.

set_graph_notrace:

Similar to set_graph_function, but will disable function graph tracing when the function is hit until it exits the function. This makes it possible to ignore tracing functions that are called by a specific function.

available_filter_functions:

This lists the functions that ftrace has processed and can trace. These are the function names that you can pass to "set_ftrace_filter", "set_ftrace_notrace", "set_graph_function", or "set_graph_notrace". (See the section "dynamic ftrace" below for more details.)

dyn_ftrace_total_info:

This file is for debugging purposes. The number of functions that have been converted to nops and are available to be traced.

enabled_functions:

This file is more for debugging ftrace, but can also be useful in seeing if any function has a callback attached to it. Not only does the trace infrastructure use ftrace function trace utility, but other subsystems might too. This file displays all functions that have a callback attached to them as well as the number of callbacks that have been attached. Note, a callback may also call multiple functions which will not be listed in this count.

If the callback registered to be traced by a function with the "save regs" attribute (thus even more overhead), a 'R' will be displayed on the same line as the function that is returning registers.

If the callback registered to be traced by a function with the "ip modify" attribute (thus the regs->ip can be changed), an 'I' will be displayed on the same line as the function that can be overridden.

If the architecture supports it, it will also show what callback is being directly called by the function. If the count is greater than 1 it most likely will be ftrace_ops_list_func().

If the callback of the function jumps to a trampoline that is specific to a the callback and not the standard trampoline, its address will be printed as well as the function that the trampoline calls.

function_profile_enabled:

When set it will enable all functions with either the function tracer, or if configured, the function graph tracer. It will keep a histogram of the number of functions that were called and if the function graph tracer was configured, it will also keep track of the time spent in those functions. The histogram content can be displayed in the files:

trace_stat/function<cpu> (function0, function1, etc).

trace_stat:

A directory that holds different tracing stats.

kprobe_events:

Enable dynamic trace points. See kprobetrace.rst.

kprobe_profile:

Dynamic trace points stats. See kprobetrace.rst.

max_graph_depth:

Used with the function graph tracer. This is the max depth it will trace into a function. Setting this to a value of one will show only the first kernel function that is called from user space.

printk_formats:

This is for tools that read the raw format files. If an event in the ring buffer references a string, only a pointer to the string is recorded into the buffer and not the string itself. This prevents tools from knowing what that string was. This file displays the string and address for the string allowing tools to map the pointers to what the strings were.

saved_cmdlines:

Only the pid of the task is recorded in a trace event unless the event specifically saves the task comm as well. Ftrace makes a cache of pid mappings to comms to try to display comms for events. If a pid for a comm is not listed, then "<…>" is displayed in the output.

If the option "record-cmd" is set to "0", then comms of tasks will not be saved during recording. By default, it is enabled.

saved_cmdlines_size:

By default, 128 comms are saved (see "saved_cmdlines" above). To increase or decrease the amount of comms that are cached, echo the number of comms to cache into this file.

saved_tgids:

If the option "record-tgid" is set, on each scheduling context switch the Task Group ID of a task is saved in a table mapping the PID of the thread to its TGID. By default, the "record-tgid" option is disabled.

snapshot:

This displays the "snapshot" buffer and also lets the user take a snapshot of the current running trace. See the "Snapshot" section below for more details.

stack_max_size:

When the stack tracer is activated, this will display the maximum stack size it has encountered. See the "Stack Trace" section below.

stack_trace:

This displays the stack back trace of the largest stack that was encountered when the stack tracer is activated.

See the "Stack Trace" section below.

stack_trace_filter:

This is similar to "set_ftrace_filter" but it limits what functions the stack tracer will check.

trace_clock:

Whenever an event is recorded into the ring buffer, a "timestamp" is added. This stamp comes from a specified clock. By default, ftrace uses the "local" clock. This clock is very fast and strictly per cpu, but on some systems it may not be monotonic with respect to other CPUs. In other words, the local clocks may not be in sync with local clocks on other CPUs.

Usual clocks for tracing:

```
# cat trace_clock
[local] global counter x86-tsc
```

The clock with the square brackets around it is the one in effect.

local:
Default clock, but may not be in sync across CPUs

global:
This clock is in sync with all CPUs but may be a bit slower than the local clock.

counter:
This is not a clock at all, but literally an atomic counter. It counts up one by one, but is in sync with all CPUs. This is useful when you need to know exactly the order events occurred with respect to each other on different CPUs.

uptime:
This uses the jiffies counter and the time stamp is relative to the time since boot up.

perf:
This makes ftrace use the same clock that perf uses. Eventually perf will be able to read ftrace buffers and this will help out in interleaving the data.

x86-tsc:
Architectures may define their own clocks. For example, x86 uses its own TSC cycle clock here.

ppc-tb:
This uses the powerpc timebase register value. This is in sync across CPUs and can also be used to correlate events across hypervisor/guest if tb_offset is known.

mono:
This uses the fast monotonic clock (CLOCK_MONOTONIC) which is monotonic and is subject to NTP rate adjustments.

mono_raw:
This is the raw monotonic clock (CLOCK_MONOTONIC_RAW) which is monotonic but is not subject to any rate adjustments and ticks at the same rate as the hardware clocksource.

boot:
This is the boot clock (CLOCK_BOOTTIME) and is based on the fast monotonic clock, but also accounts for time spent in suspend. Since the clock access is designed for use in tracing in the suspend path, some side effects are possible if clock is accessed after the suspend time is accounted before the fast mono clock is updated. In this case, the clock update appears to happen slightly sooner than it normally would have. Also on 32-bit systems, it's possible that the 64-bit boot offset sees a partial update. These effects are rare and post processing should be able to handle them. See comments in the ktime_get_boot_fast_ns() function for more information.

To set a clock, simply echo the clock name into this file:

```
# echo global > trace_clock
```

Setting a clock clears the ring buffer content as well as the "snapshot" buffer.

trace_marker:

This is a very useful file for synchronizing user space with events happening in the kernel. Writing strings into this file will be written into the ftrace buffer.

It is useful in applications to open this file at the start of the application and just reference the file descriptor for the file:

```
void trace_write(const char *fmt, ...)
{
        va_list ap;
        char buf[256];
        int n;

        if (trace_fd < 0)
                return;

        va_start(ap, fmt);
        n = vsnprintf(buf, 256, fmt, ap);
```

(continues on next page)

(continued from previous page)

```
        va_end(ap);

        write(trace_fd, buf, n);
}
```

start:

```
trace_fd = open("trace_marker", WR_ONLY);
```

Note: Writing into the trace_marker file can also initiate triggers

that are written into /sys/kernel/tracing/events/ftrace/print/trigger See "Event triggers" in *Event Tracing* and an example in *Event Histograms* (Section 3.)

trace_marker_raw:

This is similar to trace_marker above, but is meant for binary data to be written to it, where a tool can be used to parse the data from trace_pipe_raw.

uprobe_events:

Add dynamic tracepoints in programs. See uprobetracer.rst

uprobe_profile:

Uprobe statistics. See uprobetrace.txt

instances:

This is a way to make multiple trace buffers where different events can be recorded in different buffers. See "Instances" section below.

events:

This is the trace event directory. It holds event tracepoints (also known as static tracepoints) that have been compiled into the kernel. It shows what event tracepoints exist and how they are grouped by system. There are "enable" files at various levels that can enable the tracepoints when a "1" is written to them.

See events.rst for more information.

set_event:

By echoing in the event into this file, will enable that event.

See events.rst for more information.

available_events:

A list of events that can be enabled in tracing.

See events.rst for more information.

timestamp_mode:

Certain tracers may change the timestamp mode used when logging trace events into the event buffer. Events with different modes can coexist within a buffer but the mode in effect when an event is logged determines which timestamp mode is used for that event. The default timestamp mode is 'delta'.

Usual timestamp modes for tracing:

cat timestamp_mode [delta] absolute

The timestamp mode with the square brackets around it is the one in effect.

delta: Default timestamp mode - timestamp is a delta against
a per-buffer timestamp.

absolute: The timestamp is a full timestamp, not a delta
against some other value. As such it takes up more space and is less efficient.

hwlat_detector:

Directory for the Hardware Latency Detector. See "Hardware Latency Detector" section below.

per_cpu:

This is a directory that contains the trace per_cpu information.

per_cpu/cpu0/buffer_size_kb:

The ftrace buffer is defined per_cpu. That is, there's a separate buffer for each CPU to allow writes to be done atomically, and free from cache bouncing. These buffers may have different size buffers. This file is similar to the buffer_size_kb file, but it only displays or sets the buffer size for the specific CPU. (here cpu0).

per_cpu/cpu0/trace:

This is similar to the "trace" file, but it will only display the data specific for the CPU. If written to, it only clears the specific CPU buffer.

per_cpu/cpu0/trace_pipe

This is similar to the "trace_pipe" file, and is a consuming read, but it will only display (and consume) the data specific for the CPU.

per_cpu/cpu0/trace_pipe_raw

For tools that can parse the ftrace ring buffer binary format, the trace_pipe_raw file can be used to extract the data from the ring buffer directly. With the use of the splice() system call, the buffer data can be quickly transferred to a file or to the network where a server is collecting the data.

Like trace_pipe, this is a consuming reader, where multiple reads will always produce different data.

per_cpu/cpu0/snapshot:

This is similar to the main "snapshot" file, but will only snapshot the current CPU (if supported). It only displays the content of the snapshot for a given CPU, and if written to, only clears this CPU buffer.

per_cpu/cpu0/snapshot_raw:

Similar to the trace_pipe_raw, but will read the binary format from the snapshot buffer for the given CPU.

per_cpu/cpu0/stats:

This displays certain stats about the ring buffer:

entries:
The number of events that are still in the buffer.

overrun:
The number of lost events due to overwriting when the buffer was full.

commit overrun:
Should always be zero. This gets set if so many events happened within a nested event (ring buffer is re-entrant), that it fills the buffer and starts dropping events.

bytes:
Bytes actually read (not overwritten).

oldest event ts:
The oldest timestamp in the buffer

now ts:
The current timestamp

dropped events:
Events lost due to overwrite option being off.

read events:
The number of events read.

3.4 The Tracers

Here is the list of current tracers that may be configured.

"function"

Function call tracer to trace all kernel functions.

"function_graph"

Similar to the function tracer except that the function tracer probes the functions on their entry whereas the function graph tracer traces on both entry and exit of the functions. It then provides the ability to draw a graph of function calls similar to C code source.

"blk"

The block tracer. The tracer used by the blktrace user application.

"hwlat"

The Hardware Latency tracer is used to detect if the hardware produces any latency. See "Hardware Latency Detector" section below.

"irqsoff"

Traces the areas that disable interrupts and saves the trace with the longest max latency. See tracing_max_latency. When a new max is recorded, it replaces the old trace. It is best to view this trace with the latency-format option enabled, which happens automatically when the tracer is selected.

"preemptoff"

Similar to irqsoff but traces and records the amount of time for which preemption is disabled.

"preemptirqsoff"

Similar to irqsoff and preemptoff, but traces and records the largest time for which irqs and/or preemption is disabled.

"wakeup"

Traces and records the max latency that it takes for the highest priority task to get scheduled after it has been woken up. Traces all tasks as an average developer would expect.

"wakeup_rt"

Traces and records the max latency that it takes for just RT tasks (as the current "wakeup" does). This is useful for those interested in wake up timings of RT tasks.

"wakeup_dl"

Traces and records the max latency that it takes for a SCHED_DEADLINE task to be woken (as the "wakeup" and "wakeup_rt" does).

"mmiotrace"

A special tracer that is used to trace binary module. It will trace all the calls that a module makes to the hardware. Everything it writes and reads from the I/O as well.

"branch"

This tracer can be configured when tracing likely/unlikely calls within the kernel. It will trace when a likely and unlikely branch is hit and if it was correct in its prediction of being correct.

"nop"

This is the "trace nothing" tracer. To remove all tracers from tracing simply echo "nop" into current_tracer.

3.5 Error conditions

For most ftrace commands, failure modes are obvious and communicated using standard return codes.

For other more involved commands, extended error information may be available via the tracing/error_log file. For the commands that support it, reading the tracing/error_log file after an error will display more detailed information about what went wrong, if information is available. The tracing/error_log file is a circular error log displaying a small number (currently, 8) of ftrace errors for the last (8) failed commands.

The extended error information and usage takes the form shown in this example:

```
# echo xxx > /sys/kernel/debug/tracing/events/sched/sched_
↪wakeup/trigger
echo: write error: Invalid argument

# cat /sys/kernel/debug/tracing/error_log
[ 5348.887237] location: error: Couldn't yyy: zzz
  Command: xxx
           ^
[ 7517.023364] location: error: Bad rrr: sss
  Command: ppp qqq
               ^
```

To clear the error log, echo the empty string into it:

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

3.6 Examples of using the tracer

Here are typical examples of using the tracers when controlling them only with the tracefs interface (without using any user-land utilities).

3.7 Output format:

Here is an example of the output format of the file "trace" :

```
# tracer: function
#
# entries-in-buffer/entries-written: 140080/250280   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
            bash-1977  [000] .... 17284.993652: sys_close <-system_
↪call_fastpath
            bash-1977  [000] .... 17284.993653: __close_fd <-sys_
↪close
            bash-1977  [000] .... 17284.993653: _raw_spin_lock <-__
↪close_fd
            sshd-1974  [003] .... 17284.993653: __srcu_read_unlock
↪<-fsnotify
            bash-1977  [000] .... 17284.993654: add_preempt_count <-
↪_raw_spin_lock
            bash-1977  [000] ...1 17284.993655: _raw_spin_unlock <-_
↪_close_fd
            bash-1977  [000] ...1 17284.993656: sub_preempt_count <-
↪_raw_spin_unlock
            bash-1977  [000] .... 17284.993657: filp_close <-__
↪close_fd
            bash-1977  [000] .... 17284.993657: dnotify_flush <-
↪filp_close
            sshd-1974  [003] .... 17284.993658: sys_select <-system_
↪call_fastpath
            ....
```

A header is printed with the tracer name that is represented by the trace. In this case the tracer is "function" . Then it shows the number of events in the buffer as well as the total number of entries that were written. The difference is the number of entries that were lost due to the buffer filling up (250280 - 140080 = 110200 events lost).

The header explains the content of the events. Task name "bash" , the task PID "1977" , the CPU that it was running on "000" , the latency format (explained below), the timestamp in <secs>.<usecs> format, the function name that was traced "sys_close" and the parent function that called this function "system_call_fastpath" . The timestamp is the time at which the function was entered.

3.8 Latency trace format

When the latency-format option is enabled or when one of the latency tracers is set, the trace file gives somewhat more information to see why a latency happened. Here is a typical trace:

```
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.8.0-test+
# --------------------------------------------------------------------
↪--
# latency: 259 us, #4/4, CPU#2 | (M:preempt VP:0, KP:0, SP:0 HP:0
↪#P:4)
#    -----------------
#    | task: ps-6143 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: __lock_task_sighand
#  => ended at:   _raw_spin_unlock_irqrestore
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
      ps-6143    2d...    0us!: trace_hardirqs_off <-__lock_task_
↪sighand
      ps-6143    2d..1  259us+: trace_hardirqs_on <-_raw_spin_
↪unlock_irqrestore
      ps-6143    2d..1  263us+: time_hardirqs_on <-_raw_spin_unlock_
↪irqrestore
      ps-6143    2d..1  306us : <stack trace>
 => trace_hardirqs_on_caller
 => trace_hardirqs_on
 => _raw_spin_unlock_irqrestore
 => do_task_stat
 => proc_tgid_stat
 => proc_single_show
 => seq_read
 => vfs_read
 => sys_read
 => system_call_fastpath
```

This shows that the current tracer is "irqsoff" tracing the time for which interrupts were disabled. It gives the trace version (which never changes) and the version of the kernel upon which this was executed on (3.8). Then it displays the max latency in microseconds (259 us). The number of trace entries displayed and the total number (both are four: #4/4). VP, KP, SP, and HP are always zero and are reserved for later use. #P is the number of online CPUs (#P:4).

The task is the process that was running when the latency occurred. (ps pid: 6143).

The start and stop (the functions in which the interrupts were disabled and enabled respectively) that caused the latencies:

- __lock_task_sighand is where the interrupts were disabled.
- _raw_spin_unlock_irqrestore is where they were enabled again.

The next lines after the header are the trace itself. The header explains which is which.

cmd: The name of the process in the trace.

pid: The PID of that process.

CPU#: The CPU which the process was running on.

irqs-off: 'd' interrupts are disabled. '.' otherwise.

> **Caution:** If the architecture does not support a way to read the irq flags variable, an 'X' will always be printed here.

need-resched:

- 'N'both TIF_NEED_RESCHED and PREEMPT_NEED_RESCHED is set,
- 'n' only TIF_NEED_RESCHED is set,
- 'p' only PREEMPT_NEED_RESCHED is set,
- '.' otherwise.

hardirq/softirq:

- 'Z' - NMI occurred inside a hardirq
- 'z' - NMI is running
- 'H' - hard irq occurred inside a softirq.
- 'h' - hard irq is running
- 's' - soft irq is running
- '.' - normal context.

preempt-depth: The level of preempt_disabled

The above is mostly meaningful for kernel developers.

time:
When the latency-format option is enabled, the trace file output includes a timestamp relative to the start of the trace. This differs from the output when latency-format is disabled, which includes an absolute timestamp.

delay:
This is just to help catch your eye a bit better. And needs to be fixed to be only relative to the same CPU. The marks are determined by the difference between this current trace and the next trace.

- '$' - greater than 1 second
- '@' - greater than 100 millisecond
- '*' - greater than 10 millisecond
- '#' - greater than 1000 microsecond
- '!' - greater than 100 microsecond
- '+' - greater than 10 microsecond
- ' '- less than or equal to 10 microsecond.

The rest is the same as the 'trace' file.

Note, the latency tracers will usually end with a back trace to easily find where the latency occurred.

3.9 trace_options

The trace_options file (or the options directory) is used to control what gets printed in the trace output, or manipulate the tracers. To see what is available, simply cat the file:

```
cat trace_options
      print-parent
      nosym-offset
      nosym-addr
      noverbose
      noraw
      nohex
      nobin
      noblock
      trace_printk
      annotate
      nouserstacktrace
      nosym-userobj
      noprintk-msg-only
      context-info
      nolatency-format
      record-cmd
      norecord-tgid
```

(continues on next page)

(continued from previous page)

```
      overwrite
      nodisable_on_free
      irq-info
      markers
      noevent-fork
      function-trace
      nofunction-fork
      nodisplay-graph
      nostacktrace
      nobranch
```

To disable one of the options, echo in the option prepended with "no":

```
echo noprint-parent > trace_options
```

To enable an option, leave off the "no":

```
echo sym-offset > trace_options
```

Here are the available options:

print-parent
On function traces, display the calling (parent) function as well as the function being traced.

```
print-parent:
 bash-4000  [01]  1477.606694: simple_strtoul <-kstrtoul

noprint-parent:
 bash-4000  [01]  1477.606694: simple_strtoul
```

sym-offset
Display not only the function name, but also the offset in the function. For example, instead of seeing just "ktime_get", you will see "ktime_get+0xb/0x20".

```
sym-offset:
 bash-4000  [01]  1477.606694: simple_strtoul+0x6/0xa0
```

sym-addr
This will also display the function address as well as the function name.

```
sym-addr:
 bash-4000  [01]  1477.606694: simple_strtoul <c0339346>
```

verbose
This deals with the trace file when the latency-format option is enabled.

```
bash  4000 1 0 00000000 00010a95 [58127d26] 1720.415ms \
(+0.000ms): simple_strtoul (kstrtoul)
```

raw

This will display raw numbers. This option is best for use with user applications that can translate the raw numbers better than having it done in the kernel.

hex

Similar to raw, but the numbers will be in a hexadecimal format.

bin

This will print out the formats in raw binary.

block

When set, reading trace_pipe will not block when polled.

trace_printk

Can disable trace_printk() from writing into the buffer.

annotate

It is sometimes confusing when the CPU buffers are full and one CPU buffer had a lot of events recently, thus a shorter time frame, were another CPU may have only had a few events, which lets it have older events. When the trace is reported, it shows the oldest events first, and it may look like only one CPU ran (the one with the oldest events). When the annotate option is set, it will display when a new CPU buffer started:

```
          <idle>-0     [001] dNs4 21169.031481: wake_up_
↪idle_cpu <-add_timer_on
          <idle>-0     [001] dNs4 21169.031482: _raw_
↪spin_unlock_irqrestore <-add_timer_on
          <idle>-0     [001] .Ns4 21169.031484: sub_
↪preempt_count <-_raw_spin_unlock_irqrestore
##### CPU 2 buffer started ####
          <idle>-0     [002] .N.1 21169.031484: rcu_
↪idle_exit <-cpu_idle
          <idle>-0     [001] .Ns3 21169.031484: _raw_
↪spin_unlock <-clocksource_watchdog
          <idle>-0     [001] .Ns3 21169.031485: sub_
↪preempt_count <-_raw_spin_unlock
```

userstacktrace

This option changes the trace. It records a stacktrace of the current user space thread after each trace event.

sym-userobj

when user stacktrace are enabled, look up which object the address belongs to, and print a relative address. This is especially useful when ASLR is on, otherwise you don’t get a chance to resolve the address to object/file/line after the app is no longer running

The lookup is performed when you read trace,trace_pipe. Example:

```
a.out-1623  [000] 40874.465068: /root/a.out[+0x480] <-/
↪root/a.out[+0
```

(continues on next page)

(continued from previous page)

```
x494] <- /root/a.out[+0x4a8] <- /lib/libc-2.7.
↪so[+0x1e1a6]
```

printk-msg-only
When set, trace_printk()s will only show the format and not their parameters (if trace_bprintk() or trace_bputs() was used to save the trace_printk()).

context-info
Show only the event data. Hides the comm, PID, timestamp, CPU, and other useful data.

latency-format
This option changes the trace output. When it is enabled, the trace displays additional information about the latency, as described in "Latency trace format" .

pause-on-trace
When set, opening the trace file for read, will pause writing to the ring buffer (as if tracing_on was set to zero). This simulates the original behavior of the trace file. When the file is closed, tracing will be enabled again.

record-cmd
When any event or tracer is enabled, a hook is enabled in the sched_switch trace point to fill comm cache with mapped pids and comms. But this may cause some overhead, and if you only care about pids, and not the name of the task, disabling this option can lower the impact of tracing. See "saved_cmdlines" .

record-tgid
When any event or tracer is enabled, a hook is enabled in the sched_switch trace point to fill the cache of mapped Thread Group IDs (TGID) mapping to pids. See "saved_tgids" .

overwrite
This controls what happens when the trace buffer is full. If "1" (default), the oldest events are discarded and overwritten. If "0" , then the newest events are discarded. (see per_cpu/cpu0/stats for overrun and dropped)

disable_on_free
When the free_buffer is closed, tracing will stop (tracing_on set to 0).

irq-info
Shows the interrupt, preempt count, need resched data. When disabled, the trace looks like:

```
# tracer: function
#
# entries-in-buffer/entries-written: 144405/9452052
↪#P:4
```

(continues on next page)

(continued from previous page)

```
#
#           TASK-PID   CPU#      TIMESTAMP  FUNCTION
#              | |       |          |         |
          <idle>-0     [002]  23636.756054: ttwu_do_
↪activate.constprop.89 <-try_to_wake_up
          <idle>-0     [002]  23636.756054: activate_
↪task <-ttwu_do_activate.constprop.89
          <idle>-0     [002]  23636.756055: enqueue_
↪task <-activate_task
```

markers
When set, the trace_marker is writable (only by root). When disabled, the trace_marker will error with EINVAL on write.

event-fork
When set, tasks with PIDs listed in set_event_pid will have the PIDs of their children added to set_event_pid when those tasks fork. Also, when tasks with PIDs in set_event_pid exit, their PIDs will be removed from the file.

This affects PIDs listed in set_event_notrace_pid as well.

function-trace
The latency tracers will enable function tracing if this option is enabled (default it is). When it is disabled, the latency tracers do not trace functions. This keeps the overhead of the tracer down when performing latency tests.

function-fork
When set, tasks with PIDs listed in set_ftrace_pid will have the PIDs of their children added to set_ftrace_pid when those tasks fork. Also, when tasks with PIDs in set_ftrace_pid exit, their PIDs will be removed from the file.

This affects PIDs in set_ftrace_notrace_pid as well.

display-graph
When set, the latency tracers (irqsoff, wakeup, etc) will use function graph tracing instead of function tracing.

stacktrace
When set, a stack trace is recorded after any trace event is recorded.

branch
Enable branch tracing with the tracer. This enables branch tracer along with the currently set tracer. Enabling this with the "nop" tracer is the same as just enabling the "branch" tracer.

Tip: Some tracers have their own options. They only appear in this file when the tracer is active. They always appear in the options directory.

Here are the per tracer options:

Options for function tracer:

func_stack_trace
When set, a stack trace is recorded after every function that is recorded. NOTE! Limit the functions that are recorded before enabling this, with "set_ftrace_filter" otherwise the system performance will be critically degraded. Remember to disable this option before clearing the function filter.

Options for function_graph tracer:

Since the function_graph tracer has a slightly different output it has its own options to control what is displayed.

funcgraph-overrun
When set, the "overrun" of the graph stack is displayed after each function traced. The overrun, is when the stack depth of the calls is greater than what is reserved for each task. Each task has a fixed array of functions to trace in the call graph. If the depth of the calls exceeds that, the function is not traced. The overrun is the number of functions missed due to exceeding this array.

funcgraph-cpu
When set, the CPU number of the CPU where the trace occurred is displayed.

funcgraph-overhead
When set, if the function takes longer than A certain amount, then a delay marker is displayed. See "delay"above, under the header description.

funcgraph-proc
Unlike other tracers, the process' command line is not displayed by default, but instead only when a task is traced in and out during a context switch. Enabling this options has the command of each process displayed at every line.

funcgraph-duration
At the end of each function (the return) the duration of the amount of time in the function is displayed in microseconds.

funcgraph-abstime
When set, the timestamp is displayed at each line.

funcgraph-irqs
When disabled, functions that happen inside an interrupt will not be traced.

funcgraph-tail
When set, the return event will include the function that it represents. By default this is off, and only a closing curly bracket "}" is displayed for the return of a function.

sleep-time
When running function graph tracer, to include the time a

task schedules out in its function. When enabled, it will account time the task has been scheduled out as part of the function call.

graph-time
When running function profiler with function graph tracer, to include the time to call nested functions. When this is not set, the time reported for the function will only include the time the function itself executed for, not the time for functions that it called.

Options for blk tracer:

blk_classic
Shows a more minimalistic output.

3.10 irqsoff

When interrupts are disabled, the CPU can not react to any other external event (besides NMIs and SMIs). This prevents the timer interrupt from triggering or the mouse interrupt from letting the kernel know of a new mouse event. The result is a latency with the reaction time.

The irqsoff tracer tracks the time for which interrupts are disabled. When a new maximum latency is hit, the tracer saves the trace leading up to that latency point so that every time a new maximum is reached, the old saved trace is discarded and the new trace is saved.

To reset the maximum, echo 0 into tracing_max_latency. Here is an example:

```
# echo 0 > options/function-trace
# echo irqsoff > current_tracer
# echo 1 > tracing_on
# echo 0 > tracing_max_latency
# ls -ltr
[...]
# echo 0 > tracing_on
# cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.8.0-test+
# ------------------------------------------------------------------
↪--
# latency: 16 us, #4/4, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0
↪#P:4)
#    -----------------
#    | task: swapper/0-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: run_timer_softirq
#  => ended at:   run_timer_softirq
#
#
```

(continues on next page)

(continued from previous page)

```
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>-0       0d.s2    0us+: _raw_spin_lock_irq <-run_timer_
↪softirq
  <idle>-0       0dNs3   17us : _raw_spin_unlock_irq <-run_timer_
↪softirq
  <idle>-0       0dNs3   17us+: trace_hardirqs_on <-run_timer_
↪softirq
  <idle>-0       0dNs3   25us : <stack trace>
 => _raw_spin_unlock_irq
 => run_timer_softirq
 => __do_softirq
 => call_softirq
 => do_softirq
 => irq_exit
 => smp_apic_timer_interrupt
 => apic_timer_interrupt
 => rcu_idle_exit
 => cpu_idle
 => rest_init
 => start_kernel
 => x86_64_start_reservations
 => x86_64_start_kernel
```

Here we see that we had a latency of 16 microseconds (which is very good). The _raw_spin_lock_irq in run_timer_softirq disabled interrupts. The difference between the 16 and the displayed timestamp 25us occurred because the clock was incremented between the time of recording the max latency and the time of recording the function that had that latency.

Note the above example had function-trace not set. If we set function-trace, we get a much larger output:

```
with echo 1 > options/function-trace

 # tracer: irqsoff
 #
 # irqsoff latency trace v1.1.5 on 3.8.0-test+
 # --------------------------------------------------------------------
↪---
 # latency: 71 us, #168/168, CPU#3 | (M:preempt VP:0, KP:0, SP:0␣
↪HP:0 #P:4)
 #    -----------------
 #    | task: bash-2042 (uid:0 nice:0 policy:0 rt_prio:0)
```

(continues on next page)

(continued from previous page)

```
#    -----------------
#  => started at: ata_scsi_queuecmd
#  => ended at:   ata_scsi_queuecmd
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
    bash-2042    3d...    0us : _raw_spin_lock_irqsave <-ata_scsi_
↪queuecmd
    bash-2042    3d...    0us : add_preempt_count <-_raw_spin_lock_
↪irqsave
    bash-2042    3d..1    1us : ata_scsi_find_dev <-ata_scsi_
↪queuecmd
    bash-2042    3d..1    1us : __ata_scsi_find_dev <-ata_scsi_
↪find_dev
    bash-2042    3d..1    2us : ata_find_dev.part.14 <-__ata_scsi_
↪find_dev
    bash-2042    3d..1    2us : ata_qc_new_init <-__ata_scsi_
↪queuecmd
    bash-2042    3d..1    3us : ata_sg_init <-__ata_scsi_queuecmd
    bash-2042    3d..1    4us : ata_scsi_rw_xlat <-__ata_scsi_
↪queuecmd
    bash-2042    3d..1    4us : ata_build_rw_tf <-ata_scsi_rw_xlat
[...]
    bash-2042    3d..1   67us : delay_tsc <-__delay
    bash-2042    3d..1   67us : add_preempt_count <-delay_tsc
    bash-2042    3d..2   67us : sub_preempt_count <-delay_tsc
    bash-2042    3d..1   67us : add_preempt_count <-delay_tsc
    bash-2042    3d..2   68us : sub_preempt_count <-delay_tsc
    bash-2042    3d..1   68us+: ata_bmdma_start <-ata_bmdma_qc_
↪issue
    bash-2042    3d..1   71us : _raw_spin_unlock_irqrestore <-ata_
↪scsi_queuecmd
    bash-2042    3d..1   71us : _raw_spin_unlock_irqrestore <-ata_
↪scsi_queuecmd
    bash-2042    3d..1   72us+: trace_hardirqs_on <-ata_scsi_
↪queuecmd
    bash-2042    3d..1  120us : <stack trace>
 => _raw_spin_unlock_irqrestore
 => ata_scsi_queuecmd
 => scsi_dispatch_cmd
 => scsi_request_fn
 => __blk_run_queue_uncond
```

(continues on next page)

(continued from previous page)

```
  => __blk_run_queue
  => blk_queue_bio
  => submit_bio_noacct
  => submit_bio
  => submit_bh
  => __ext3_get_inode_loc
  => ext3_iget
  => ext3_lookup
  => lookup_real
  => __lookup_hash
  => walk_component
  => lookup_last
  => path_lookupat
  => filename_lookup
  => user_path_at_empty
  => user_path_at
  => vfs_fstatat
  => vfs_stat
  => sys_newstat
  => system_call_fastpath
```

Here we traced a 71 microsecond latency. But we also see all the functions that were called during that time. Note that by enabling function tracing, we incur an added overhead. This overhead may extend the latency times. But nevertheless, this trace has provided some very helpful debugging information.

If we prefer function graph output instead of function, we can set display-graph option:

```
with echo 1 > options/display-graph

 # tracer: irqsoff
 #
 # irqsoff latency trace v1.1.5 on 4.20.0-rc6+
 # ------------------------------------------------------------------
↪---
 # latency: 3751 us, #274/274, CPU#0 | (M:desktop VP:0, KP:0, SP:0␣
↪HP:0 #P:4)
 #    -----------------
 #    | task: bash-1507 (uid:0 nice:0 policy:0 rt_prio:0)
 #    -----------------
 #  => started at: free_debug_processing
 #  => ended at:   return_to_handler
 #
 #
 #                                       _-----=> irqs-off
 #                                      / _----=> need-resched
 #                                     | / _---=> hardirq/softirq
 #                                     || / _--=> preempt-depth
 #                                     ||| /
```

(continues on next page)

(continued from previous page)

```
#   REL TIME      CPU  TASK/PID       ||||     DURATION                  ␣
↪      FUNCTION CALLS
#      |          |     |    |        ||||      |   |                    ␣
↪       |   |   |   |
        0 us |   0)   bash-1507    | d... |   0.000 us    |  _raw_
↪spin_lock_irqsave();
        0 us |   0)   bash-1507    | d..1 |   0.378 us    |    do_
↪raw_spin_trylock();
        1 us |   0)   bash-1507    | d..2 |               |  ␣
↪set_track() {
        2 us |   0)   bash-1507    | d..2 |               |    ␣
↪save_stack_trace() {
        2 us |   0)   bash-1507    | d..2 |               |     ␣
↪ __save_stack_trace() {
        3 us |   0)   bash-1507    | d..2 |               |     ␣
↪   __unwind_start() {
        3 us |   0)   bash-1507    | d..2 |               |     ␣
↪     get_stack_info() {
        3 us |   0)   bash-1507    | d..2 |   0.351 us    |     ␣
↪       in_task_stack();
        4 us |   0)   bash-1507    | d..2 |   1.107 us    |     ␣
↪     }
[...]
     3750 us |   0)   bash-1507    | d..1 |   0.516 us    |    ␣
↪do_raw_spin_unlock();
     3750 us |   0)   bash-1507    | d..1 |   0.000 us    |  _raw_
↪spin_unlock_irqrestore();
     3764 us |   0)   bash-1507    | d..1 |   0.000 us    | ␣
↪tracer_hardirqs_on();
    bash-1507    0d..1 3792us : <stack trace>
 => free_debug_processing
 => __slab_free
 => kmem_cache_free
 => vm_area_free
 => remove_vma
 => exit_mmap
 => mmput
 => begin_new_exec
 => load_elf_binary
 => search_binary_handler
 => __do_execve_file.isra.32
 => __x64_sys_execve
 => do_syscall_64
 => entry_SYSCALL_64_after_hwframe
```

3.11 preemptoff

When preemption is disabled, we may be able to receive interrupts but the task cannot be preempted and a higher priority task must wait for preemption to be enabled again before it can preempt a lower priority task.

The preemptoff tracer traces the places that disable preemption. Like the irqsoff tracer, it records the maximum latency for which preemption was disabled. The control of preemptoff tracer is much like the irqsoff tracer.

```
# echo 0 > options/function-trace
# echo preemptoff > current_tracer
# echo 1 > tracing_on
# echo 0 > tracing_max_latency
# ls -ltr
[...]
# echo 0 > tracing_on
# cat trace
# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.8.0-test+
# -------------------------------------------------------------------
↪--
# latency: 46 us, #4/4, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0
↪#P:4)
#    -----------------
#    | task: sshd-1991 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: do_IRQ
#  => ended at:   do_IRQ
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
    sshd-1991    1d.h.    0us+: irq_enter <-do_IRQ
    sshd-1991    1d..1   46us : irq_exit <-do_IRQ
    sshd-1991    1d..1   47us+: trace_preempt_on <-do_IRQ
    sshd-1991    1d..1   52us : <stack trace>
 => sub_preempt_count
 => irq_exit
 => do_IRQ
 => ret_from_intr
```

This has some more changes. Preemption was disabled when an interrupt came in (notice the ‘h’), and was enabled on exit. But we also see that interrupts have been disabled when entering the preempt off section and leaving it (the 'd'). We do not know if interrupts were enabled in the mean time or shortly after this was over.

```
# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.8.0-test+
# --------------------------------------------------------------------
↪--
# latency: 83 us, #241/241, CPU#1 | (M:preempt VP:0, KP:0, SP:0␣
↪HP:0 #P:4)
#    -----------------
#    | task: bash-1994 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: wake_up_new_task
#  => ended at:   task_rq_unlock
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
    bash-1994    1d..1    0us : _raw_spin_lock_irqsave <-wake_up_
↪new_task
    bash-1994    1d..1    0us : select_task_rq_fair <-select_task_rq
    bash-1994    1d..1    1us : __rcu_read_lock <-select_task_rq_
↪fair
    bash-1994    1d..1    1us : source_load <-select_task_rq_fair
    bash-1994    1d..1    1us : source_load <-select_task_rq_fair
[...]
    bash-1994    1d..1   12us : irq_enter <-smp_apic_timer_interrupt
    bash-1994    1d..1   12us : rcu_irq_enter <-irq_enter
    bash-1994    1d..1   13us : add_preempt_count <-irq_enter
    bash-1994    1d.h1   13us : exit_idle <-smp_apic_timer_interrupt
    bash-1994    1d.h1   13us : hrtimer_interrupt <-smp_apic_timer_
↪interrupt
    bash-1994    1d.h1   13us : _raw_spin_lock <-hrtimer_interrupt
    bash-1994    1d.h1   14us : add_preempt_count <-_raw_spin_lock
    bash-1994    1d.h2   14us : ktime_get_update_offsets <-hrtimer_
↪interrupt
[...]
    bash-1994    1d.h1   35us : lapic_next_event <-clockevents_
↪program_event
    bash-1994    1d.h1   35us : irq_exit <-smp_apic_timer_interrupt
    bash-1994    1d.h1   36us : sub_preempt_count <-irq_exit
    bash-1994    1d..2   36us : do_softirq <-irq_exit
    bash-1994    1d..2   36us : __do_softirq <-call_softirq
```

(continues on next page)

(continued from previous page)

```
    bash-1994    1d..2   36us : __local_bh_disable <-__do_softirq
    bash-1994    1d.s2   37us : add_preempt_count <-_raw_spin_lock_
↪irq
    bash-1994    1d.s3   38us : _raw_spin_unlock <-run_timer_softirq
    bash-1994    1d.s3   39us : sub_preempt_count <-_raw_spin_unlock
    bash-1994    1d.s2   39us : call_timer_fn <-run_timer_softirq
[...]
    bash-1994    1dNs2   81us : cpu_needs_another_gp <-rcu_process_
↪callbacks
    bash-1994    1dNs2   82us : __local_bh_enable <-__do_softirq
    bash-1994    1dNs2   82us : sub_preempt_count <-__local_bh_
↪enable
    bash-1994    1dN.2   82us : idle_cpu <-irq_exit
    bash-1994    1dN.2   83us : rcu_irq_exit <-irq_exit
    bash-1994    1dN.2   83us : sub_preempt_count <-irq_exit
    bash-1994    1.N.1   84us : _raw_spin_unlock_irqrestore <-task_
↪rq_unlock
    bash-1994    1.N.1   84us+: trace_preempt_on <-task_rq_unlock
    bash-1994    1.N.1  104us : <stack trace>
 => sub_preempt_count
 => _raw_spin_unlock_irqrestore
 => task_rq_unlock
 => wake_up_new_task
 => do_fork
 => sys_clone
 => stub_clone
```

The above is an example of the preemptoff trace with function-trace set. Here we see that interrupts were not disabled the entire time. The irq_enter code lets us know that we entered an interrupt 'h'. Before that, the functions being traced still show that it is not in an interrupt, but we can see from the functions themselves that this is not the case.

3.12 preemptirqsoff

Knowing the locations that have interrupts disabled or preemption disabled for the longest times is helpful. But sometimes we would like to know when either preemption and/or interrupts are disabled.

Consider the following code:

```
local_irq_disable();
call_function_with_irqs_off();
preempt_disable();
call_function_with_irqs_and_preemption_off();
local_irq_enable();
call_function_with_preemption_off();
preempt_enable();
```

The irqsoff tracer will record the total length of call_function_with_irqs_off() and call_function_with_irqs_and_preemption_off().

The preemptoff tracer will record the total length of call_function_with_irqs_and_preemption_off() and call_function_with_preemption_off().

But neither will trace the time that interrupts and/or preemption is disabled. This total time is the time that we can not schedule. To record this time, use the preemptirqsoff tracer.

Again, using this trace is much like the irqsoff and preemptoff tracers.

```
# echo 0 > options/function-trace
# echo preemptirqsoff > current_tracer
# echo 1 > tracing_on
# echo 0 > tracing_max_latency
# ls -ltr
[...]
# echo 0 > tracing_on
# cat trace
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.8.0-test+
# --------------------------------------------------------------------
↪--
# latency: 100 us, #4/4, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0
↪#P:4)
#    -----------------
#    | task: ls-2230 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: ata_scsi_queuecmd
#  => ended at:   ata_scsi_queuecmd
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
      ls-2230    3d...    0us+: _raw_spin_lock_irqsave <-ata_scsi_
↪queuecmd
      ls-2230    3...1  100us : _raw_spin_unlock_irqrestore <-ata_
↪scsi_queuecmd
      ls-2230    3...1  101us+: trace_preempt_on <-ata_scsi_queuecmd
      ls-2230    3...1  111us : <stack trace>
 => sub_preempt_count
 => _raw_spin_unlock_irqrestore
 => ata_scsi_queuecmd
 => scsi_dispatch_cmd
```

(continues on next page)

(continued from previous page)

```
=> scsi_request_fn
=> __blk_run_queue_uncond
=> __blk_run_queue
=> blk_queue_bio
=> submit_bio_noacct
=> submit_bio
=> submit_bh
=> ext3_bread
=> ext3_dir_bread
=> htree_dirblock_to_tree
=> ext3_htree_fill_tree
=> ext3_readdir
=> vfs_readdir
=> sys_getdents
=> system_call_fastpath
```

The trace_hardirqs_off_thunk is called from assembly on x86 when interrupts are disabled in the assembly code. Without the function tracing, we do not know if interrupts were enabled within the preemption points. We do see that it started with preemption enabled.

Here is a trace with function-trace set:

```
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.8.0-test+
# -------------------------------------------------------------------
↪--
# latency: 161 us, #339/339, CPU#3 | (M:preempt VP:0, KP:0, SP:0␣
↪HP:0 #P:4)
#    -----------------
#    | task: ls-2269 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: schedule
#  => ended at:   mutex_unlock
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
kworker/-59      3...1    0us : __schedule <-schedule
kworker/-59      3d..1    0us : rcu_preempt_qs <-rcu_note_context_
↪switch
kworker/-59      3d..1    1us : add_preempt_count <-_raw_spin_lock_
↪irq
```

(continues on next page)

(continued from previous page)

```
kworker/-59      3d..2    1us : deactivate_task <-__schedule
kworker/-59      3d..2    1us : dequeue_task <-deactivate_task
kworker/-59      3d..2    2us : update_rq_clock <-dequeue_task
kworker/-59      3d..2    2us : dequeue_task_fair <-dequeue_task
kworker/-59      3d..2    2us : update_curr <-dequeue_task_fair
kworker/-59      3d..2    2us : update_min_vruntime <-update_curr
kworker/-59      3d..2    3us : cpuacct_charge <-update_curr
kworker/-59      3d..2    3us : __rcu_read_lock <-cpuacct_charge
kworker/-59      3d..2    3us : __rcu_read_unlock <-cpuacct_charge
kworker/-59      3d..2    3us : update_cfs_rq_blocked_load <-
↪dequeue_task_fair
kworker/-59      3d..2    4us : clear_buddies <-dequeue_task_fair
kworker/-59      3d..2    4us : account_entity_dequeue <-dequeue_
↪task_fair
kworker/-59      3d..2    4us : update_min_vruntime <-dequeue_task_
↪fair
kworker/-59      3d..2    4us : update_cfs_shares <-dequeue_task_
↪fair
kworker/-59      3d..2    5us : hrtick_update <-dequeue_task_fair
kworker/-59      3d..2    5us : wq_worker_sleeping <-__schedule
kworker/-59      3d..2    5us : kthread_data <-wq_worker_sleeping
kworker/-59      3d..2    5us : put_prev_task_fair <-__schedule
kworker/-59      3d..2    6us : pick_next_task_fair <-pick_next_task
kworker/-59      3d..2    6us : clear_buddies <-pick_next_task_fair
kworker/-59      3d..2    6us : set_next_entity <-pick_next_task_
↪fair
kworker/-59      3d..2    6us : update_stats_wait_end <-set_next_
↪entity
      ls-2269    3d..2    7us : finish_task_switch <-__schedule
      ls-2269    3d..2    7us : _raw_spin_unlock_irq <-finish_task_
↪switch
      ls-2269    3d..2    8us : do_IRQ <-ret_from_intr
      ls-2269    3d..2    8us : irq_enter <-do_IRQ
      ls-2269    3d..2    8us : rcu_irq_enter <-irq_enter
      ls-2269    3d..2    9us : add_preempt_count <-irq_enter
      ls-2269    3d.h2    9us : exit_idle <-do_IRQ
[...]
      ls-2269    3d.h3   20us : sub_preempt_count <-_raw_spin_unlock
      ls-2269    3d.h2   20us : irq_exit <-do_IRQ
      ls-2269    3d.h2   21us : sub_preempt_count <-irq_exit
      ls-2269    3d..3   21us : do_softirq <-irq_exit
      ls-2269    3d..3   21us : __do_softirq <-call_softirq
      ls-2269    3d..3   21us+: __local_bh_disable <-__do_softirq
      ls-2269    3d.s4   29us : sub_preempt_count <-_local_bh_
↪enable_ip
      ls-2269    3d.s5   29us : sub_preempt_count <-_local_bh_
↪enable_ip
      ls-2269    3d.s5   31us : do_IRQ <-ret_from_intr
      ls-2269    3d.s5   31us : irq_enter <-do_IRQ
```

(continues on next page)

(continued from previous page)

```
      ls-2269    3d.s5   31us : rcu_irq_enter <-irq_enter
[...]
      ls-2269    3d.s5   31us : rcu_irq_enter <-irq_enter
      ls-2269    3d.s5   32us : add_preempt_count <-irq_enter
      ls-2269    3d.H5   32us : exit_idle <-do_IRQ
      ls-2269    3d.H5   32us : handle_irq <-do_IRQ
      ls-2269    3d.H5   32us : irq_to_desc <-handle_irq
      ls-2269    3d.H5   33us : handle_fasteoi_irq <-handle_irq
[...]
      ls-2269    3d.s5  158us : _raw_spin_unlock_irqrestore <-
↪rtl8139_poll
      ls-2269    3d.s3  158us : net_rps_action_and_irq_enable.isra.
↪65 <-net_rx_action
      ls-2269    3d.s3  159us : __local_bh_enable <-__do_softirq
      ls-2269    3d.s3  159us : sub_preempt_count <-__local_bh_
↪enable
      ls-2269    3d..3  159us : idle_cpu <-irq_exit
      ls-2269    3d..3  159us : rcu_irq_exit <-irq_exit
      ls-2269    3d..3  160us : sub_preempt_count <-irq_exit
      ls-2269    3d...  161us : __mutex_unlock_slowpath <-mutex_
↪unlock
      ls-2269    3d...  162us+: trace_hardirqs_on <-mutex_unlock
      ls-2269    3d...  186us : <stack trace>
 => __mutex_unlock_slowpath
 => mutex_unlock
 => process_output
 => n_tty_write
 => tty_write
 => vfs_write
 => sys_write
 => system_call_fastpath
```

This is an interesting trace. It started with kworker running and scheduling out and ls taking over. But as soon as ls released the rq lock and enabled interrupts (but not preemption) an interrupt triggered. When the interrupt finished, it started running softirqs. But while the softirq was running, another interrupt triggered. When an interrupt is running inside a softirq, the annotation is 'H'.

3.13 wakeup

One common case that people are interested in tracing is the time it takes for a task that is woken to actually wake up. Now for non Real-Time tasks, this can be arbitrary. But tracing it none the less can be interesting.

Without function tracing:

```
# echo 0 > options/function-trace
# echo wakeup > current_tracer
# echo 1 > tracing_on
```

(continues on next page)

(continued from previous page)

```
# echo 0 > tracing_max_latency
# chrt -f 5 sleep 1
# echo 0 > tracing_on
# cat trace
# tracer: wakeup
#
# wakeup latency trace v1.1.5 on 3.8.0-test+
# -----------------------------------------------------------------
↪--
# latency: 15 us, #4/4, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0
↪#P:4)
#    -----------------
#    | task: kworker/3:1H-312 (uid:0 nice:-20 policy:0 rt_prio:0)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>-0       3dNs7    0us :      0:120:R   + [003]   312:100:R␣
↪kworker/3:1H
  <idle>-0       3dNs7    1us+: ttwu_do_activate.constprop.87 <-try_
↪to_wake_up
  <idle>-0       3d..3   15us : __schedule <-schedule
  <idle>-0       3d..3   15us :      0:120:R ==> [003]   312:100:R␣
↪kworker/3:1H
```

The tracer only traces the highest priority task in the system to avoid tracing the normal circumstances. Here we see that the kworker with a nice priority of -20 (not very nice), took just 15 microseconds from the time it woke up, to the time it ran.

Non Real-Time tasks are not that interesting. A more interesting trace is to concentrate only on Real-Time tasks.

3.14 wakeup_rt

In a Real-Time environment it is very important to know the wakeup time it takes for the highest priority task that is woken up to the time that it executes. This is also known as "schedule latency". I stress the point that this is about RT tasks. It is also important to know the scheduling latency of non-RT tasks, but the average schedule latency is better for non-RT tasks. Tools like LatencyTop are more appropriate for such measurements.

Real-Time environments are interested in the worst case latency. That is the longest latency it takes for something to happen, and not the average. We can have a very fast scheduler that may only have a large latency once in a while, but that would not work well with Real-Time tasks. The wakeup_rt tracer was designed to record the worst case wakeups of RT tasks. Non-RT tasks are not recorded because the tracer only records one worst case and tracing non-RT tasks that are unpredictable will overwrite the worst case latency of RT tasks (just run the normal wakeup tracer for a while to see that effect).

Since this tracer only deals with RT tasks, we will run this slightly differently than we did with the previous tracers. Instead of performing an 'ls', we will run 'sleep 1' under 'chrt' which changes the priority of the task.

```
# echo 0 > options/function-trace
# echo wakeup_rt > current_tracer
# echo 1 > tracing_on
# echo 0 > tracing_max_latency
# chrt -f 5 sleep 1
# echo 0 > tracing_on
# cat trace
# tracer: wakeup
#
# tracer: wakeup_rt
#
# wakeup_rt latency trace v1.1.5 on 3.8.0-test+
# -------------------------------------------------------------------
↪--
# latency: 5 us, #4/4, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0
↪#P:4)
#    -----------------
#    | task: sleep-2389 (uid:0 nice:0 policy:1 rt_prio:5)
#    -----------------
#
#                   _------=> CPU#
#                  / _-----=> irqs-off
#                 | / _----=> need-resched
#                 || / _---=> hardirq/softirq
#                 ||| / _--=> preempt-depth
#                 |||| /     delay
#  cmd     pid    ||||| time  |   caller
#     \   /       |||||  \    |   /
  <idle>-0       3d.h4    0us :      0:120:R   + [003]  2389: 94:R␣
↪sleep
  <idle>-0       3d.h4    1us+: ttwu_do_activate.constprop.87 <-try_
↪to_wake_up
  <idle>-0       3d..3    5us : __schedule <-schedule
  <idle>-0       3d..3    5us :      0:120:R ==> [003]  2389: 94:R␣
↪sleep
```

Running this on an idle system, we see that it only took 5 microseconds to perform the task switch. Note, since the trace point in the schedule is before the actual "switch", we stop the tracing when the recorded task is about to schedule in. This may change if we add a new marker at the end of the scheduler.

Notice that the recorded task is 'sleep' with the PID of 2389 and it has an rt_prio of 5. This priority is user-space priority and not the internal kernel priority. The policy is 1 for SCHED_FIFO and 2 for SCHED_RR.

Note, that the trace data shows the internal priority (99 - rtprio).

```
<idle>-0       3d..3    5us :      0:120:R ==> [003]  2389: 94:R␣
↪sleep
```

The 0:120:R means idle was running with a nice priority of 0 (120 - 120) and in the running state 'R'. The sleep task was scheduled in with 2389: 94:R. That is the priority is the kernel rtprio (99 - 5 = 94) and it too is in the running state.

Doing the same with chrt -r 5 and function-trace set.

```
echo 1 > options/function-trace

# tracer: wakeup_rt
#
# wakeup_rt latency trace v1.1.5 on 3.8.0-test+
# ------------------------------------------------------------------
↪--
# latency: 29 us, #85/85, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0
↪#P:4)
#    -----------------
#    | task: sleep-2448 (uid:0 nice:0 policy:1 rt_prio:5)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>-0       3d.h4    1us+:      0:120:R   + [003]  2448: 94:R␣
↪sleep
  <idle>-0       3d.h4    2us : ttwu_do_activate.constprop.87 <-try_
↪to_wake_up
  <idle>-0       3d.h3    3us : check_preempt_curr <-ttwu_do_wakeup
  <idle>-0       3d.h3    3us : resched_curr <-check_preempt_curr
  <idle>-0       3dNh3    4us : task_woken_rt <-ttwu_do_wakeup
  <idle>-0       3dNh3    4us : _raw_spin_unlock <-try_to_wake_up
  <idle>-0       3dNh3    4us : sub_preempt_count <-_raw_spin_unlock
  <idle>-0       3dNh2    5us : ttwu_stat <-try_to_wake_up
  <idle>-0       3dNh2    5us : _raw_spin_unlock_irqrestore <-try_
↪to_wake_up
  <idle>-0       3dNh2    6us : sub_preempt_count <-_raw_spin_
↪unlock_irqrestore
  <idle>-0       3dNh1    6us : _raw_spin_lock <-__run_hrtimer
  <idle>-0       3dNh1    6us : add_preempt_count <-_raw_spin_lock
```

(continues on next page)

(continued from previous page)

```
  <idle>-0       3dNh2    7us : _raw_spin_unlock <-hrtimer_interrupt
  <idle>-0       3dNh2    7us : sub_preempt_count <-_raw_spin_unlock
  <idle>-0       3dNh1    7us : tick_program_event <-hrtimer_
↪interrupt
  <idle>-0       3dNh1    7us : clockevents_program_event <-tick_
↪program_event
  <idle>-0       3dNh1    8us : ktime_get <-clockevents_program_
↪event
  <idle>-0       3dNh1    8us : lapic_next_event <-clockevents_
↪program_event
  <idle>-0       3dNh1    8us : irq_exit <-smp_apic_timer_interrupt
  <idle>-0       3dNh1    9us : sub_preempt_count <-irq_exit
  <idle>-0       3dN.2    9us : idle_cpu <-irq_exit
  <idle>-0       3dN.2    9us : rcu_irq_exit <-irq_exit
  <idle>-0       3dN.2   10us : rcu_eqs_enter_common.isra.45 <-rcu_
↪irq_exit
  <idle>-0       3dN.2   10us : sub_preempt_count <-irq_exit
  <idle>-0       3.N.1   11us : rcu_idle_exit <-cpu_idle
  <idle>-0       3dN.1   11us : rcu_eqs_exit_common.isra.43 <-rcu_
↪idle_exit
  <idle>-0       3.N.1   11us : tick_nohz_idle_exit <-cpu_idle
  <idle>-0       3dN.1   12us : menu_hrtimer_cancel <-tick_nohz_
↪idle_exit
  <idle>-0       3dN.1   12us : ktime_get <-tick_nohz_idle_exit
  <idle>-0       3dN.1   12us : tick_do_update_jiffies64 <-tick_
↪nohz_idle_exit
  <idle>-0       3dN.1   13us : cpu_load_update_nohz <-tick_nohz_
↪idle_exit
  <idle>-0       3dN.1   13us : _raw_spin_lock <-cpu_load_update_
↪nohz
  <idle>-0       3dN.1   13us : add_preempt_count <-_raw_spin_lock
  <idle>-0       3dN.2   13us : __cpu_load_update <-cpu_load_update_
↪nohz
  <idle>-0       3dN.2   14us : sched_avg_update <-__cpu_load_update
  <idle>-0       3dN.2   14us : _raw_spin_unlock <-cpu_load_update_
↪nohz
  <idle>-0       3dN.2   14us : sub_preempt_count <-_raw_spin_unlock
  <idle>-0       3dN.1   15us : calc_load_nohz_stop <-tick_nohz_
↪idle_exit
  <idle>-0       3dN.1   15us : touch_softlockup_watchdog <-tick_
↪nohz_idle_exit
  <idle>-0       3dN.1   15us : hrtimer_cancel <-tick_nohz_idle_exit
  <idle>-0       3dN.1   15us : hrtimer_try_to_cancel <-hrtimer_
↪cancel
  <idle>-0       3dN.1   16us : lock_hrtimer_base.isra.18 <-hrtimer_
↪try_to_cancel
  <idle>-0       3dN.1   16us : _raw_spin_lock_irqsave <-lock_
↪hrtimer_base.isra.18
  <idle>-0       3dN.1   16us : add_preempt_count <-_raw_spin_lock_
```

(continues on next page)

(continued from previous page)

```
↪irqsave
  <idle>-0       3dN.2   17us : __remove_hrtimer <-remove_hrtimer.
↪part.16
  <idle>-0       3dN.2   17us : hrtimer_force_reprogram <-__remove_
↪hrtimer
  <idle>-0       3dN.2   17us : tick_program_event <-hrtimer_force_
↪reprogram
  <idle>-0       3dN.2   18us : clockevents_program_event <-tick_
↪program_event
  <idle>-0       3dN.2   18us : ktime_get <-clockevents_program_
↪event
  <idle>-0       3dN.2   18us : lapic_next_event <-clockevents_
↪program_event
  <idle>-0       3dN.2   19us : _raw_spin_unlock_irqrestore <-
↪hrtimer_try_to_cancel
  <idle>-0       3dN.2   19us : sub_preempt_count <-_raw_spin_
↪unlock_irqrestore
  <idle>-0       3dN.1   19us : hrtimer_forward <-tick_nohz_idle_
↪exit
  <idle>-0       3dN.1   20us : ktime_add_safe <-hrtimer_forward
  <idle>-0       3dN.1   20us : ktime_add_safe <-hrtimer_forward
  <idle>-0       3dN.1   20us : hrtimer_start_range_ns <-hrtimer_
↪start_expires.constprop.11
  <idle>-0       3dN.1   20us : __hrtimer_start_range_ns <-hrtimer_
↪start_range_ns
  <idle>-0       3dN.1   21us : lock_hrtimer_base.isra.18 <-__
↪hrtimer_start_range_ns
  <idle>-0       3dN.1   21us : _raw_spin_lock_irqsave <-lock_
↪hrtimer_base.isra.18
  <idle>-0       3dN.1   21us : add_preempt_count <-_raw_spin_lock_
↪irqsave
  <idle>-0       3dN.2   22us : ktime_add_safe <-__hrtimer_start_
↪range_ns
  <idle>-0       3dN.2   22us : enqueue_hrtimer <-__hrtimer_start_
↪range_ns
  <idle>-0       3dN.2   22us : tick_program_event <-__hrtimer_
↪start_range_ns
  <idle>-0       3dN.2   23us : clockevents_program_event <-tick_
↪program_event
  <idle>-0       3dN.2   23us : ktime_get <-clockevents_program_
↪event
  <idle>-0       3dN.2   23us : lapic_next_event <-clockevents_
↪program_event
  <idle>-0       3dN.2   24us : _raw_spin_unlock_irqrestore <-__
↪hrtimer_start_range_ns
  <idle>-0       3dN.2   24us : sub_preempt_count <-_raw_spin_
↪unlock_irqrestore
  <idle>-0       3dN.1   24us : account_idle_ticks <-tick_nohz_idle_
↪exit
```

(continues on next page)

(continued from previous page)

```
  <idle>-0       3dN.1   24us : account_idle_time <-account_idle_
↪ticks
  <idle>-0       3.N.1   25us : sub_preempt_count <-cpu_idle
  <idle>-0       3.N..   25us : schedule <-cpu_idle
  <idle>-0       3.N..   25us : __schedule <-preempt_schedule
  <idle>-0       3.N..   26us : add_preempt_count <-__schedule
  <idle>-0       3.N.1   26us : rcu_note_context_switch <-__schedule
  <idle>-0       3.N.1   26us : rcu_sched_qs <-rcu_note_context_
↪switch
  <idle>-0       3dN.1   27us : rcu_preempt_qs <-rcu_note_context_
↪switch
  <idle>-0       3.N.1   27us : _raw_spin_lock_irq <-__schedule
  <idle>-0       3dN.1   27us : add_preempt_count <-_raw_spin_lock_
↪irq
  <idle>-0       3dN.2   28us : put_prev_task_idle <-__schedule
  <idle>-0       3dN.2   28us : pick_next_task_stop <-pick_next_task
  <idle>-0       3dN.2   28us : pick_next_task_rt <-pick_next_task
  <idle>-0       3dN.2   29us : dequeue_pushable_task <-pick_next_
↪task_rt
  <idle>-0       3d..3   29us : __schedule <-preempt_schedule
  <idle>-0       3d..3   30us :      0:120:R ==> [003]  2448: 94:R␣
↪sleep
```

This isn’t that big of a trace, even with function tracing enabled, so I included the entire trace.

The interrupt went off while when the system was idle. Somewhere before task_woken_rt() was called, the NEED_RESCHED flag was set, this is indicated by the first occurrence of the ‘N’ flag.

3.15 Latency tracing and events

As function tracing can induce a much larger latency, but without seeing what happens within the latency it is hard to know what caused it. There is a middle ground, and that is with enabling events.

```
# echo 0 > options/function-trace
# echo wakeup_rt > current_tracer
# echo 1 > events/enable
# echo 1 > tracing_on
# echo 0 > tracing_max_latency
# chrt -f 5 sleep 1
# echo 0 > tracing_on
# cat trace
# tracer: wakeup_rt
#
# wakeup_rt latency trace v1.1.5 on 3.8.0-test+
# -------------------------------------------------------------------
↪--
```

(continues on next page)

(continued from previous page)

```
# latency: 6 us, #12/12, CPU#2 | (M:preempt VP:0, KP:0, SP:0 HP:0
↪#P:4)
#    -----------------
#    | task: sleep-5882 (uid:0 nice:0 policy:1 rt_prio:5)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>-0       2d.h4    0us :      0:120:R   + [002]  5882: 94:R␣
↪sleep
  <idle>-0       2d.h4    0us : ttwu_do_activate.constprop.87 <-try_
↪to_wake_up
  <idle>-0       2d.h4    1us : sched_wakeup: comm=sleep pid=5882␣
↪prio=94 success=1 target_cpu=002
  <idle>-0       2dNh2    1us : hrtimer_expire_exit:␣
↪hrtimer=ffff88007796feb8
  <idle>-0       2.N.2    2us : power_end: cpu_id=2
  <idle>-0       2.N.2    3us : cpu_idle: state=4294967295 cpu_id=2
  <idle>-0       2dN.3    4us : hrtimer_cancel:␣
↪hrtimer=ffff88007d50d5e0
  <idle>-0       2dN.3    4us : hrtimer_start:␣
↪hrtimer=ffff88007d50d5e0 function=tick_sched_timer␣
↪expires=34311211000000 softexpires=34311211000000
  <idle>-0       2.N.2    5us : rcu_utilization: Start context␣
↪switch
  <idle>-0       2.N.2    5us : rcu_utilization: End context switch
  <idle>-0       2d..3    6us : __schedule <-schedule
  <idle>-0       2d..3    6us :      0:120:R ==> [002]  5882: 94:R␣
↪sleep
```

3.16 Hardware Latency Detector

The hardware latency detector is executed by enabling the "hwlat" tracer.

NOTE, this tracer will affect the performance of the system as it will periodically make a CPU constantly busy with interrupts disabled.

```
# echo hwlat > current_tracer
# sleep 100
# cat trace
# tracer: hwlat
#
```

(continues on next page)

(continued from previous page)

```
# entries-in-buffer/entries-written: 13/13   #P:8
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          <...>-1729  [001] d...   678.473449: #1     inner/
↪outer(us):   11/12    ts:1581527483.343962693 count:6
          <...>-1729  [004] d...   689.556542: #2     inner/
↪outer(us):   16/9     ts:1581527494.889008092 count:1
          <...>-1729  [005] d...   714.756290: #3     inner/
↪outer(us):   16/16    ts:1581527519.678961629 count:5
          <...>-1729  [001] d...   718.788247: #4     inner/
↪outer(us):    9/17    ts:1581527523.889012713 count:1
          <...>-1729  [002] d...   719.796341: #5     inner/
↪outer(us):   13/9     ts:1581527524.912872606 count:1
          <...>-1729  [006] d...   844.787091: #6     inner/
↪outer(us):    9/12    ts:1581527649.889048502 count:2
          <...>-1729  [003] d...   849.827033: #7     inner/
↪outer(us):   18/9     ts:1581527654.889013793 count:1
          <...>-1729  [007] d...   853.859002: #8     inner/
↪outer(us):    9/12    ts:1581527658.889065736 count:1
          <...>-1729  [001] d...   855.874978: #9     inner/
↪outer(us):    9/11    ts:1581527660.861991877 count:1
          <...>-1729  [001] d...   863.938932: #10    inner/
↪outer(us):    9/11    ts:1581527668.970010500 count:1 nmi-total:7␣
↪nmi-count:1
          <...>-1729  [007] d...   878.050780: #11    inner/
↪outer(us):    9/12    ts:1581527683.385002600 count:1 nmi-total:5␣
↪nmi-count:1
          <...>-1729  [007] d...   886.114702: #12    inner/
↪outer(us):    9/12    ts:1581527691.385001600 count:1
```

The above output is somewhat the same in the header. All events will have interrupts disabled 'd'. Under the FUNCTION title there is:

#1

This is the count of events recorded that were greater than the tracing_threshold (See below).

inner/outer(us): 11/11

This shows two numbers as "inner latency" and "outer latency". The test runs in a loop checking a timestamp twice. The latency detected within the two timestamps is the "inner latency" and the latency detected after the previous timestamp and the next timestamp in the loop is the "outer latency".

ts:1581527483.343962693

The absolute timestamp that the first latency was recorded in the window.

count:6

The number of times a latency was detected during the window.

nmi-total:7 nmi-count:1

On architectures that support it, if an NMI comes in during the test, the time spent in NMI is reported in "nmi-total" (in microseconds).

All architectures that have NMIs will show the "nmi-count" if an NMI comes in during the test.

hwlat files:

tracing_threshold
This gets automatically set to "10" to represent 10 microseconds. This is the threshold of latency that needs to be detected before the trace will be recorded.

Note, when hwlat tracer is finished (another tracer is written into "current_tracer"), the original value for tracing_threshold is placed back into this file.

hwlat_detector/width
The length of time the test runs with interrupts disabled.

hwlat_detector/window
The length of time of the window which the test runs. That is, the test will run for "width" microseconds per "window" microseconds

tracing_cpumask
When the test is started. A kernel thread is created that runs the test. This thread will alternate between CPUs listed in the tracing_cpumask between each period (one "window"). To limit the test to specific CPUs set the mask in this file to only the CPUs that the test should run on.

3.17 function

This tracer is the function tracer. Enabling the function tracer can be done from the debug file system. Make sure the ftrace_enabled is set; otherwise this tracer is a nop. See the "ftrace_enabled" section below.

```
# sysctl kernel.ftrace_enabled=1
# echo function > current_tracer
# echo 1 > tracing_on
# usleep 1
# echo 0 > tracing_on
# cat trace
# tracer: function
#
```

(continues on next page)

(continued from previous page)

```
# entries-in-buffer/entries-written: 24799/24799   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
            bash-1994  [002] ....  3082.063030: mutex_unlock <-rb_
↪simple_write
            bash-1994  [002] ....  3082.063031: __mutex_unlock_
↪slowpath <-mutex_unlock
            bash-1994  [002] ....  3082.063031: __fsnotify_parent <-
↪fsnotify_modify
            bash-1994  [002] ....  3082.063032: fsnotify <-fsnotify_
↪modify
            bash-1994  [002] ....  3082.063032: __srcu_read_lock <-
↪fsnotify
            bash-1994  [002] ....  3082.063032: add_preempt_count <-
↪__srcu_read_lock
            bash-1994  [002] ...1  3082.063032: sub_preempt_count <-
↪__srcu_read_lock
            bash-1994  [002] ....  3082.063033: __srcu_read_unlock
↪<-fsnotify
[...]
```

Note: function tracer uses ring buffers to store the above entries. The newest data may overwrite the oldest data. Sometimes using echo to stop the trace is not sufficient because the tracing could have overwritten the data that you wanted to record. For this reason, it is sometimes better to disable tracing directly from a program. This allows you to stop the tracing at the point that you hit the part that you are interested in. To disable the tracing directly from a C program, something like following code snippet can be used:

```
int trace_fd;
[...]
int main(int argc, char *argv[]) {
        [...]
        trace_fd = open(tracing_file("tracing_on"), O_WRONLY);
        [...]
        if (condition_hit()) {
                write(trace_fd, "0", 1);
        }
        [...]
}
```

3.18 Single thread tracing

By writing into set_ftrace_pid you can trace a single thread. For example:

```
# cat set_ftrace_pid
no pid
# echo 3111 > set_ftrace_pid
# cat set_ftrace_pid
3111
# echo function > current_tracer
# cat trace | head
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
    yum-updatesd-3111  [003]  1637.254676: finish_task_switch <-
↪thread_return
    yum-updatesd-3111  [003]  1637.254681: hrtimer_cancel <-
↪schedule_hrtimeout_range
    yum-updatesd-3111  [003]  1637.254682: hrtimer_try_to_cancel <-
↪hrtimer_cancel
    yum-updatesd-3111  [003]  1637.254683: lock_hrtimer_base <-
↪hrtimer_try_to_cancel
    yum-updatesd-3111  [003]  1637.254685: fget_light <-do_sys_poll
    yum-updatesd-3111  [003]  1637.254686: pipe_poll <-do_sys_poll
# echo > set_ftrace_pid
# cat trace |head
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
##### CPU 3 buffer started ####
    yum-updatesd-3111  [003]  1701.957688: free_poll_entry <-poll_
↪freewait
    yum-updatesd-3111  [003]  1701.957689: remove_wait_queue <-free_
↪poll_entry
    yum-updatesd-3111  [003]  1701.957691: fput <-free_poll_entry
    yum-updatesd-3111  [003]  1701.957692: audit_syscall_exit <-
↪sysret_audit
    yum-updatesd-3111  [003]  1701.957693: path_put <-audit_syscall_
↪exit
```

If you want to trace a function when executing, you could use something like this simple program.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

(continues on next page)

(continued from previous page)

```
#include <unistd.h>
#include <string.h>

#define _STR(x) #x
#define STR(x) _STR(x)
#define MAX_PATH 256

const char *find_tracefs(void)
{
       static char tracefs[MAX_PATH+1];
       static int tracefs_found;
       char type[100];
       FILE *fp;

       if (tracefs_found)
               return tracefs;

       if ((fp = fopen("/proc/mounts","r")) == NULL) {
               perror("/proc/mounts");
               return NULL;
       }

       while (fscanf(fp, "%*s %"
                     STR(MAX_PATH)
                     "s %99s %*s %*d %*d\n",
                     tracefs, type) == 2) {
               if (strcmp(type, "tracefs") == 0)
                       break;
       }
       fclose(fp);

       if (strcmp(type, "tracefs") != 0) {
               fprintf(stderr, "tracefs not mounted");
               return NULL;
       }

       strcat(tracefs, "/tracing/");
       tracefs_found = 1;

       return tracefs;
}

const char *tracing_file(const char *file_name)
{
       static char trace_file[MAX_PATH+1];
       snprintf(trace_file, MAX_PATH, "%s/%s", find_tracefs(), file_
↪name);
       return trace_file;
}
```

(continues on next page)

(continued from previous page)

```
int main (int argc, char **argv)
{
        if (argc < 1)
                exit(-1);

        if (fork() > 0) {
                int fd, ffd;
                char line[64];
                int s;

                ffd = open(tracing_file("current_tracer"), O_
↪WRONLY);
                if (ffd < 0)
                        exit(-1);
                write(ffd, "nop", 3);

                fd = open(tracing_file("set_ftrace_pid"), O_WRONLY);
                s = sprintf(line, "%d\n", getpid());
                write(fd, line, s);

                write(ffd, "function", 8);

                close(fd);
                close(ffd);

                execvp(argv[1], argv+1);
        }

        return 0;
}
```

Or this simple script!

```
#!/bin/bash

tracefs=`sed -ne 's/^tracefs \(.*\) tracefs.*/\1/p' /proc/mounts`
echo nop > $tracefs/tracing/current_tracer
echo 0 > $tracefs/tracing/tracing_on
echo $$ > $tracefs/tracing/set_ftrace_pid
echo function > $tracefs/tracing/current_tracer
echo 1 > $tracefs/tracing/tracing_on
exec "$@"
```

3.19 function graph tracer

This tracer is similar to the function tracer except that it probes a function on its entry and its exit. This is done by using a dynamically allocated stack of return addresses in each task_struct. On function entry the tracer overwrites the return address of each function traced to set a custom probe. Thus the original return address is stored on the stack of return address in the task_struct.

Probing on both ends of a function leads to special features such as:

- measure of a function’s time execution
- having a reliable call stack to draw function calls graph

This tracer is useful in several situations:

- you want to find the reason of a strange kernel behavior and need to see what happens in detail on any areas (or specific ones).
- you are experiencing weird latencies but it’s difficult to find its origin.
- you want to find quickly which path is taken by a specific function
- you just want to peek inside a working kernel and want to see what happens there.

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |

 0)               |  sys_open() {
 0)               |    do_sys_open() {
 0)               |      getname() {
 0)               |        kmem_cache_alloc() {
 0)   1.382 us    |          __might_sleep();
 0)   2.478 us    |        }
 0)               |        strncpy_from_user() {
 0)               |          might_fault() {
 0)   1.389 us    |            __might_sleep();
 0)   2.553 us    |          }
 0)   3.807 us    |        }
 0)   7.876 us    |      }
 0)               |      alloc_fd() {
 0)   0.668 us    |        _spin_lock();
 0)   0.570 us    |        expand_files();
 0)   0.586 us    |        _spin_unlock();
```

There are several columns that can be dynamically enabled/disabled. You can use every combination of options you want, depending on your needs.

- The cpu number on which the function executed is default enabled. It is sometimes better to only trace one cpu (see tracing_cpu_mask file) or you might sometimes see unordered function calls while cpu tracing switch.
  - hide: echo nofuncgraph-cpu > trace_options

   - show: echo funcgraph-cpu > trace_options

- The duration (function's time of execution) is displayed on the closing bracket line of a function or on the same line than the current function in case of a leaf one. It is default enabled.
   - hide: echo nofuncgraph-duration > trace_options
   - show: echo funcgraph-duration > trace_options
- The overhead field precedes the duration field in case of reached duration thresholds.
   - hide: echo nofuncgraph-overhead > trace_options
   - show: echo funcgraph-overhead > trace_options
   - depends on: funcgraph-duration

  ie:

```
3) # 1837.709 us |          } /* __switch_to */
3)               |          finish_task_switch() {
3)   0.313 us    |            _raw_spin_unlock_irq();
3)   3.177 us    |          }
3) # 1889.063 us |        } /* __schedule */
3) ! 140.417 us  |      } /* __schedule */
3) # 2034.948 us |    } /* schedule */
3) * 33998.59 us |  } /* schedule_preempt_disabled */

[...]

1)   0.260 us    |              msecs_to_jiffies();
1)   0.313 us    |              __rcu_read_unlock();
1) + 61.770 us   |            }
1) + 64.479 us   |          }
1)   0.313 us    |          rcu_bh_qs();
1)   0.313 us    |          __local_bh_enable();
1) ! 217.240 us  |        }
1)   0.365 us    |        idle_cpu();
1)               |        rcu_irq_exit() {
1)   0.417 us    |          rcu_eqs_enter_common.isra.47();
1)   3.125 us    |        }
1) ! 227.812 us  |      }
1) ! 457.395 us  |    }
1) @ 119760.2 us |  }

[...]

2)               |    handle_IPI() {
1)   6.979 us    |                  }
2)   0.417 us    |      scheduler_ipi();
1)   9.791 us    |                }
1) + 12.917 us   |              }
2)   3.490 us    |    }
```

(continues on next page)

(continued from previous page)

```
1) + 15.729 us   |            }
1) + 18.542 us   |          }
2) $ 3594274 us  |  }
```

Flags:

```
+ means that the function exceeded 10 usecs.
! means that the function exceeded 100 usecs.
# means that the function exceeded 1000 usecs.
* means that the function exceeded 10 msecs.
@ means that the function exceeded 100 msecs.
$ means that the function exceeded 1 sec.
```

- The task/pid field displays the thread cmdline and pid which executed the function. It is default disabled.
  - hide: echo nofuncgraph-proc > trace_options
  - show: echo funcgraph-proc > trace_options

  ie:

```
# tracer: function_graph
#
# CPU  TASK/PID        DURATION                  FUNCTION CALLS
# |    |    |           |   |                     |   |   |   |
0)    sh-4802     |               |                  d_free() {
0)    sh-4802     |               |                    call_
↪rcu() {
0)    sh-4802     |               |                        __call_
↪rcu() {
0)    sh-4802     |   0.616 us    |                          rcu_
↪process_gp_end();
0)    sh-4802     |   0.586 us    |                          ␣
↪check_for_new_grace_period();
0)    sh-4802     |   2.899 us    |                        }
0)    sh-4802     |   4.040 us    |                      }
0)    sh-4802     |   5.151 us    |                    }
0)    sh-4802     | + 49.370 us   |                  }
```

- The absolute time field is an absolute timestamp given by the system clock since it started. A snapshot of this time is given on each entry/exit of functions
  - hide: echo nofuncgraph-abstime > trace_options
  - show: echo funcgraph-abstime > trace_options

  ie:

```
#
#      TIME       CPU  DURATION                  FUNCTION CALLS
#       |         |     |   |                     |   |   |   |
360.774522 |   1)   0.541 us    |                                        ␣
```

(continues on next page)

(continued from previous page)

```
↪           }
360.774522 |   1)   4.663 us    |                                                     ␣
↪         }
360.774523 |   1)   0.541 us    |                                                     ␣
↪         __wake_up_bit();
360.774524 |   1)   6.796 us    |                                                     ␣
↪       }
360.774524 |   1)   7.952 us    |                                                     ␣
↪     }
360.774525 |   1)   9.063 us    |                                                     ␣
↪   }
360.774525 |   1)   0.615 us    |                                                     ␣
↪   journal_mark_dirty();
360.774527 |   1)   0.578 us    |                                                     ␣
↪   __brelse();
360.774528 |   1)               |                                                     ␣
↪   reiserfs_prepare_for_journal() {
360.774528 |   1)               |                                                     ␣
↪     unlock_buffer() {
360.774529 |   1)               |                                                     ␣
↪       wake_up_bit() {
360.774529 |   1)               |                                                     ␣
↪         bit_waitqueue() {
360.774530 |   1)   0.594 us    |                                                     ␣
↪           __phys_addr();
```

The function name is always displayed after the closing bracket for a function if the start of that function is not in the trace buffer.

Display of the function name after the closing bracket may be enabled for functions whose start is in the trace buffer, allowing easier searching with grep for function durations. It is default disabled.

- hide: echo nofuncgraph-tail > trace_options
- show: echo funcgraph-tail > trace_options

Example with nofuncgraph-tail (default):

```
0)                |      putname() {
0)                |        kmem_cache_free() {
0)   0.518 us     |          __phys_addr();
0)   1.757 us     |        }
0)   2.861 us     |      }
```

Example with funcgraph-tail:

```
0)                |      putname() {
0)                |        kmem_cache_free() {
0)   0.518 us     |          __phys_addr();
0)   1.757 us     |        } /* kmem_cache_free() */
0)   2.861 us     |      } /* putname() */
```

You can put some comments on specific functions by using trace_printk() For example, if you want to put a comment inside the __might_sleep() function, you just have to include <linux/ftrace.h> and call trace_printk() inside __might_sleep():

```
trace_printk("I'm a comment!\n")
```

will produce:

```
1)                 |             __might_sleep() {
1)                 |                /* I'm a comment! */
1)   1.449 us      |             }
```

You might find other useful features for this tracer in the following "dynamic ftrace" section such as tracing only specific functions or tasks.

3.20 dynamic ftrace

If CONFIG_DYNAMIC_FTRACE is set, the system will run with virtually no overhead when function tracing is disabled. The way this works is the mcount function call (placed at the start of every kernel function, produced by the -pg switch in gcc), starts of pointing to a simple return. (Enabling FTRACE will include the -pg switch in the compiling of the kernel.)

At compile time every C file object is run through the recordmcount program (located in the scripts directory). This program will parse the ELF headers in the C object to find all the locations in the .text section that call mcount. Starting with gcc version 4.6, the -mfentry has been added for x86, which calls "__fentry__" instead of "mcount". Which is called before the creation of the stack frame.

Note, not all sections are traced. They may be prevented by either a notrace, or blocked another way and all inline functions are not traced. Check the "available_filter_functions" file to see what functions can be traced.

A section called "__mcount_loc" is created that holds references to all the mcount/fentry call sites in the .text section. The recordmcount program re-links this section back into the original object. The final linking stage of the kernel will add all these references into a single table.

On boot up, before SMP is initialized, the dynamic ftrace code scans this table and updates all the locations into nops. It also records the locations, which are added to the available_filter_functions list. Modules are processed as they are loaded and before they are executed. When a module is unloaded, it also removes its functions from the ftrace function list. This is automatic in the module unload code, and the module author does not need to worry about it.

When tracing is enabled, the process of modifying the function tracepoints is dependent on architecture. The old method is to use kstop_machine to prevent races with the CPUs executing code being modified (which can cause the CPU to do undesirable things, especially if the modified code crosses cache (or page) boundaries), and the nops are patched back to calls. But this time, they do not call mcount (which is just a function stub). They now call into the ftrace infrastructure.

The new method of modifying the function tracepoints is to place a breakpoint at the location to be modified, sync all CPUs, modify the rest of the instruction not covered by the breakpoint. Sync all CPUs again, and then remove the breakpoint with the finished version to the ftrace call site.

Some archs do not even need to monkey around with the synchronization, and can just slap the new code on top of the old without any problems with other CPUs executing it at the same time.

One special side-effect to the recording of the functions being traced is that we can now selectively choose which functions we wish to trace and which ones we want the mcount calls to remain as nops.

Two files are used, one for enabling and one for disabling the tracing of specified functions. They are:

set_ftrace_filter

and

set_ftrace_notrace

A list of available functions that you can add to these files is listed in:

available_filter_functions

```
# cat available_filter_functions
put_prev_task_idle
kmem_cache_create
pick_next_task_rt
get_online_cpus
pick_next_task_fair
mutex_lock
[...]
```

If I am only interested in sys_nanosleep and hrtimer_interrupt:

```
# echo sys_nanosleep hrtimer_interrupt > set_ftrace_filter
# echo function > current_tracer
# echo 1 > tracing_on
# usleep 1
# echo 0 > tracing_on
# cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 5/5   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          usleep-2665  [001] ....  4186.475355: sys_nanosleep <-
→system_call_fastpath
```

(continues on next page)

(continued from previous page)

```
          <idle>-0     [001] d.h1  4186.475409: hrtimer_interrupt <-
↪smp_apic_timer_interrupt
          usleep-2665  [001] d.h1  4186.475426: hrtimer_interrupt <-
↪smp_apic_timer_interrupt
          <idle>-0     [003] d.h1  4186.475426: hrtimer_interrupt <-
↪smp_apic_timer_interrupt
          <idle>-0     [002] d.h1  4186.475427: hrtimer_interrupt <-
↪smp_apic_timer_interrupt
```

To see which functions are being traced, you can cat the file:

```
# cat set_ftrace_filter
hrtimer_interrupt
sys_nanosleep
```

Perhaps this is not enough. The filters also allow glob(7) matching.

<match>*
will match functions that begin with <match>

***<match>**
will match functions that end with <match>

<match>
will match functions that have <match> in it

<match1>*<match2>
will match functions that begin with <match1> and end with <match2>

Note: It is better to use quotes to enclose the wild cards, otherwise the shell may expand the parameters into names of files in the local directory.

```
# echo 'hrtimer_*' > set_ftrace_filter
```

Produces:

```
# tracer: function
#
# entries-in-buffer/entries-written: 897/897   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          <idle>-0     [003] dN.1  4228.547803: hrtimer_cancel <-
↪tick_nohz_idle_exit
```

(continues on next page)

(continued from previous page)

```
          <idle>-0     [003] dN.1  4228.547804: hrtimer_try_to_
↪cancel <-hrtimer_cancel
          <idle>-0     [003] dN.2  4228.547805: hrtimer_force_
↪reprogram <-__remove_hrtimer
          <idle>-0     [003] dN.1  4228.547805: hrtimer_forward <-
↪tick_nohz_idle_exit
          <idle>-0     [003] dN.1  4228.547805: hrtimer_start_range_
↪ns <-hrtimer_start_expires.constprop.11
          <idle>-0     [003] d..1  4228.547858: hrtimer_get_next_
↪event <-get_next_timer_interrupt
          <idle>-0     [003] d..1  4228.547859: hrtimer_start <-__
↪tick_nohz_idle_enter
          <idle>-0     [003] d..2  4228.547860: hrtimer_force_
↪reprogram <-__rem
```

Notice that we lost the sys_nanosleep.

```
# cat set_ftrace_filter
hrtimer_run_queues
hrtimer_run_pending
hrtimer_init
hrtimer_cancel
hrtimer_try_to_cancel
hrtimer_forward
hrtimer_start
hrtimer_reprogram
hrtimer_force_reprogram
hrtimer_get_next_event
hrtimer_interrupt
hrtimer_nanosleep
hrtimer_wakeup
hrtimer_get_remaining
hrtimer_get_res
hrtimer_init_sleeper
```

This is because the '>' and '>>' act just like they do in bash. To rewrite the filters, use '>' To append to the filters, use '>>'

To clear out a filter so that all functions will be recorded again:

```
# echo > set_ftrace_filter
# cat set_ftrace_filter
#
```

Again, now we want to append.

```
# echo sys_nanosleep > set_ftrace_filter
# cat set_ftrace_filter
sys_nanosleep
# echo 'hrtimer_*' >> set_ftrace_filter
```

(continues on next page)

(continued from previous page)

```
# cat set_ftrace_filter
hrtimer_run_queues
hrtimer_run_pending
hrtimer_init
hrtimer_cancel
hrtimer_try_to_cancel
hrtimer_forward
hrtimer_start
hrtimer_reprogram
hrtimer_force_reprogram
hrtimer_get_next_event
hrtimer_interrupt
sys_nanosleep
hrtimer_nanosleep
hrtimer_wakeup
hrtimer_get_remaining
hrtimer_get_res
hrtimer_init_sleeper
```

The set_ftrace_notrace prevents those functions from being traced.

```
# echo '*preempt*' '*lock*' > set_ftrace_notrace
```

Produces:

```
# tracer: function
#
# entries-in-buffer/entries-written: 39608/39608   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
            bash-1994  [000] ....  4342.324896: file_ra_state_init
↪<-do_dentry_open
            bash-1994  [000] ....  4342.324897: open_check_o_direct
↪<-do_last
            bash-1994  [000] ....  4342.324897: ima_file_check <-do_
↪last
            bash-1994  [000] ....  4342.324898: process_measurement
↪<-ima_file_check
            bash-1994  [000] ....  4342.324898: ima_get_action <-
↪process_measurement
            bash-1994  [000] ....  4342.324898: ima_match_policy <-
↪ima_get_action
            bash-1994  [000] ....  4342.324899: do_truncate <-do_
↪last
```

(continues on next page)

(continued from previous page)

```
           bash-1994  [000] ....  4342.324899: setattr_should_drop_
↪suidgid <-do_truncate
           bash-1994  [000] ....  4342.324899: notify_change <-do_
↪truncate
           bash-1994  [000] ....  4342.324900: current_fs_time <-
↪notify_change
           bash-1994  [000] ....  4342.324900: current_kernel_time
↪<-current_fs_time
           bash-1994  [000] ....  4342.324900: timespec_trunc <-
↪current_fs_time
```

We can see that there's no more lock or preempt tracing.

3.21 Selecting function filters via index

Because processing of strings is expensive (the address of the function needs to be looked up before comparing to the string being passed in), an index can be used as well to enable functions. This is useful in the case of setting thousands of specific functions at a time. By passing in a list of numbers, no string processing will occur. Instead, the function at the specific location in the internal array (which corresponds to the functions in the "available_filter_functions" file), is selected.

```
# echo 1 > set_ftrace_filter
```

Will select the first function listed in "available_filter_functions"

```
# head -1 available_filter_functions
trace_initcall_finish_cb

# cat set_ftrace_filter
trace_initcall_finish_cb

# head -50 available_filter_functions | tail -1
x86_pmu_commit_txn

# echo 1 50 > set_ftrace_filter
# cat set_ftrace_filter
trace_initcall_finish_cb
x86_pmu_commit_txn
```

3.22 Dynamic ftrace with the function graph tracer

Although what has been explained above concerns both the function tracer and the function-graph-tracer, there are some special features only available in the function-graph tracer.

If you want to trace only one function and all of its children, you just have to echo its name into set_graph_function:

```
echo __do_fault > set_graph_function
```

will produce the following "expanded" trace of the __do_fault() function:

```
0)               |  __do_fault() {
0)               |    filemap_fault() {
0)               |      find_lock_page() {
0)   0.804 us    |        find_get_page();
0)               |        __might_sleep() {
0)   1.329 us    |        }
0)   3.904 us    |      }
0)   4.979 us    |    }
0)   0.653 us    |    _spin_lock();
0)   0.578 us    |    page_add_file_rmap();
0)   0.525 us    |    native_set_pte_at();
0)   0.585 us    |    _spin_unlock();
0)               |    unlock_page() {
0)   0.541 us    |      page_waitqueue();
0)   0.639 us    |      __wake_up_bit();
0)   2.786 us    |    }
0) + 14.237 us   |  }
0)               |  __do_fault() {
0)               |    filemap_fault() {
0)               |      find_lock_page() {
0)   0.698 us    |        find_get_page();
0)               |        __might_sleep() {
0)   1.412 us    |        }
0)   3.950 us    |      }
0)   5.098 us    |    }
0)   0.631 us    |    _spin_lock();
0)   0.571 us    |    page_add_file_rmap();
0)   0.526 us    |    native_set_pte_at();
0)   0.586 us    |    _spin_unlock();
0)               |    unlock_page() {
0)   0.533 us    |      page_waitqueue();
0)   0.638 us    |      __wake_up_bit();
0)   2.793 us    |    }
0) + 14.012 us   |  }
```

You can also expand several functions at once:

```
echo sys_open > set_graph_function
echo sys_close >> set_graph_function
```

Now if you want to go back to trace all functions you can clear this special filter via:

```
echo > set_graph_function
```

3.23 ftrace_enabled

Note, the proc sysctl ftrace_enable is a big on/off switch for the function tracer. By default it is enabled (when function tracing is enabled in the kernel). If it is disabled, all function tracing is disabled. This includes not only the function tracers for ftrace, but also for any other uses (perf, kprobes, stack tracing, profiling, etc). It cannot be disabled if there is a callback with FTRACE_OPS_FL_PERMANENT set registered.

Please disable this with care.

This can be disable (and enabled) with:

```
 sysctl kernel.ftrace_enabled=0
 sysctl kernel.ftrace_enabled=1

or

 echo 0 > /proc/sys/kernel/ftrace_enabled
 echo 1 > /proc/sys/kernel/ftrace_enabled
```

3.24 Filter commands

A few commands are supported by the set_ftrace_filter interface. Trace commands have the following format:

```
<function>:<command>:<parameter>
```

The following commands are supported:

- mod: This command enables function filtering per module. The parameter defines the module. For example, if only the write* functions in the ext3 module are desired, run:

  echo 'write*:mod:ext3' > set_ftrace_filter

  This command interacts with the filter in the same way as filtering based on function names. Thus, adding more functions in a different module is accomplished by appending (>>) to the filter file. Remove specific module functions by prepending '!' :

  ```
  echo '!writeback*:mod:ext3' >> set_ftrace_filter
  ```

  Mod command supports module globbing. Disable tracing for all functions except a specific module:

```
echo '!*:mod:!ext3' >> set_ftrace_filter
```

Disable tracing for all modules, but still trace kernel:

```
echo '!*:mod:*' >> set_ftrace_filter
```

Enable filter only for kernel:

```
echo '*write*:mod:!*' >> set_ftrace_filter
```

Enable filter for module globbing:

```
echo '*write*:mod:*snd*' >> set_ftrace_filter
```

- traceon/traceoff: These commands turn tracing on and off when the specified functions are hit. The parameter determines how many times the tracing system is turned on and off. If unspecified, there is no limit. For example, to disable tracing when a schedule bug is hit the first 5 times, run:

  ```
  echo '__schedule_bug:traceoff:5' > set_ftrace_filter
  ```

  To always disable tracing when __schedule_bug is hit:

  ```
  echo '__schedule_bug:traceoff' > set_ftrace_filter
  ```

  These commands are cumulative whether or not they are appended to set_ftrace_filter. To remove a command, prepend it by '!' and drop the parameter:

  ```
  echo '!__schedule_bug:traceoff:0' > set_ftrace_filter
  ```

  The above removes the traceoff command for __schedule_bug that have a counter. To remove commands without counters:

  ```
  echo '!__schedule_bug:traceoff' > set_ftrace_filter
  ```

- snapshot: Will cause a snapshot to be triggered when the function is hit.

  ```
  echo 'native_flush_tlb_others:snapshot' > set_ftrace_filter
  ```

  To only snapshot once:

  ```
  echo 'native_flush_tlb_others:snapshot:1' > set_ftrace_filter
  ```

  To remove the above commands:

  ```
  echo '!native_flush_tlb_others:snapshot' > set_ftrace_filter
  echo '!native_flush_tlb_others:snapshot:0' > set_ftrace_filter
  ```

- enable_event/disable_event: These commands can enable or disable a trace event. Note, because function tracing callbacks are very sensitive, when these commands are registered, the trace point is activated, but disabled in a "soft" mode. That is, the tracepoint will be called, but just will not be traced.

The event tracepoint stays in this mode as long as there's a command that triggers it.

```
echo 'try_to_wake_up:enable_event:sched:sched_switch:2' > \
      set_ftrace_filter
```

The format is:

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

To remove the events commands:

```
echo '!try_to_wake_up:enable_event:sched:sched_switch:0' > \
      set_ftrace_filter
echo '!schedule:disable_event:sched:sched_switch' > \
      set_ftrace_filter
```

- dump: When the function is hit, it will dump the contents of the ftrace ring buffer to the console. This is useful if you need to debug something, and want to dump the trace when a certain function is hit. Perhaps it's a function that is called before a triple fault happens and does not allow you to get a regular dump.
- cpudump: When the function is hit, it will dump the contents of the ftrace ring buffer for the current CPU to the console. Unlike the "dump" command, it only prints out the contents of the ring buffer for the CPU that executed the function that triggered the dump.
- stacktrace: When the function is hit, a stack trace is recorded.

3.25 trace_pipe

The trace_pipe outputs the same content as the trace file, but the effect on the tracing is different. Every read from trace_pipe is consumed. This means that subsequent reads will be different. The trace is live.

```
# echo function > current_tracer
# cat trace_pipe > /tmp/trace.out &
[1] 4153
# echo 1 > tracing_on
# usleep 1
# echo 0 > tracing_on
# cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 0/0   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
```

(continues on next page)

(continued from previous page)

```
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |

#
# cat /tmp/trace.out
           bash-1994  [000] ....  5281.568961: mutex_unlock <-rb_
↪simple_write
           bash-1994  [000] ....  5281.568963: __mutex_unlock_
↪slowpath <-mutex_unlock
           bash-1994  [000] ....  5281.568963: __fsnotify_parent <-
↪fsnotify_modify
           bash-1994  [000] ....  5281.568964: fsnotify <-fsnotify_
↪modify
           bash-1994  [000] ....  5281.568964: __srcu_read_lock <-
↪fsnotify
           bash-1994  [000] ....  5281.568964: add_preempt_count <-_
↪_srcu_read_lock
           bash-1994  [000] ...1  5281.568965: sub_preempt_count <-_
↪_srcu_read_lock
           bash-1994  [000] ....  5281.568965: __srcu_read_unlock <-
↪fsnotify
           bash-1994  [000] ....  5281.568967: sys_dup2 <-system_
↪call_fastpath
```

Note, reading the trace_pipe file will block until more input is added. This is contrary to the trace file. If any process opened the trace file for reading, it will actually disable tracing and prevent new entries from being added. The trace_pipe file does not have this limitation.

3.26 trace entries

Having too much or not enough data can be troublesome in diagnosing an issue in the kernel. The file buffer_size_kb is used to modify the size of the internal trace buffers. The number listed is the number of entries that can be recorded per CPU. To know the full size, multiply the number of possible CPUs with the number of entries.

```
# cat buffer_size_kb
1408 (units kilobytes)
```

Or simply read buffer_total_size_kb

```
# cat buffer_total_size_kb
5632
```

To modify the buffer, simple echo in a number (in 1024 byte segments).

```
# echo 10000 > buffer_size_kb
# cat buffer_size_kb
10000 (units kilobytes)
```

It will try to allocate as much as possible. If you allocate too much, it can cause Out-Of-Memory to trigger.

```
# echo 1000000000000 > buffer_size_kb
-bash: echo: write error: Cannot allocate memory
# cat buffer_size_kb
85
```

The per_cpu buffers can be changed individually as well:

```
# echo 10000 > per_cpu/cpu0/buffer_size_kb
# echo 100 > per_cpu/cpu1/buffer_size_kb
```

When the per_cpu buffers are not the same, the buffer_size_kb at the top level will just show an X

```
# cat buffer_size_kb
X
```

This is where the buffer_total_size_kb is useful:

```
# cat buffer_total_size_kb
12916
```

Writing to the top level buffer_size_kb will reset all the buffers to be the same again.

3.27 Snapshot

CONFIG_TRACER_SNAPSHOT makes a generic snapshot feature available to all non latency tracers. (Latency tracers which record max latency, such as "irqsoff" or "wakeup", can't use this feature, since those are already using the snapshot mechanism internally.)

Snapshot preserves a current trace buffer at a particular point in time without stopping tracing. Ftrace swaps the current buffer with a spare buffer, and tracing continues in the new current (=previous spare) buffer.

The following tracefs files in "tracing" are related to this feature:

snapshot:

> This is used to take a snapshot and to read the output of the snapshot. Echo 1 into this file to allocate a spare buffer and to take a snapshot (swap), then read the snapshot from this file in the same format as "trace" (described above in the section "The File System"). Both reads snapshot and tracing are executable in parallel. When the spare buffer is allocated, echoing

0 frees it, and echoing else (positive) values clear the snapshot contents. More details are shown in the table below.

status\input	0	1	else
not allocated	(do nothing)	alloc+swap	(do nothing)
allocated	free	swap	clear

Here is an example of using the snapshot feature.

```
# echo 1 > events/sched/enable
# echo 1 > snapshot
# cat snapshot
# tracer: nop
#
# entries-in-buffer/entries-written: 71/71   #P:8
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          <idle>-0     [005] d...  2440.603828: sched_switch: prev_
↪comm=swapper/5 prev_pid=0 prev_prio=120   prev_state=R ==> next_
↪comm=snapshot-test-2 next_pid=2242 next_prio=120
           sleep-2242  [005] d...  2440.603846: sched_switch: prev_
↪comm=snapshot-test-2 prev_pid=2242 prev_prio=120   prev_state=R␣
↪==> next_comm=kworker/5:1 next_pid=60 next_prio=120
[...]
        <idle>-0     [002] d...  2440.707230: sched_switch: prev_
↪comm=swapper/2 prev_pid=0 prev_prio=120 prev_state=R ==> next_
↪comm=snapshot-test-2 next_pid=2229 next_prio=120

# cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 77/77   #P:8
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          <idle>-0     [007] d...  2440.707395: sched_switch: prev_
↪comm=swapper/7 prev_pid=0 prev_prio=120 prev_state=R ==> next_
↪comm=snapshot-test-2 next_pid=2243 next_prio=120
```

(continues on next page)

(continued from previous page)

```
 snapshot-test-2-2229  [002] d...  2440.707438: sched_switch: prev_
↪comm=snapshot-test-2 prev_pid=2229 prev_prio=120 prev_state=S ==>␣
↪next_comm=swapper/2 next_pid=0 next_prio=120
[...]
```

If you try to use this snapshot feature when current tracer is one of the latency tracers, you will get the following results.

```
# echo wakeup > current_tracer
# echo 1 > snapshot
bash: echo: write error: Device or resource busy
# cat snapshot
cat: snapshot: Device or resource busy
```

3.28 Instances

In the tracefs tracing directory is a directory called "instances". This directory can have new directories created inside of it using mkdir, and removing directories with rmdir. The directory created with mkdir in this directory will already contain files and other directories after it is created.

```
# mkdir instances/foo
# ls instances/foo
buffer_size_kb  buffer_total_size_kb  events  free_buffer  per_cpu
set_event  snapshot  trace  trace_clock  trace_marker  trace_options
trace_pipe  tracing_on
```

As you can see, the new directory looks similar to the tracing directory itself. In fact, it is very similar, except that the buffer and events are agnostic from the main directory, or from any other instances that are created.

The files in the new directory work just like the files with the same name in the tracing directory except the buffer that is used is a separate and new buffer. The files affect that buffer but do not affect the main buffer with the exception of trace_options. Currently, the trace_options affect all instances and the top level buffer the same, but this may change in future releases. That is, options may become specific to the instance they reside in.

Notice that none of the function tracer files are there, nor is current_tracer and available_tracers. This is because the buffers can currently only have events enabled for them.

```
# mkdir instances/foo
# mkdir instances/bar
# mkdir instances/zoot
# echo 100000 > buffer_size_kb
# echo 1000 > instances/foo/buffer_size_kb
# echo 5000 > instances/bar/per_cpu/cpu1/buffer_size_kb
# echo function > current_trace
```

(continues on next page)

(continued from previous page)

```
# echo 1 > instances/foo/events/sched/sched_wakeup/enable
# echo 1 > instances/foo/events/sched/sched_wakeup_new/enable
# echo 1 > instances/foo/events/sched/sched_switch/enable
# echo 1 > instances/bar/events/irq/enable
# echo 1 > instances/zoot/events/syscalls/enable
# cat trace_pipe
CPU:2 [LOST 11745 EVENTS]
            bash-2044  [002] .... 10594.481032: _raw_spin_lock_
↪irqsave <-get_page_from_freelist
            bash-2044  [002] d... 10594.481032: add_preempt_count <-
↪_raw_spin_lock_irqsave
            bash-2044  [002] d..1 10594.481032: __rmqueue <-get_
↪page_from_freelist
            bash-2044  [002] d..1 10594.481033: _raw_spin_unlock <-
↪get_page_from_freelist
            bash-2044  [002] d..1 10594.481033: sub_preempt_count <-
↪_raw_spin_unlock
            bash-2044  [002] d... 10594.481033: get_pageblock_flags_
↪group <-get_pageblock_migratetype
            bash-2044  [002] d... 10594.481034: __mod_zone_page_
↪state <-get_page_from_freelist
            bash-2044  [002] d... 10594.481034: zone_statistics <-
↪get_page_from_freelist
            bash-2044  [002] d... 10594.481034: __inc_zone_state <-
↪zone_statistics
            bash-2044  [002] d... 10594.481034: __inc_zone_state <-
↪zone_statistics
            bash-2044  [002] .... 10594.481035: arch_dup_task_
↪struct <-copy_process
[...]

# cat instances/foo/trace_pipe
            bash-1998  [000] d..4   136.676759: sched_wakeup:␣
↪comm=kworker/0:1 pid=59 prio=120 success=1 target_cpu=000
            bash-1998  [000] dN.4   136.676760: sched_wakeup:␣
↪comm=bash pid=1998 prio=120 success=1 target_cpu=000
          <idle>-0     [003] d.h3   136.676906: sched_wakeup:␣
↪comm=rcu_preempt pid=9 prio=120 success=1 target_cpu=003
          <idle>-0     [003] d..3   136.676909: sched_switch: prev_
↪comm=swapper/3 prev_pid=0 prev_prio=120 prev_state=R ==> next_
↪comm=rcu_preempt next_pid=9 next_prio=120
     rcu_preempt-9     [003] d..3   136.676916: sched_switch: prev_
↪comm=rcu_preempt prev_pid=9 prev_prio=120 prev_state=S ==> next_
↪comm=swapper/3 next_pid=0 next_prio=120
            bash-1998  [000] d..4   136.677014: sched_wakeup:␣
↪comm=kworker/0:1 pid=59 prio=120 success=1 target_cpu=000
            bash-1998  [000] dN.4   136.677016: sched_wakeup:␣
↪comm=bash pid=1998 prio=120 success=1 target_cpu=000
            bash-1998  [000] d..3   136.677018: sched_switch: prev_
```

(continues on next page)

(continued from previous page)

```
↪comm=bash prev_pid=1998 prev_prio=120 prev_state=R+ ==> next_
↪comm=kworker/0:1 next_pid=59 next_prio=120
     kworker/0:1-59    [000] d..4   136.677022: sched_wakeup:␣
↪comm=sshd pid=1995 prio=120 success=1 target_cpu=001
     kworker/0:1-59    [000] d..3   136.677025: sched_switch: prev_
↪comm=kworker/0:1 prev_pid=59 prev_prio=120 prev_state=S ==> next_
↪comm=bash next_pid=1998 next_prio=120
[...]

# cat instances/bar/trace_pipe
     migration/1-14    [001] d.h3   138.732674: softirq_raise:␣
↪vec=3 [action=NET_RX]
          <idle>-0     [001] dNh3   138.732725: softirq_raise:␣
↪vec=3 [action=NET_RX]
            bash-1998  [000] d.h1   138.733101: softirq_raise:␣
↪vec=1 [action=TIMER]
            bash-1998  [000] d.h1   138.733102: softirq_raise:␣
↪vec=9 [action=RCU]
            bash-1998  [000] ..s2   138.733105: softirq_entry:␣
↪vec=1 [action=TIMER]
            bash-1998  [000] ..s2   138.733106: softirq_exit: vec=1␣
↪[action=TIMER]
            bash-1998  [000] ..s2   138.733106: softirq_entry:␣
↪vec=9 [action=RCU]
            bash-1998  [000] ..s2   138.733109: softirq_exit: vec=9␣
↪[action=RCU]
            sshd-1995  [001] d.h1   138.733278: irq_handler_entry:␣
↪irq=21 name=uhci_hcd:usb4
            sshd-1995  [001] d.h1   138.733280: irq_handler_exit:␣
↪irq=21 ret=unhandled
            sshd-1995  [001] d.h1   138.733281: irq_handler_entry:␣
↪irq=21 name=eth0
            sshd-1995  [001] d.h1   138.733283: irq_handler_exit:␣
↪irq=21 ret=handled
[...]

# cat instances/zoot/trace
# tracer: nop
#
# entries-in-buffer/entries-written: 18996/18996   #P:4
#
#                               _-----=> irqs-off
#                              / _----=> need-resched
#                             | / _---=> hardirq/softirq
#                             || / _--=> preempt-depth
#                             ||| /     delay
#           TASK-PID   CPU#   ||||    TIMESTAMP  FUNCTION
#              | |       |    ||||       |         |
            bash-1998  [000] d...   140.733501: sys_write -> 0x2
```

(continues on next page)

(continued from previous page)

```
           bash-1998  [000] d...   140.733504: sys_dup2(oldfd: a,␣
↪newfd: 1)
           bash-1998  [000] d...   140.733506: sys_dup2 -> 0x1
           bash-1998  [000] d...   140.733508: sys_fcntl(fd: a,␣
↪cmd: 1, arg: 0)
           bash-1998  [000] d...   140.733509: sys_fcntl -> 0x1
           bash-1998  [000] d...   140.733510: sys_close(fd: a)
           bash-1998  [000] d...   140.733510: sys_close -> 0x0
           bash-1998  [000] d...   140.733514: sys_rt_
↪sigprocmask(how: 0, nset: 0, oset: 6e2768, sigsetsize: 8)
           bash-1998  [000] d...   140.733515: sys_rt_sigprocmask -
↪> 0x0
           bash-1998  [000] d...   140.733516: sys_rt_
↪sigaction(sig: 2, act: 7fff718846f0, oact: 7fff71884650,␣
↪sigsetsize: 8)
           bash-1998  [000] d...   140.733516: sys_rt_sigaction ->␣
↪0x0
```

You can see that the trace of the top most trace buffer shows only the function tracing. The foo instance displays wakeups and task switches.

To remove the instances, simply delete their directories:

```
# rmdir instances/foo
# rmdir instances/bar
# rmdir instances/zoot
```

Note, if a process has a trace file open in one of the instance directories, the rmdir will fail with EBUSY.

3.29 Stack trace

Since the kernel has a fixed sized stack, it is important not to waste it in functions. A kernel developer must be conscience of what they allocate on the stack. If they add too much, the system can be in danger of a stack overflow, and corruption will occur, usually leading to a system panic.

There are some tools that check this, usually with interrupts periodically checking usage. But if you can perform a check at every function call that will become very useful. As ftrace provides a function tracer, it makes it convenient to check the stack size at every function call. This is enabled via the stack tracer.

CONFIG_STACK_TRACER enables the ftrace stack tracing functionality. To enable it, write a '1' into /proc/sys/kernel/stack_tracer_enabled.

```
# echo 1 > /proc/sys/kernel/stack_tracer_enabled
```

You can also enable it from the kernel command line to trace the stack size of the kernel during boot up, by adding "stacktrace" to the kernel command line parameter.

After running it for a few minutes, the output looks like:

```
# cat stack_max_size
2928

# cat stack_trace
        Depth    Size   Location    (18 entries)
        -----    ----   --------
  0)     2928     224   update_sd_lb_stats+0xbc/0x4ac
  1)     2704     160   find_busiest_group+0x31/0x1f1
  2)     2544     256   load_balance+0xd9/0x662
  3)     2288      80   idle_balance+0xbb/0x130
  4)     2208     128   __schedule+0x26e/0x5b9
  5)     2080      16   schedule+0x64/0x66
  6)     2064     128   schedule_timeout+0x34/0xe0
  7)     1936     112   wait_for_common+0x97/0xf1
  8)     1824      16   wait_for_completion+0x1d/0x1f
  9)     1808     128   flush_work+0xfe/0x119
 10)     1680      16   tty_flush_to_ldisc+0x1e/0x20
 11)     1664      48   input_available_p+0x1d/0x5c
 12)     1616      48   n_tty_poll+0x6d/0x134
 13)     1568      64   tty_poll+0x64/0x7f
 14)     1504     880   do_select+0x31e/0x511
 15)      624     400   core_sys_select+0x177/0x216
 16)      224      96   sys_select+0x91/0xb9
 17)      128     128   system_call_fastpath+0x16/0x1b
```

Note, if -mfentry is being used by gcc, functions get traced before they set up the stack frame. This means that leaf level functions are not tested by the stack tracer when -mfentry is used.

Currently, -mfentry is used by gcc 4.6.0 and above on x86 only.

3.30 More

More details can be found in the source code, in the *kernel/trace/*.c* files.

CHAPTER

FOUR

USING FTRACE TO HOOK TO FUNCTIONS

Written for: 4.14

4.1 Introduction

The ftrace infrastructure was originally created to attach callbacks to the beginning of functions in order to record and trace the flow of the kernel. But callbacks to the start of a function can have other use cases. Either for live kernel patching, or for security monitoring. This document describes how to use ftrace to implement your own function callbacks.

4.2 The ftrace context

> **Warning:** The ability to add a callback to almost any function within the kernel comes with risks. A callback can be called from any context (normal, softirq, irq, and NMI). Callbacks can also be called just before going to idle, during CPU bring up and takedown, or going to user space. This requires extra care to what can be done inside a callback. A callback can be called outside the protective scope of RCU.

The ftrace infrastructure has some protections against recursions and RCU but one must still be very careful how they use the callbacks.

4.3 The ftrace_ops structure

To register a function callback, a ftrace_ops is required. This structure is used to tell ftrace what function should be called as the callback as well as what protections the callback will perform and not require ftrace to handle.

There is only one field that is needed to be set when registering an ftrace_ops with ftrace:

```
struct ftrace_ops ops = {
      .func                    = my_callback_func,
```

(continues on next page)

(continued from previous page)

```
      .flags                        = MY_FTRACE_FLAGS
      .private                      = any_private_data_structure,
};
```

Both .flags and .private are optional. Only .func is required.

To enable tracing call:

```
register_ftrace_function(&ops);
```

To disable tracing call:

```
unregister_ftrace_function(&ops);
```

The above is defined by including the header:

```
#include <linux/ftrace.h>
```

The registered callback will start being called some time after the register_ftrace_function() is called and before it returns. The exact time that callbacks start being called is dependent upon architecture and scheduling of services. The callback itself will have to handle any synchronization if it must begin at an exact moment.

The unregister_ftrace_function() will guarantee that the callback is no longer being called by functions after the unregister_ftrace_function() returns. Note that to perform this guarantee, the unregister_ftrace_function() may take some time to finish.

4.4 The callback function

The prototype of the callback function is as follows (as of v4.14):

```
void callback_func(unsigned long ip, unsigned long parent_ip,
                   struct ftrace_ops *op, struct pt_regs *regs);
```

@ip
: This is the instruction pointer of the function that is being traced. (where the fentry or mcount is within the function)

@parent_ip
: This is the instruction pointer of the function that called the the function being traced (where the call of the function occurred).

@op
: This is a pointer to ftrace_ops that was used to register the callback. This can be used to pass data to the callback via the private pointer.

@regs
: If the FTRACE_OPS_FL_SAVE_REGS or FTRACE_OPS_FL_SAVE_REGS_IF_SUPPORTED flags are set in the ftrace_ops structure, then this will be pointing to the pt_regs structure like it would be if an breakpoint was placed at the start of

the function where ftrace was tracing. Otherwise it either contains garbage, or NULL.

4.5 The ftrace FLAGS

The ftrace_ops flags are all defined and documented in include/linux/ftrace.h. Some of the flags are used for internal infrastructure of ftrace, but the ones that users should be aware of are the following:

FTRACE_OPS_FL_SAVE_REGS
If the callback requires reading or modifying the pt_regs passed to the callback, then it must set this flag. Registering a ftrace_ops with this flag set on an architecture that does not support passing of pt_regs to the callback will fail.

FTRACE_OPS_FL_SAVE_REGS_IF_SUPPORTED
Similar to SAVE_REGS but the registering of a ftrace_ops on an architecture that does not support passing of regs will not fail with this flag set. But the callback must check if regs is NULL or not to determine if the architecture supports it.

FTRACE_OPS_FL_RECURSION_SAFE
By default, a wrapper is added around the callback to make sure that recursion of the function does not occur. That is, if a function that is called as a result of the callback's execution is also traced, ftrace will prevent the callback from being called again. But this wrapper adds some overhead, and if the callback is safe from recursion, it can set this flag to disable the ftrace protection.

Note, if this flag is set, and recursion does occur, it could cause the system to crash, and possibly reboot via a triple fault.

It is OK if another callback traces a function that is called by a callback that is marked recursion safe. Recursion safe callbacks must never trace any function that are called by the callback itself or any nested functions that those functions call.

If this flag is set, it is possible that the callback will also be called with preemption enabled (when CONFIG_PREEMPTION is set), but this is not guaranteed.

FTRACE_OPS_FL_IPMODIFY
Requires FTRACE_OPS_FL_SAVE_REGS set. If the callback is to "hijack" the traced function (have another function called instead of the traced function), it requires setting this flag. This is what live kernel patches uses. Without this flag the pt_regs->ip can not be modified.

Note, only one ftrace_ops with FTRACE_OPS_FL_IPMODIFY set may be registered to any given function at a time.

FTRACE_OPS_FL_RCU
If this is set, then the callback will only be called by functions where RCU is "watching". This is required if the callback function performs any rcu_read_lock() operation.

RCU stops watching when the system goes idle, the time when a CPU is taken down and comes back online, and when entering from kernel to user space and back to kernel space. During these transitions, a callback may be executed and RCU synchronization will not protect it.

FTRACE_OPS_FL_PERMANENT
If this is set on any ftrace ops, then the tracing cannot disabled by writing 0 to the proc sysctl ftrace_enabled. Equally, a callback with the flag set cannot be registered if ftrace_enabled is 0.

Livepatch uses it not to lose the function redirection, so the system stays protected.

4.6 Filtering which functions to trace

If a callback is only to be called from specific functions, a filter must be set up. The filters are added by name, or ip if it is known.

```
int ftrace_set_filter(struct ftrace_ops *ops, unsigned char *buf,
                      int len, int reset);
```

@ops
The ops to set the filter with

@buf
The string that holds the function filter text.

@len
The length of the string.

@reset
Non-zero to reset all filters before applying this filter.

Filters denote which functions should be enabled when tracing is enabled. If @buf is NULL and reset is set, all functions will be enabled for tracing.

The @buf can also be a glob expression to enable all functions that match a specific pattern.

See Filter Commands in `Documentation/trace/ftrace.rst`.

To just trace the schedule function:

```
ret = ftrace_set_filter(&ops, "schedule", strlen("schedule"), 0);
```

To add more functions, call the ftrace_set_filter() more than once with the @reset parameter set to zero. To remove the current filter set and replace it with new functions defined by @buf, have @reset be non-zero.

To remove all the filtered functions and trace all functions:

```
ret = ftrace_set_filter(&ops, NULL, 0, 1);
```

Sometimes more than one function has the same name. To trace just a specific function in this case, ftrace_set_filter_ip() can be used.

```
ret = ftrace_set_filter_ip(&ops, ip, 0, 0);
```

Although the ip must be the address where the call to fentry or mcount is located in the function. This function is used by perf and kprobes that gets the ip address from the user (usually using debug info from the kernel).

If a glob is used to set the filter, functions can be added to a "notrace" list that will prevent those functions from calling the callback. The "notrace" list takes precedence over the "filter" list. If the two lists are non-empty and contain the same functions, the callback will not be called by any function.

An empty "notrace" list means to allow all functions defined by the filter to be traced.

```
int ftrace_set_notrace(struct ftrace_ops *ops, unsigned char *buf,
                       int len, int reset);
```

This takes the same parameters as ftrace_set_filter() but will add the functions it finds to not be traced. This is a separate list from the filter list, and this function does not modify the filter list.

A non-zero @reset will clear the "notrace" list before adding functions that match @buf to it.

Clearing the "notrace" list is the same as clearing the filter list

```
ret = ftrace_set_notrace(&ops, NULL, 0, 1);
```

The filter and notrace lists may be changed at any time. If only a set of functions should call the callback, it is best to set the filters before registering the callback. But the changes may also happen after the callback has been registered.

If a filter is in place, and the @reset is non-zero, and @buf contains a matching glob to functions, the switch will happen during the time of the ftrace_set_filter() call. At no time will all functions call the callback.

```
ftrace_set_filter(&ops, "schedule", strlen("schedule"), 1);

register_ftrace_function(&ops);

msleep(10);

ftrace_set_filter(&ops, "try_to_wake_up", strlen("try_to_wake_up"),
↪1);
```

is not the same as:

```
ftrace_set_filter(&ops, "schedule", strlen("schedule"), 1);

register_ftrace_function(&ops);

msleep(10);

```

(continues on next page)

(continued from previous page)

```
ftrace_set_filter(&ops, NULL, 0, 1);

ftrace_set_filter(&ops, "try_to_wake_up", strlen("try_to_wake_up"),␣
↪0);
```

As the latter will have a short time where all functions will call the callback, between the time of the reset, and the time of the new setting of the filter.

CHAPTER

FIVE

KERNEL PROBES (KPROBES)

Author
Jim Keniston <jkenisto@us.ibm.com>

Author
Prasanna S Panchamukhi <prasanna.panchamukhi@gmail.com>

Author
Masami Hiramatsu <mhiramat@redhat.com>

5.1 Concepts: Kprobes and Return Probes

Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. You can trap at almost any kernel code address[1], specifying a handler routine to be invoked when the breakpoint is hit.

There are currently two types of probes: kprobes, and kretprobes (also called return probes). A kprobe can be inserted on virtually any instruction in the kernel. A return probe fires when a specified function returns.

In the typical case, Kprobes-based instrumentation is packaged as a kernel module. The module's init function installs ("registers") one or more probes, and the exit function unregisters them. A registration function such as register_kprobe() specifies where the probe is to be inserted and what handler is to be called when the probe is hit.

There are also `register_/unregister_*probes()` functions for batch registration/unregistration of a group of `*probes`. These functions can speed up unregistration process when you have to unregister a lot of probes at once.

The next four subsections explain how the different types of probes work and how jump optimization works. They explain certain things that you'll need to know in order to make the best use of Kprobes – e.g., the difference between a pre_handler and a post_handler, and how to use the maxactive and nmissed fields of a kretprobe. But if you're in a hurry to start using Kprobes, you can skip ahead to *Architectures Supported*.

[1] some parts of the kernel code can not be trapped, see *Blacklist*)

5.1.1 How Does a Kprobe Work?

When a kprobe is registered, Kprobes makes a copy of the probed instruction and replaces the first byte(s) of the probed instruction with a breakpoint instruction (e.g., int3 on i386 and x86_64).

When a CPU hits the breakpoint instruction, a trap occurs, the CPU's registers are saved, and control passes to Kprobes via the notifier_call_chain mechanism. Kprobes executes the "pre_handler" associated with the kprobe, passing the handler the addresses of the kprobe struct and the saved registers.

Next, Kprobes single-steps its copy of the probed instruction. (It would be simpler to single-step the actual instruction in place, but then Kprobes would have to temporarily remove the breakpoint instruction. This would open a small time window when another CPU could sail right past the probepoint.)

After the instruction is single-stepped, Kprobes executes the "post_handler," if any, that is associated with the kprobe. Execution then continues with the instruction following the probepoint.

5.1.2 Changing Execution Path

Since kprobes can probe into a running kernel code, it can change the register set, including instruction pointer. This operation requires maximum care, such as keeping the stack frame, recovering the execution path etc. Since it operates on a running kernel and needs deep knowledge of computer architecture and concurrent computing, you can easily shoot your foot.

If you change the instruction pointer (and set up other related registers) in pre_handler, you must return !0 so that kprobes stops single stepping and just returns to the given address. This also means post_handler should not be called anymore.

Note that this operation may be harder on some architectures which use TOC (Table of Contents) for function call, since you have to setup a new TOC for your function in your module, and recover the old one after returning from it.

5.1.3 Return Probes

How Does a Return Probe Work?

When you call register_kretprobe(), Kprobes establishes a kprobe at the entry to the function. When the probed function is called and this probe is hit, Kprobes saves a copy of the return address, and replaces the return address with the address of a "trampoline." The trampoline is an arbitrary piece of code – typically just a nop instruction. At boot time, Kprobes registers a kprobe at the trampoline.

When the probed function executes its return instruction, control passes to the trampoline and that probe is hit. Kprobes' trampoline handler calls the user-specified return handler associated with the kretprobe, then sets the saved instruction pointer to the saved return address, and that's where execution resumes upon return from the trap.

While the probed function is executing, its return address is stored in an object of type kretprobe_instance. Before calling register_kretprobe(), the user sets the maxactive field of the kretprobe struct to specify how many instances of the specified function can be probed simultaneously. register_kretprobe() pre-allocates the indicated number of kretprobe_instance objects.

For example, if the function is non-recursive and is called with a spinlock held, maxactive = 1 should be enough. If the function is non-recursive and can never relinquish the CPU (e.g., via a semaphore or preemption), NR_CPUS should be enough. If maxactive <= 0, it is set to a default value. If CONFIG_PREEMPT is enabled, the default is max(10, 2*NR_CPUS). Otherwise, the default is NR_CPUS.

It's not a disaster if you set maxactive too low; you'll just miss some probes. In the kretprobe struct, the nmissed field is set to zero when the return probe is registered, and is incremented every time the probed function is entered but there is no kretprobe_instance object available for establishing the return probe.

Kretprobe entry-handler

Kretprobes also provides an optional user-specified handler which runs on function entry. This handler is specified by setting the entry_handler field of the kretprobe struct. Whenever the kprobe placed by kretprobe at the function entry is hit, the user-defined entry_handler, if any, is invoked. If the entry_handler returns 0 (success) then a corresponding return handler is guaranteed to be called upon function return. If the entry_handler returns a non-zero error then Kprobes leaves the return address as is, and the kretprobe has no further effect for that particular function instance.

Multiple entry and return handler invocations are matched using the unique kretprobe_instance object associated with them. Additionally, a user may also specify per return-instance private data to be part of each kretprobe_instance object. This is especially useful when sharing private data between corresponding user entry and return handlers. The size of each private data object can be specified at kretprobe registration time by setting the data_size field of the kretprobe struct. This data can be accessed through the data field of each kretprobe_instance object.

In case probed function is entered but there is no kretprobe_instance object available, then in addition to incrementing the nmissed count, the user entry_handler invocation is also skipped.

5.1.4 How Does Jump Optimization Work?

If your kernel is built with CONFIG_OPTPROBES=y (currently this flag is automatically set 'y' on x86/x86-64, non-preemptive kernel) and the "debug.kprobes_optimization" kernel parameter is set to 1 (see sysctl(8)), Kprobes tries to reduce probe-hit overhead by using a jump instruction instead of a breakpoint instruction at each probepoint.

Init a Kprobe

When a probe is registered, before attempting this optimization, Kprobes inserts an ordinary, breakpoint-based kprobe at the specified address. So, even if it's not possible to optimize this particular probepoint, there'll be a probe there.

Safety Check

Before optimizing a probe, Kprobes performs the following safety checks:

- Kprobes verifies that the region that will be replaced by the jump instruction (the "optimized region") lies entirely within one function. (A jump instruction is multiple bytes, and so may overlay multiple instructions.)
- Kprobes analyzes the entire function and verifies that there is no jump into the optimized region. Specifically:
  - the function contains no indirect jump;
  - the function contains no instruction that causes an exception (since the fixup code triggered by the exception could jump back into the optimized region – Kprobes checks the exception tables to verify this);
  - there is no near jump to the optimized region (other than to the first byte).
- For each instruction in the optimized region, Kprobes verifies that the instruction can be executed out of line.

Preparing Detour Buffer

Next, Kprobes prepares a "detour" buffer, which contains the following instruction sequence:

- code to push the CPU's registers (emulating a breakpoint trap)
- a call to the trampoline code which calls user's probe handlers.
- code to restore registers
- the instructions from the optimized region
- a jump back to the original execution path.

Pre-optimization

After preparing the detour buffer, Kprobes verifies that none of the following situations exist:

- The probe has a post_handler.
- Other instructions in the optimized region are probed.
- The probe is disabled.

In any of the above cases, Kprobes won't start optimizing the probe. Since these are temporary situations, Kprobes tries to start optimizing it again if the situation is changed.

If the kprobe can be optimized, Kprobes enqueues the kprobe to an optimizing list, and kicks the kprobe-optimizer workqueue to optimize it. If the to-be-optimized probepoint is hit before being optimized, Kprobes returns control to the original instruction path by setting the CPU's instruction pointer to the copied code in the detour buffer – thus at least avoiding the single-step.

Optimization

The Kprobe-optimizer doesn't insert the jump instruction immediately; rather, it calls synchronize_rcu() for safety first, because it's possible for a CPU to be interrupted in the middle of executing the optimized region[3]. As you know, synchronize_rcu() can ensure that all interruptions that were active when synchronize_rcu() was called are done, but only if CONFIG_PREEMPT=n. So, this version of kprobe optimization supports only kernels with CONFIG_PREEMPT=n[4].

After that, the Kprobe-optimizer calls stop_machine() to replace the optimized region with a jump instruction to the detour buffer, using text_poke_smp().

Unoptimization

When an optimized kprobe is unregistered, disabled, or blocked by another kprobe, it will be unoptimized. If this happens before the optimization is complete, the kprobe is just dequeued from the optimized list. If the optimization has been done, the jump is replaced with the original code (except for an int3 breakpoint in the first byte) by using text_poke_smp().

NOTE for geeks: The jump optimization changes the kprobe's pre_handler behavior. Without optimization, the pre_handler can change the kernel's execution path by changing regs->ip and returning 1. However, when the probe is optimized, that modification is ignored. Thus, if you want to tweak the kernel's execution path, you need to suppress optimization, using one of the following techniques:

- Specify an empty function for the kprobe's post_handler.

or

- Execute 'sysctl -w debug.kprobes_optimization=n'

[3] Please imagine that the 2nd instruction is interrupted and then the optimizer replaces the 2nd instruction with the jump *address* while the interrupt handler is running. When the interrupt returns to original address, there is no valid instruction, and it causes an unexpected result.

[4] This optimization-safety checking may be replaced with the stop-machine method that ksplice uses for supporting a CONFIG_PREEMPT=y kernel.

5.1.5 Blacklist

Kprobes can probe most of the kernel except itself. This means that there are some functions where kprobes cannot probe. Probing (trapping) such functions can cause a recursive trap (e.g. double fault) or the nested probe handler may never be called. Kprobes manages such functions as a blacklist. If you want to add a function into the blacklist, you just need to (1) include linux/kprobes.h and (2) use NOKPROBE_SYMBOL() macro to specify a blacklisted function. Kprobes checks the given probe address against the blacklist and rejects registering it, if the given address is in the blacklist.

5.2 Architectures Supported

Kprobes and return probes are implemented on the following architectures:

- i386 (Supports jump optimization)
- x86_64 (AMD-64, EM64T) (Supports jump optimization)
- ppc64
- ia64 (Does not support probes on instruction slot1.)
- sparc64 (Return probes not yet implemented.)
- arm
- ppc
- mips
- s390
- parisc

5.3 Configuring Kprobes

When configuring the kernel using make menuconfig/xconfig/oldconfig, ensure that CONFIG_KPROBES is set to "y". Under "General setup", look for "Kprobes".

So that you can load and unload Kprobes-based instrumentation modules, make sure "Loadable module support" (CONFIG_MODULES) and "Module unloading" (CONFIG_MODULE_UNLOAD) are set to "y".

Also make sure that CONFIG_KALLSYMS and perhaps even CONFIG_KALLSYMS_ALL are set to "y", since kallsyms_lookup_name() is used by the in-kernel kprobe address resolution code.

If you need to insert a probe in the middle of a function, you may find it useful to "Compile the kernel with debug info" (CONFIG_DEBUG_INFO), so you can use "objdump -d -l vmlinux" to see the source-to-object code mapping.

5.4 API Reference

The Kprobes API includes a "register"function and an "unregister"function for each type of probe. The API also includes "register_*probes" and "unregister_*probes" functions for (un)registering arrays of probes. Here are terse, mini-man-page specifications for these functions and the associated probe handlers that you'll write. See the files in the samples/kprobes/ sub-directory for examples.

5.4.1 register_kprobe

```
#include <linux/kprobes.h>
int register_kprobe(struct kprobe *kp);
```

Sets a breakpoint at the address kp->addr. When the breakpoint is hit, Kprobes calls kp->pre_handler. After the probed instruction is single-stepped, Kprobe calls kp->post_handler. If a fault occurs during execution of kp->pre_handler or kp->post_handler, or during single-stepping of the probed instruction, Kprobes calls kp->fault_handler. Any or all handlers can be NULL. If kp->flags is set KPROBE_FLAG_DISABLED, that kp will be registered but disabled, so, its handlers aren't hit until calling enable_kprobe(kp).

Note:

1. With the introduction of the "symbol_name" field to struct kprobe, the probepoint address resolution will now be taken care of by the kernel. The following will now work:

   ```
   kp.symbol_name = "symbol_name";
   ```

   (64-bit powerpc intricacies such as function descriptors are handled transparently)

2. Use the "offset" field of struct kprobe if the offset into the symbol to install a probepoint is known. This field is used to calculate the probepoint.

3. Specify either the kprobe "symbol_name" OR the "addr". If both are specified, kprobe registration will fail with -EINVAL.

4. With CISC architectures (such as i386 and x86_64), the kprobes code does not validate if the kprobe.addr is at an instruction boundary. Use "offset" with caution.

---

register_kprobe() returns 0 on success, or a negative errno otherwise.

User's pre-handler (kp->pre_handler):

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int pre_handler(struct kprobe *p, struct pt_regs *regs);
```

Called with p pointing to the kprobe associated with the breakpoint, and regs pointing to the struct containing the registers saved when the breakpoint was hit. Return 0 here unless you're a Kprobes geek.

User's post-handler (kp->post_handler):

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
void post_handler(struct kprobe *p, struct pt_regs *regs,
                  unsigned long flags);
```

p and regs are as described for the pre_handler. flags always seems to be zero.

User's fault-handler (kp->fault_handler):

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int fault_handler(struct kprobe *p, struct pt_regs *regs, int␣
↪trapnr);
```

p and regs are as described for the pre_handler. trapnr is the architecture-specific trap number associated with the fault (e.g., on i386, 13 for a general protection fault or 14 for a page fault). Returns 1 if it successfully handled the exception.

5.4.2 register_kretprobe

```
#include <linux/kprobes.h>
int register_kretprobe(struct kretprobe *rp);
```

Establishes a return probe for the function whose address is rp->kp.addr. When that function returns, Kprobes calls rp->handler. You must set rp->maxactive appropriately before you call register_kretprobe(); see "How Does a Return Probe Work?" for details.

register_kretprobe() returns 0 on success, or a negative errno otherwise.

User's return-probe handler (rp->handler):

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int kretprobe_handler(struct kretprobe_instance *ri,
                      struct pt_regs *regs);
```

regs is as described for kprobe.pre_handler. ri points to the kretprobe_instance object, of which the following fields may be of interest:

- ret_addr: the return address
- rp: points to the corresponding kretprobe object
- task: points to the corresponding task struct
- **data: points to per return-instance private data; see "Kretprobe** entry-handler" for details.

The regs_return_value(regs) macro provides a simple abstraction to extract the return value from the appropriate register as defined by the architecture's ABI.

The handler's return value is currently ignored.

5.4.3 unregister_*probe

```
#include <linux/kprobes.h>
void unregister_kprobe(struct kprobe *kp);
void unregister_kretprobe(struct kretprobe *rp);
```

Removes the specified probe. The unregister function can be called at any time after the probe has been registered.

Note: If the functions find an incorrect probe (ex. an unregistered probe), they clear the addr field of the probe.

5.4.4 register_*probes

```
#include <linux/kprobes.h>
int register_kprobes(struct kprobe **kps, int num);
int register_kretprobes(struct kretprobe **rps, int num);
```

Registers each of the num probes in the specified array. If any error occurs during registration, all probes in the array, up to the bad probe, are safely unregistered before the register_*probes function returns.

- kps/rps: an array of pointers to `*probe` data structures
- num: the number of the array entries.

Note: You have to allocate(or define) an array of pointers and set all of the array entries before using these functions.

5.4.5 unregister_*probes

```
#include <linux/kprobes.h>
void unregister_kprobes(struct kprobe **kps, int num);
void unregister_kretprobes(struct kretprobe **rps, int num);
```

Removes each of the num probes in the specified array at once.

Note: If the functions find some incorrect probes (ex. unregistered probes) in the specified array, they clear the addr field of those incorrect probes. However, other probes in the array are unregistered correctly.

5.4.6 disable_*probe

```
#include <linux/kprobes.h>
int disable_kprobe(struct kprobe *kp);
int disable_kretprobe(struct kretprobe *rp);
```

Temporarily disables the specified `*probe`. You can enable it again by using enable_*probe(). You must specify the probe which has been registered.

5.4.7 enable_*probe

```
#include <linux/kprobes.h>
int enable_kprobe(struct kprobe *kp);
int enable_kretprobe(struct kretprobe *rp);
```

Enables `*probe` which has been disabled by disable_*probe(). You must specify the probe which has been registered.

5.5 Kprobes Features and Limitations

Kprobes allows multiple probes at the same address. Also, a probepoint for which there is a post_handler cannot be optimized. So if you install a kprobe with a post_handler, at an optimized probepoint, the probepoint will be unoptimized automatically.

In general, you can install a probe anywhere in the kernel. In particular, you can probe interrupt handlers. Known exceptions are discussed in this section.

The register_*probe functions will return -EINVAL if you attempt to install a probe in the code that implements Kprobes (mostly kernel/kprobes.c and `arch/*/kernel/kprobes.c`, but also functions such as do_page_fault and notifier_call_chain).

If you install a probe in an inline-able function, Kprobes makes no attempt to chase down all inline instances of the function and install probes there. gcc may inline a function without being asked, so keep this in mind if you're not seeing the probe hits you expect.

A probe handler can modify the environment of the probed function – e.g., by modifying kernel data structures, or by modifying the contents of the pt_regs struct (which are restored to the registers upon return from the breakpoint). So Kprobes can be used, for example, to install a bug fix or to inject faults for testing. Kprobes, of course, has no way to distinguish the deliberately injected faults from the accidental ones. Don't drink and probe.

Kprobes makes no attempt to prevent probe handlers from stepping on each other – e.g., probing printk() and then calling printk() from a probe handler. If a probe handler hits a probe, that second probe's handlers won't be run in that instance, and the kprobe.nmissed member of the second probe will be incremented.

As of Linux v2.6.15-rc1, multiple handlers (or multiple instances of the same handler) may run concurrently on different CPUs.

Kprobes does not use mutexes or allocate memory except during registration and unregistration.

Probe handlers are run with preemption disabled or interrupt disabled, which depends on the architecture and optimization state. (e.g., kretprobe handlers and optimized kprobe handlers run without interrupt disabled on x86/x86-64). In any case, your handler should not yield the CPU (e.g., by attempting to acquire a semaphore, or waiting I/O).

Since a return probe is implemented by replacing the return address with the trampoline' s address, stack backtraces and calls to __builtin_return_address() will typically yield the trampoline' s address instead of the real return address for kretprobed functions. (As far as we can tell, __builtin_return_address() is used only for instrumentation and error reporting.)

If the number of times a function is called does not match the number of times it returns, registering a return probe on that function may produce undesirable results. In such a case, a line: kretprobe BUG!: Processing kretprobe d000000000041aa8 @ c00000000004f48c gets printed. With this information, one will be able to correlate the exact instance of the kretprobe that caused the problem. We have the do_exit() case covered. do_execve() and do_fork() are not an issue. We're unaware of other specific cases where this could be a problem.

If, upon entry to or exit from a function, the CPU is running on a stack other than that of the current task, registering a return probe on that function may produce undesirable results. For this reason, Kprobes doesn' t support return probes (or kprobes) on the x86_64 version of __switch_to(); the registration functions return -EINVAL.

On x86/x86-64, since the Jump Optimization of Kprobes modifies instructions widely, there are some limitations to optimization. To explain it, we introduce some terminology. Imagine a 3-instruction sequence consisting of a two 2-byte instructions and one 3-byte instruction.

```
        IA
        |
[-2][-1][0][1][2][3][4][5][6][7]
        [ins1][ins2][  ins3 ]
        [<-     DCR       ->]
        [<- JTPR ->]

ins1: 1st Instruction
ins2: 2nd Instruction
ins3: 3rd Instruction
IA:  Insertion Address
JTPR: Jump Target Prohibition Region
DCR: Detoured Code Region
```

The instructions in DCR are copied to the out-of-line buffer of the kprobe, because the bytes in DCR are replaced by a 5-byte jump instruction. So there are several limitations.

a) The instructions in DCR must be relocatable.

b) The instructions in DCR must not include a call instruction.

c) JTPR must not be targeted by any jump or call instruction.

d) DCR must not straddle the border between functions.

Anyway, these limitations are checked by the in-kernel instruction decoder, so you don't need to worry about that.

5.6 Probe Overhead

On a typical CPU in use in 2005, a kprobe hit takes 0.5 to 1.0 microseconds to process. Specifically, a benchmark that hits the same probepoint repeatedly, firing a simple handler each time, reports 1-2 million hits per second, depending on the architecture. A return-probe hit typically takes 50-75% longer than a kprobe hit. When you have a return probe set on a function, adding a kprobe at the entry to that function adds essentially no overhead.

Here are sample overhead figures (in usec) for different architectures:

```
k = kprobe; r = return probe; kr = kprobe + return probe
on same function

i386: Intel Pentium M, 1495 MHz, 2957.31 bogomips
k = 0.57 usec; r = 0.92; kr = 0.99

x86_64: AMD Opteron 246, 1994 MHz, 3971.48 bogomips
k = 0.49 usec; r = 0.80; kr = 0.82

ppc64: POWER5 (gr), 1656 MHz (SMT disabled, 1 virtual CPU per␣
↪physical CPU)
k = 0.77 usec; r = 1.26; kr = 1.45
```

5.6.1 Optimized Probe Overhead

Typically, an optimized kprobe hit takes 0.07 to 0.1 microseconds to process. Here are sample overhead figures (in usec) for x86 architectures:

```
k = unoptimized kprobe, b = boosted (single-step skipped), o =␣
↪optimized kprobe,
r = unoptimized kretprobe, rb = boosted kretprobe, ro = optimized␣
↪kretprobe.

i386: Intel(R) Xeon(R) E5410, 2.33GHz, 4656.90 bogomips
k = 0.80 usec; b = 0.33; o = 0.05; r = 1.10; rb = 0.61; ro = 0.33

x86-64: Intel(R) Xeon(R) E5410, 2.33GHz, 4656.90 bogomips
k = 0.99 usec; b = 0.43; o = 0.06; r = 1.24; rb = 0.68; ro = 0.30
```

5.7 TODO

a. SystemTap (http://sourceware.org/systemtap): Provides a simplified programming interface for probe-based instrumentation. Try it out.

b. Kernel return probes for sparc64.

c. Support for other architectures.

d. User-space probes.

e. Watchpoint probes (which fire on data references).

5.8 Kprobes Example

See samples/kprobes/kprobe_example.c

5.9 Kretprobes Example

See samples/kprobes/kretprobe_example.c

5.10 Deprecated Features

Jprobes is now a deprecated feature. People who are depending on it should migrate to other tracing features or use older kernels. Please consider to migrate your tool to one of the following options:

- Use trace-event to trace target function with arguments.

  trace-event is a low-overhead (and almost no visible overhead if it is off) statically defined event interface. You can define new events and trace it via ftrace or any other tracing tools.

  See the following urls:

  - https://lwn.net/Articles/379903/
  - https://lwn.net/Articles/381064/
  - https://lwn.net/Articles/383362/

- Use ftrace dynamic events (kprobe event) with perf-probe.

  If you build your kernel with debug info (CONFIG_DEBUG_INFO=y), you can find which register/stack is assigned to which local variable or arguments by using perf-probe and set up new event to trace it.

  See following documents:

  - *Kprobe-based Event Tracing*
  - *Event Tracing*
  - tools/perf/Documentation/perf-probe.txt

5.11 The kprobes debugfs interface

With recent kernels (> 2.6.20) the list of registered kprobes is visible under the /sys/kernel/debug/kprobes/ directory (assuming debugfs is mounted at //sys/kernel/debug).

/sys/kernel/debug/kprobes/list: Lists all registered probes on the system:

```
c015d71a  k  vfs_read+0x0
c03dedc5  r  tcp_v4_rcv+0x0
```

The first column provides the kernel address where the probe is inserted. The second column identifies the type of probe (k - kprobe and r - kretprobe) while the third column specifies the symbol+offset of the probe. If the probed function belongs to a module, the module name is also specified. Following columns show probe status. If the probe is on a virtual address that is no longer valid (module init sections, module virtual addresses that correspond to modules that' ve been unloaded), such probes are marked with [GONE]. If the probe is temporarily disabled, such probes are marked with [DISABLED]. If the probe is optimized, it is marked with [OPTIMIZED]. If the probe is ftrace-based, it is marked with [FTRACE].

/sys/kernel/debug/kprobes/enabled: Turn kprobes ON/OFF forcibly.

Provides a knob to globally and forcibly turn registered kprobes ON or OFF. By default, all kprobes are enabled. By echoing "0" to this file, all registered probes will be disarmed, till such time a "1" is echoed to this file. Note that this knob just disarms and arms all kprobes and doesn' t change each probe' s disabling state. This means that disabled kprobes (marked [DISABLED]) will be not enabled if you turn ON all kprobes by this knob.

5.12 The kprobes sysctl interface

/proc/sys/debug/kprobes-optimization: Turn kprobes optimization ON/OFF.

When CONFIG_OPTPROBES=y, this sysctl interface appears and it provides a knob to globally and forcibly turn jump optimization (see section *How Does Jump Optimization Work?*) ON or OFF. By default, jump optimization is allowed (ON). If you echo "0" to this file or set "debug.kprobes_optimization" to 0 via sysctl, all optimized probes will be unoptimized, and any new probes registered after that will not be optimized.

Note that this knob *changes* the optimized state. This means that optimized probes (marked [OPTIMIZED]) will be unoptimized ([OPTIMIZED] tag will be removed). If the knob is turned on, they will be optimized again.

5.13 References

For additional information on Kprobes, refer to the following URLs:

- https://www.ibm.com/developerworks/library/l-kprobes/index.html
- https://www.kernel.org/doc/ols/2006/ols2006v2-pages-109-124.pdf

CHAPTER

SIX

KPROBE-BASED EVENT TRACING

Author
Masami Hiramatsu

6.1 Overview

These events are similar to tracepoint based events. Instead of Tracepoint, this is based on kprobes (kprobe and kretprobe). So it can probe wherever kprobes can probe (this means, all functions except those with __kprobes/nokprobe_inline annotation and those marked NOKPROBE_SYMBOL). Unlike the Tracepoint based event, this can be added and removed dynamically, on the fly.

To enable this feature, build your kernel with CONFIG_KPROBE_EVENTS=y.

Similar to the events tracer, this doesn't need to be activated via current_tracer. Instead of that, add probe points via /sys/kernel/debug/tracing/kprobe_events, and enable it via /sys/kernel/debug/tracing/events/kprobes/<EVENT>/enable.

You can also use /sys/kernel/debug/tracing/dynamic_events instead of kprobe_events. That interface will provide unified access to other dynamic events too.

6.2 Synopsis of kprobe_events

```
 p[:[GRP/]EVENT] [MOD:]SYM[+offs]|MEMADDR [FETCHARGS]  : Set a probe
 r[MAXACTIVE][:[GRP/]EVENT] [MOD:]SYM[+0] [FETCHARGS]  : Set a return probe
 p:[GRP/]EVENT] [MOD:]SYM[+0]%return [FETCHARGS]       : Set a return probe
 -:[GRP/]EVENT                                         : Clear a probe

GRP             : Group name. If omitted, use "kprobes" for it.
EVENT           : Event name. If omitted, the event name is generated
                  based on SYM+offs or MEMADDR.
MOD             : Module name which has given SYM.
SYM[+offs]      : Symbol+offset where the probe is inserted.
SYM%return      : Return address of the symbol
```

(continues on next page)

(continued from previous page)

```
MEMADDR         : Address where the probe is inserted.
MAXACTIVE       : Maximum number of instances of the specified␣
↪function that
                  can be probed simultaneously, or 0 for the default␣
↪value
                  as defined in Documentation/trace/kprobes.rst␣
↪section 1.3.1.

FETCHARGS       : Arguments. Each probe can have up to 128 args.
 %REG           : Fetch register REG
 @ADDR          : Fetch memory at ADDR (ADDR should be in kernel)
 @SYM[+|-offs]  : Fetch memory at SYM +|- offs (SYM should be a data␣
↪symbol)
 $stackN        : Fetch Nth entry of stack (N >= 0)
 $stack         : Fetch stack address.
 $argN          : Fetch the Nth function argument. (N >= 1) (\*1)
 $retval        : Fetch return value.(\*2)
 $comm          : Fetch current task comm.
 +|-[u]OFFS(FETCHARG) : Fetch memory at FETCHARG +|- OFFS address.(\
↪*3)(\*4)
 \IMM           : Store an immediate value to the argument.
 NAME=FETCHARG  : Set NAME as the argument name of FETCHARG.
 FETCHARG:TYPE  : Set TYPE as the type of FETCHARG. Currently, basic␣
↪types
                  (u8/u16/u32/u64/s8/s16/s32/s64), hexadecimal types
                  (x8/x16/x32/x64), "string", "ustring" and bitfield
                  are supported.

 (\*1) only for the probe on function entry (offs == 0).
 (\*2) only for return probe.
 (\*3) this is useful for fetching a field of data structures.
 (\*4) "u" means user-space dereference. See :ref:`user_mem_access`.
```

6.3 Types

Several types are supported for fetch-args. Kprobe tracer will access memory by given type. Prefix 's' and 'u' means those types are signed and unsigned respectively. 'x' prefix implies it is unsigned. Traced arguments are shown in decimal ('s' and 'u') or hexadecimal ('x'). Without type casting, 'x32' or 'x64' is used depends on the architecture (e.g. x86-32 uses x32, and x86-64 uses x64). These value types can be an array. To record array data, you can add '[N]' (where N is a fixed number, less than 64) to the base type. E.g. 'x16[4]' means an array of x16 (2bytes hex) with 4 elements. Note that the array can be applied to memory type fetchargs, you can not apply it to registers/stack-entries etc. (for example, '$stack1:x8[8]' is wrong, but '+8($stack):x8[8]' is OK.) String type is a special type, which fetches a "null-terminated" string from kernel space. This means it will fail and store NULL if the string container has been paged out. "ustring" type is an alternative of string for user-space. See *User Memory Access* for more info.. The string array type is a bit different from other types. For other base types, <base-type>[1] is equal to <base-type> (e.g. +0(%di):x32[1] is same as +0(%di):x32.) But string[1] is not equal to string. The string type itself represents "char array", but string array type represents "char * array". So, for example, +0(%di):string[1] is equal to +0(+0(%di)):string. Bitfield is another special type, which takes 3 parameters, bit-width, bit- offset, and container-size (usually 32). The syntax is:

```
b<bit-width>@<bit-offset>/<container-size>
```

Symbol type('symbol') is an alias of u32 or u64 type (depends on BITS_PER_LONG) which shows given pointer in "symbol+offset" style. For $comm, the default type is "string"; any other type is invalid.

6.4 User Memory Access

Kprobe events supports user-space memory access. For that purpose, you can use either user-space dereference syntax or 'ustring' type.

The user-space dereference syntax allows you to access a field of a data structure in user-space. This is done by adding the "u" prefix to the dereference syntax. For example, +u4(%si) means it will read memory from the address in the register %si offset by 4, and the memory is expected to be in user-space. You can use this for strings too, e.g. +u0(%si):string will read a string from the address in the register %si that is expected to be in user- space. 'ustring' is a shortcut way of performing the same task. That is, +0(%si):ustring is equivalent to +u0(%si):string.

Note that kprobe-event provides the user-memory access syntax but it doesn't use it transparently. This means if you use normal dereference or string type for user memory, it might fail, and may always fail on some archs. The user has to carefully check if the target data is in kernel or user space.

6.5 Per-Probe Event Filtering

Per-probe event filtering feature allows you to set different filter on each probe and gives you what arguments will be shown in trace buffer. If an event name is specified right after 'p:' or 'r:' in kprobe_events, it adds an event under tracing/events/kprobes/<EVENT>, at the directory you can see 'id', 'enable', 'format', 'filter' and 'trigger'.

enable:
You can enable/disable the probe by writing 1 or 0 on it.

format:
This shows the format of this probe event.

filter:
You can write filtering rules of this event.

id:
This shows the id of this probe event.

trigger:
This allows to install trigger commands which are executed when the event is hit (for details, see *Event Tracing*, section 6).

6.6 Event Profiling

You can check the total number of probe hits and probe miss-hits via /sys/kernel/debug/tracing/kprobe_profile. The first column is event name, the second is the number of probe hits, the third is the number of probe miss-hits.

6.7 Kernel Boot Parameter

You can add and enable new kprobe events when booting up the kernel by "kprobe_event=" parameter. The parameter accepts a semicolon-delimited kprobe events, which format is similar to the kprobe_events. The difference is that the probe definition parameters are comma-delimited instead of space. For example, adding myprobe event on do_sys_open like below

> p:myprobe do_sys_open dfd=%ax filename=%dx flags=%cx mode=+4($stack)

should be below for kernel boot parameter (just replace spaces with comma)

> p:myprobe,do_sys_open,dfd=%ax,filename=%dx,flags=%cx,mode=+4($stack)

6.8 Usage examples

To add a probe as a new event, write a new definition to kprobe_events as below:

```
echo 'p:myprobe do_sys_open dfd=%ax filename=%dx flags=%cx mode=+4($stack)' > /sys/kernel/debug/tracing/kprobe_events
```

This sets a kprobe on the top of do_sys_open() function with recording 1st to 4th arguments as "myprobe" event. Note, which register/stack entry is assigned to each function argument depends on arch-specific ABI. If you unsure the ABI, please try to use probe subcommand of perf-tools (you can find it under tools/perf/). As this example shows, users can choose more familiar names for each arguments.

```
echo 'r:myretprobe do_sys_open $retval' >> /sys/kernel/debug/tracing/kprobe_events
```

This sets a kretprobe on the return point of do_sys_open() function with recording return value as "myretprobe" event. You can see the format of these events via /sys/kernel/debug/tracing/events/kprobes/<EVENT>/format.

```
cat /sys/kernel/debug/tracing/events/kprobes/myprobe/format
name: myprobe
ID: 780
```

(continues on next page)

(continued from previous page)

```
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 __probe_ip; offset:12;      size:4; signed:0;
        field:int __probe_nargs;        offset:16;      size:4; signed:1;
        field:unsigned long dfd;        offset:20;      size:4; signed:0;
        field:unsigned long filename;   offset:24;      size:4; signed:0;
        field:unsigned long flags;      offset:28;      size:4; signed:0;
        field:unsigned long mode;       offset:32;      size:4; signed:0;


print fmt: "(%lx) dfd=%lx filename=%lx flags=%lx mode=%lx", REC->__probe_ip,
REC->dfd, REC->filename, REC->flags, REC->mode
```

You can see that the event has 4 arguments as in the expressions you specified.

```
echo > /sys/kernel/debug/tracing/kprobe_events
```

This clears all probe points.

Or,

```
echo -:myprobe >> kprobe_events
```

This clears probe points selectively.

Right after definition, each event is disabled by default. For tracing these events, you need to enable it.

```
echo 1 > /sys/kernel/debug/tracing/events/kprobes/myprobe/enable
echo 1 > /sys/kernel/debug/tracing/events/kprobes/myretprobe/enable
```

Use the following command to start tracing in an interval.

```
# echo 1 > tracing_on
Open something...
# echo 0 > tracing_on
```

And you can see the traced information via /sys/kernel/debug/tracing/trace.

```
cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
           <...>-1447  [001] 1038282.286875: myprobe: (do_sys_
↪open+0x0/0xd6) dfd=3 filename=7fffd1ec4440 flags=8000 mode=0
           <...>-1447  [001] 1038282.286878: myretprobe: (sys_
↪openat+0xc/0xe <- do_sys_open) $retval=fffffffffffffffe
           <...>-1447  [001] 1038282.286885: myprobe: (do_sys_
↪open+0x0/0xd6) dfd=ffffff9c filename=40413c flags=8000 mode=1b6
           <...>-1447  [001] 1038282.286915: myretprobe: (sys_
↪open+0x1b/0x1d <- do_sys_open) $retval=3
           <...>-1447  [001] 1038282.286969: myprobe: (do_sys_
↪open+0x0/0xd6) dfd=ffffff9c filename=4041c6 flags=98800 mode=10
           <...>-1447  [001] 1038282.286976: myretprobe: (sys_
↪open+0x1b/0x1d <- do_sys_open) $retval=3
```

Each line shows when the kernel hits an event, and <- SYMBOL means kernel returns from SYMBOL(e.g. "sys_open+0x1b/0x1d <- do_sys_open" means kernel returns from do_sys_open to sys_open+0x1b).

CHAPTER

SEVEN

UPROBE-TRACER: UPROBE-BASED EVENT TRACING

Author
Srikar Dronamraju

7.1 Overview

Uprobe based trace events are similar to kprobe based trace events. To enable this feature, build your kernel with CONFIG_UPROBE_EVENTS=y.

Similar to the kprobe-event tracer, this doesn't need to be activated via current_tracer. Instead of that, add probe points via /sys/kernel/debug/tracing/uprobe_events, and enable it via /sys/kernel/debug/tracing/events/uprobes/<EVENT>/enable.

However unlike kprobe-event tracer, the uprobe event interface expects the user to calculate the offset of the probepoint in the object.

You can also use /sys/kernel/debug/tracing/dynamic_events instead of uprobe_events. That interface will provide unified access to other dynamic events too.

7.2 Synopsis of uprobe_tracer

```
p[:[GRP/]EVENT] PATH:OFFSET [FETCHARGS] : Set a uprobe
r[:[GRP/]EVENT] PATH:OFFSET [FETCHARGS] : Set a return uprobe␣
↪(uretprobe)
p[:[GRP/]EVENT] PATH:OFFSET%return [FETCHARGS] : Set a return␣
↪uprobe (uretprobe)
-:[GRP/]EVENT                           : Clear uprobe or uretprobe␣
↪event

GRP           : Group name. If omitted, "uprobes" is the default␣
↪value.
EVENT         : Event name. If omitted, the event name is generated␣
↪based
                on PATH+OFFSET.
PATH          : Path to an executable or a library.
OFFSET        : Offset where the probe is inserted.
```

(continues on next page)

(continued from previous page)

```
OFFSET%return : Offset where the return probe is inserted.

FETCHARGS     : Arguments. Each probe can have up to 128 args.
 %REG         : Fetch register REG
 @ADDR        : Fetch memory at ADDR (ADDR should be in userspace)
 @+OFFSET     : Fetch memory at OFFSET (OFFSET from same file as␣
↪PATH)
 $stackN      : Fetch Nth entry of stack (N >= 0)
 $stack       : Fetch stack address.
 $retval      : Fetch return value.(\*1)
 $comm        : Fetch current task comm.
 +|-[u]OFFS(FETCHARG) : Fetch memory at FETCHARG +|- OFFS address.(\
↪*2)(\*3)
 \IMM         : Store an immediate value to the argument.
 NAME=FETCHARG     : Set NAME as the argument name of FETCHARG.
 FETCHARG:TYPE     : Set TYPE as the type of FETCHARG. Currently,␣
↪basic types
                     (u8/u16/u32/u64/s8/s16/s32/s64), hexadecimal␣
↪types
                     (x8/x16/x32/x64), "string" and bitfield are␣
↪supported.

(\*1) only for return probe.
(\*2) this is useful for fetching a field of data structures.
(\*3) Unlike kprobe event, "u" prefix will just be ignored, becuse␣
↪uprobe
      events can access only user-space memory.
```

7.3 Types

Several types are supported for fetch-args. Uprobe tracer will access memory by given type. Prefix ‘s’ and ‘u’ means those types are signed and unsigned respectively. ‘x’ prefix implies it is unsigned. Traced arguments are shown in decimal (‘s’ and ‘u’) or hexadecimal (‘x’). Without type casting, ‘x32’ or ‘x64’ is used depends on the architecture (e.g. x86-32 uses x32, and x86-64 uses x64). String type is a special type, which fetches a “null-terminated” string from user space. Bitfield is another special type, which takes 3 parameters, bit-width, bit- offset, and container-size (usually 32). The syntax is:

```
b<bit-width>@<bit-offset>/<container-size>
```

For $comm, the default type is “string” ; any other type is invalid.

7.4 Event Profiling

You can check the total number of probe hits per event via /sys/kernel/debug/tracing/uprobe_profile. The first column is the filename, the second is the event name, the third is the number of probe hits.

7.5 Usage examples

- Add a probe as a new uprobe event, write a new definition to uprobe_events as below (sets a uprobe at an offset of 0x4245c0 in the executable /bin/bash):

```
echo 'p /bin/bash:0x4245c0' > /sys/kernel/debug/tracing/uprobe_
↪events
```

- Add a probe as a new uretprobe event:

```
echo 'r /bin/bash:0x4245c0' > /sys/kernel/debug/tracing/uprobe_
↪events
```

- Unset registered event:

```
echo '-:p_bash_0x4245c0' >> /sys/kernel/debug/tracing/uprobe_
↪events
```

- Print out the events that are registered:

```
cat /sys/kernel/debug/tracing/uprobe_events
```

- Clear all events:

```
echo > /sys/kernel/debug/tracing/uprobe_events
```

Following example shows how to dump the instruction pointer and %ax register at the probed text address. Probe zfree function in /bin/zsh:

```
# cd /sys/kernel/debug/tracing/
# cat /proc/`pgrep zsh`/maps | grep /bin/zsh | grep r-xp
00400000-0048a000 r-xp 00000000 08:03 130904 /bin/zsh
# objdump -T /bin/zsh | grep -w zfree
0000000000446420 g    DF .text  0000000000000012  Base        zfree
```

0x46420 is the offset of zfree in object /bin/zsh that is loaded at 0x00400000. Hence the command to uprobe would be:

```
# echo 'p:zfree_entry /bin/zsh:0x46420 %ip %ax' > uprobe_events
```

And the same for the uretprobe would be:

```
# echo 'r:zfree_exit /bin/zsh:0x46420 %ip %ax' >> uprobe_events
```

Note: User has to explicitly calculate the offset of the probe-point in the object.

We can see the events that are registered by looking at the uprobe_events file.

```
# cat uprobe_events
p:uprobes/zfree_entry /bin/zsh:0x00046420 arg1=%ip arg2=%ax
r:uprobes/zfree_exit /bin/zsh:0x00046420 arg1=%ip arg2=%ax
```

Format of events can be seen by viewing the file events/uprobes/zfree_entry/format.

```
# cat events/uprobes/zfree_entry/format
name: zfree_entry
ID: 922
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:int common_padding;                 offset:8;  size:4;␣
↪signed:1;

     field:unsigned long __probe_ip;           offset:12; size:4;␣
↪signed:0;
     field:u32 arg1;                           offset:16; size:4;␣
↪signed:0;
     field:u32 arg2;                           offset:20; size:4;␣
↪signed:0;

print fmt: "(%lx) arg1=%lx arg2=%lx", REC->__probe_ip, REC->arg1,␣
↪REC->arg2
```

Right after definition, each event is disabled by default. For tracing these events, you need to enable it by:

```
# echo 1 > events/uprobes/enable
```

Lets start tracing, sleep for some time and stop tracing.

```
# echo 1 > tracing_on
# sleep 20
# echo 0 > tracing_on
```

Also, you can disable the event by:

```
# echo 0 > events/uprobes/enable
```

And you can see the traced information via /sys/kernel/debug/tracing/trace.

```
# cat trace
# tracer: nop
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
             zsh-24842 [006] 258544.995456: zfree_entry: (0x446420)␣
↪arg1=446420 arg2=79
             zsh-24842 [007] 258545.000270: zfree_exit:  (0x446540
↪<- 0x446420) arg1=446540 arg2=0
             zsh-24842 [002] 258545.043929: zfree_entry: (0x446420)␣
↪arg1=446420 arg2=79
             zsh-24842 [004] 258547.046129: zfree_exit:  (0x446540
↪<- 0x446420) arg1=446540 arg2=0
```

Output shows us uprobe was triggered for a pid 24842 with ip being 0x446420 and contents of ax register being 79. And uretprobe was triggered with ip at 0x446540 with counterpart function entry at 0x446420.

CHAPTER

EIGHT

USING THE LINUX KERNEL TRACEPOINTS

Author
Mathieu Desnoyers

This document introduces Linux Kernel Tracepoints and their use. It provides examples of how to insert tracepoints in the kernel and connect probe functions to them and provides some examples of probe functions.

8.1 Purpose of tracepoints

A tracepoint placed in code provides a hook to call a function (probe) that you can provide at runtime. A tracepoint can be "on" (a probe is connected to it) or "off" (no probe is attached). When a tracepoint is "off" it has no effect, except for adding a tiny time penalty (checking a condition for a branch) and space penalty (adding a few bytes for the function call at the end of the instrumented function and adds a data structure in a separate section). When a tracepoint is "on", the function you provide is called each time the tracepoint is executed, in the execution context of the caller. When the function provided ends its execution, it returns to the caller (continuing from the tracepoint site).

You can put tracepoints at important locations in the code. They are lightweight hooks that can pass an arbitrary number of parameters, which prototypes are described in a tracepoint declaration placed in a header file.

They can be used for tracing and performance accounting.

8.2 Usage

Two elements are required for tracepoints :

- A tracepoint definition, placed in a header file.
- The tracepoint statement, in C code.

In order to use tracepoints, you should include linux/tracepoint.h.

In include/trace/events/subsys.h:

```
#undef TRACE_SYSTEM
#define TRACE_SYSTEM subsys
```

(continues on next page)

(continued from previous page)

```
#if !defined(_TRACE_SUBSYS_H) || defined(TRACE_HEADER_MULTI_READ)
#define _TRACE_SUBSYS_H

#include <linux/tracepoint.h>

DECLARE_TRACE(subsys_eventname,
        TP_PROTO(int firstarg, struct task_struct *p),
        TP_ARGS(firstarg, p));

#endif /* _TRACE_SUBSYS_H */

/* This part must be outside protection */
#include <trace/define_trace.h>
```

In subsys/file.c (where the tracing statement must be added):

```
#include <trace/events/subsys.h>

#define CREATE_TRACE_POINTS
DEFINE_TRACE(subsys_eventname);

void somefct(void)
{
        ...
        trace_subsys_eventname(arg, task);
        ...
}
```

Where :

- subsys_eventname is an identifier unique to your event
  - subsys is the name of your subsystem.
  - eventname is the name of the event to trace.
- *TP_PROTO(int firstarg, struct task_struct *p)* is the prototype of the function called by this tracepoint.
- *TP_ARGS(firstarg, p)* are the parameters names, same as found in the prototype.
- if you use the header in multiple source files, *#define CREATE_TRACE_POINTS* should appear only in one source file.

Connecting a function (probe) to a tracepoint is done by providing a probe (function to call) for the specific tracepoint through register_trace_subsys_eventname(). Removing a probe is done through unregister_trace_subsys_eventname(); it will remove the probe.

tracepoint_synchronize_unregister() must be called before the end of the module exit function to make sure there is no caller left using the probe. This, and the fact that preemption is disabled around the probe call, make sure that probe removal and module unload are safe.

The tracepoint mechanism supports inserting multiple instances of the same tracepoint, but a single definition must be made of a given tracepoint name over all the kernel to make sure no type conflict will occur. Name mangling of the tracepoints is done using the prototypes to make sure typing is correct. Verification of probe type correctness is done at the registration site by the compiler. Tracepoints can be put in inline functions, inlined static functions, and unrolled loops as well as regular functions.

The naming scheme "subsys_event" is suggested here as a convention intended to limit collisions. Tracepoint names are global to the kernel: they are considered as being the same whether they are in the core kernel image or in modules.

If the tracepoint has to be used in kernel modules, an EXPORT_TRACEPOINT_SYMBOL_GPL() or EXPORT_TRACEPOINT_SYMBOL() can be used to export the defined tracepoints.

If you need to do a bit of work for a tracepoint parameter, and that work is only used for the tracepoint, that work can be encapsulated within an if statement with the following:

```
if (trace_foo_bar_enabled()) {
        int i;
        int tot = 0;

        for (i = 0; i < count; i++)
                tot += calculate_nuggets();

        trace_foo_bar(tot);
}
```

All trace_<tracepoint>() calls have a matching trace_<tracepoint>_enabled() function defined that returns true if the tracepoint is enabled and false otherwise. The trace_<tracepoint>() should always be within the block of the if (trace_<tracepoint>_enabled()) to prevent races between the tracepoint being enabled and the check being seen.

The advantage of using the trace_<tracepoint>_enabled() is that it uses the static_key of the tracepoint to allow the if statement to be implemented with jump labels and avoid conditional branches.

Note: The convenience macro TRACE_EVENT provides an alternative way to define tracepoints. Check http://lwn.net/Articles/379903, http://lwn.net/Articles/381064 and http://lwn.net/Articles/383362 for a series of articles with more details.

If you require calling a tracepoint from a header file, it is not recommended to call one directly or to use the trace_<tracepoint>_enabled() function call, as tracepoints in header files can have side effects if a header is included from a file that has CREATE_TRACE_POINTS set, as well as the trace_<tracepoint>() is not that small of an inline and can bloat the kernel if used by other inlined functions. Instead, include tracepoint-defs.h and use tracepoint_enabled().

In a C file:

```
void do_trace_foo_bar_wrapper(args)
{
        trace_foo_bar(args);
}
```

In the header file:

```
DECLARE_TRACEPOINT(foo_bar);

static inline void some_inline_function()
{
        [..]
        if (tracepoint_enabled(foo_bar))
                do_trace_foo_bar_wrapper(args);
        [..]
}
```

CHAPTER

NINE

EVENT TRACING

Author
Theodore Ts'o

Updated
Li Zefan and Tom Zanussi

9.1 1. Introduction

Tracepoints (see *Using the Linux Kernel Tracepoints*) can be used without creating custom kernel modules to register probe functions using the event tracing infrastructure.

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

9.2 2. Using Event Tracing

9.2.1 2.1 Via the 'set_event' interface

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

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

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

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

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

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

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

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

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

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

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

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

9.2.2 2.2 Via the 'enable' toggle

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

To enable event 'sched_wakeup':

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

To disable it:

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

To enable all events in sched subsystem:

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

To enable all events:

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

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

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

9.2.3 2.3 Boot option

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

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

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

9.3 3. Defining an event-enabled tracepoint

See The example provided in samples/trace_events

9.4 4. Event formats

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

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

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

Each field in the format has the form:

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

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

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

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

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

        field:char comm[TASK_COMM_LEN]; offset:12;      size:16;
```

(continues on next page)

(continued from previous page)

```
        field:pid_t pid;        offset:28;      size:4;
        field:int prio; offset:32;      size:4;
        field:int success;      offset:36;      size:4;
        field:int cpu;  offset:40;      size:4;

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

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

9.5 5. Event filtering

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

9.5.1 5.1 Expression syntax

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

```
field-name relational-operator value
```

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

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

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

The operators available for numeric fields are:

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

And for string fields they are:

==, !=, ~

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

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

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

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

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

9.5.2 5.2 Setting filters

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

For example:

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

A slightly more involved example:

```
# cd /sys/kernel/debug/tracing/events/signal/signal_generate
# echo "((sig >= 10 && sig < 15) || sig == 17) && comm != bash" >␣
↪filter
```

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

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

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

9.5.3 5.2.1 Filter limitations

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

9.5.4 5.3 Clearing filters

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

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

9.5.5 5.3 Subsystem filters

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

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

Clear the filters on all events in the sched subsystem:

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

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

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

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

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

9.5.6 5.4 PID filtering

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

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

Will only trace events for the current task.

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

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

9.6 6. Event triggers

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

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

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

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

The syntax for event triggers is roughly based on the syntax for set_ftrace_filter ‘ftrace filter commands’ (see the ‘Filter commands’ section of *ftrace - Function*

Tracer), but there are major differences and the implementation isn't currently tied to it in any way, so beware about making generalizations between the two.

Note: Writing into trace_marker (See *ftrace - Function Tracer*) can also enable triggers that are written into /sys/kernel/tracing/events/ftrace/print/trigger

9.6.1 6.1 Expression syntax

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

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

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

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

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

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

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

9.6.2 6.2 Supported trigger commands

The following commands are supported:

- enable_event/disable_event

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

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

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

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

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

The format is:

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

To remove the above commands:

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

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

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

- stacktrace

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

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

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

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

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

  The format is:

```
stacktrace[:count]
```

  To remove the above commands:

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

(continues on next page)

(continued from previous page)

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

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

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

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

- snapshot

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

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

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

  To only snapshot once:

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

  To remove the above commands:

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

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

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

- traceon/traceoff

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

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

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

To always disable tracing when nr_rq > 1:

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

To remove the above commands:

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

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

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

- hist

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

  See *Event Histograms* for details and examples.

9.7 7. In-kernel trace event API

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

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

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

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

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

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

9.7.1 7.1 Dyamically creating synthetic event definitions

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

The first creates the event in one step, using synth_event_create(). In this method, the name of the event to create and an array defining the fields is supplied to synth_event_create(). If successful, a synthetic event with that name and fields will exist following that call. For example, to create a new "schedtest" synthetic event:

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

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

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

See synth_field_size() for available types.

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

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

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

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

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

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

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

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

```
struct dynevent_cmd cmd;
char *buf;

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

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

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

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

```
ret = synth_event_gen_cmd_array_start(&cmd, "schedtest", THIS_
↪MODULE,
                                      fields, n_fields);
```

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

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

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

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

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

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

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

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

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

9.7.2 7.2 Tracing synthetic events from in-kernel code

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

9.7.3 7.2.1 Tracing a synthetic event all at once

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

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

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

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

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

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

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

```
u64 vals[7];

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

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

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

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

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

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

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

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

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

To disable it pass false:

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

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

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

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

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

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

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

9.7.4 7.2.2 Tracing a synthetic event piecewise

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

```
struct synth_trace_state trace_state;

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

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

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

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

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

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

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

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

/* cpu */
```

(continues on next page)

(continued from previous page)

```
ret = synth_event_add_next_val(smp_processor_id(), &trace_state);

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

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

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

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

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

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

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

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

9.7.5 7.3 Dyamically creating kprobe and kretprobe event definitions

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

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

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

```
struct dynevent_cmd cmd;
char *buf;

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

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

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

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

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

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

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

or:

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

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

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

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

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

```
gen_kprobe_test = trace_get_event_file(NULL, "kprobes", "gen_kprobe_
↪test");

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

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

```
trace_put_event_file(gen_kprobe_test);

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

9.7.6 7.4 The "dynevent_cmd" low-level API

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

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

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

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

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

buf = kzalloc(MAX_DYNEVENT_CMD_LEN, GFP_KERNEL);

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

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

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

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

```
struct dynevent_arg arg;

dynevent_arg_init(&arg, NULL, 0);

arg.str = name;

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

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

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

```
struct dynevent_arg_pair arg_pair;

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

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

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

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

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

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

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

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

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

CHAPTER

TEN

SUBSYSTEM TRACE POINTS: KMEM

The kmem tracing system captures events related to object and page allocation within the kernel. Broadly speaking there are five major subheadings.

- Slab allocation of small objects of unknown type (kmalloc)
- Slab allocation of small objects of known type
- Page allocation
- Per-CPU Allocator Activity
- External Fragmentation

This document describes what each of the tracepoints is and why they might be useful.

10.1 1. Slab allocation of small objects of unknown type

```
kmalloc               call_site=%lx ptr=%p bytes_req=%zu bytes_
↪alloc=%zu gfp_flags=%s
kmalloc_node  call_site=%lx ptr=%p bytes_req=%zu bytes_alloc=%zu␣
↪gfp_flags=%s node=%d
kfree         call_site=%lx ptr=%p
```

Heavy activity for these events may indicate that a specific cache is justified, particularly if kmalloc slab pages are getting significantly internal fragmented as a result of the allocation pattern. By correlating kmalloc with kfree, it may be possible to identify memory leaks and where the allocation sites were.

10.2 2. Slab allocation of small objects of known type

```
kmem_cache_alloc      call_site=%lx ptr=%p bytes_req=%zu bytes_
↪alloc=%zu gfp_flags=%s
kmem_cache_alloc_node call_site=%lx ptr=%p bytes_req=%zu bytes_
↪alloc=%zu gfp_flags=%s node=%d
kmem_cache_free               call_site=%lx ptr=%p
```

These events are similar in usage to the kmalloc-related events except that it is likely easier to pin the event down to a specific cache. At the time of writing, no information is available on what slab is being allocated from, but the call_site can usually be used to extrapolate that information.

10.3 3. Page allocation

```
mm_page_alloc             page=%p pfn=%lu order=%d migratetype=%d gfp_
↪flags=%s
mm_page_alloc_zone_locked page=%p pfn=%lu order=%u migratetype=%d␣
↪cpu=%d percpu_refill=%d
mm_page_free              page=%p pfn=%lu order=%d
mm_page_free_batched      page=%p pfn=%lu order=%d cold=%d
```

These four events deal with page allocation and freeing. mm_page_alloc is a simple indicator of page allocator activity. Pages may be allocated from the per-CPU allocator (high performance) or the buddy allocator.

If pages are allocated directly from the buddy allocator, the mm_page_alloc_zone_locked event is triggered. This event is important as high amounts of activity imply high activity on the zone->lock. Taking this lock impairs performance by disabling interrupts, dirtying cache lines between CPUs and serialising many CPUs.

When a page is freed directly by the caller, the only mm_page_free event is triggered. Significant amounts of activity here could indicate that the callers should be batching their activities.

When pages are freed in batch, the also mm_page_free_batched is triggered. Broadly speaking, pages are taken off the LRU lock in bulk and freed in batch with a page list. Significant amounts of activity here could indicate that the system is under memory pressure and can also indicate contention on the zone->lru_lock.

10.4 4. Per-CPU Allocator Activity

```
mm_page_alloc_zone_locked    page=%p pfn=%lu order=%u migratetype=
↪%d cpu=%d percpu_refill=%d
mm_page_pcpu_drain           page=%p pfn=%lu order=%d cpu=%d␣
↪migratetype=%d
```

In front of the page allocator is a per-cpu page allocator. It exists only for order-0 pages, reduces contention on the zone->lock and reduces the amount of writing on struct page.

When a per-CPU list is empty or pages of the wrong type are allocated, the zone->lock will be taken once and the per-CPU list refilled. The event triggered is mm_page_alloc_zone_locked for each page allocated with the event indicating whether it is for a percpu_refill or not.

When the per-CPU list is too full, a number of pages are freed, each one which triggers a mm_page_pcpu_drain event.

The individual nature of the events is so that pages can be tracked between allocation and freeing. A number of drain or refill pages that occur consecutively imply the zone->lock being taken once. Large amounts of per-CPU refills and drains could imply an imbalance between CPUs where too much work is being concentrated in one place. It could also indicate that the per-CPU lists should be a larger size. Finally, large amounts of refills on one CPU and drains on another could be a factor in causing large amounts of cache line bounces due to writes between CPUs and worth investigating if pages can be allocated and freed on the same CPU through some algorithm change.

10.5 5. External Fragmentation

```
mm_page_alloc_extfrag         page=%p pfn=%lu alloc_order=%d␣
↪fallback_order=%d pageblock_order=%d alloc_migratetype=%d␣
↪fallback_migratetype=%d fragmenting=%d change_ownership=%d
```

External fragmentation affects whether a high-order allocation will be successful or not. For some types of hardware, this is important although it is avoided where possible. If the system is using huge pages and needs to be able to resize the pool over the lifetime of the system, this value is important.

Large numbers of this event implies that memory is fragmenting and high-order allocations will start failing at some time in the future. One means of reducing the occurrence of this event is to increase the size of min_free_kbytes in increments of 3*pageblock_size*nr_online_nodes where pageblock_size is usually the size of the default hugepage size.

CHAPTER

ELEVEN

SUBSYSTEM TRACE POINTS: POWER

The power tracing system captures events related to power transitions within the kernel. Broadly speaking there are three major subheadings:

- Power state switch which reports events related to suspend (S-states), cpuidle (C-states) and cpufreq (P-states)
- System clock related changes
- Power domains related changes and transitions

This document describes what each of the tracepoints is and why they might be useful.

Cf. include/trace/events/power.h for the events definitions.

11.1 1. Power state switch events

11.1.1 1.1 Trace API

A 'cpu' event class gathers the CPU-related events: cpuidle and cpufreq.

```
cpu_idle                "state=%lu cpu_id=%lu"
cpu_frequency           "state=%lu cpu_id=%lu"
cpu_frequency_limits    "min=%lu max=%lu cpu_id=%lu"
```

A suspend event is used to indicate the system going in and out of the suspend mode:

```
machine_suspend               "state=%lu"
```

Note: the value of '-1' or '4294967295' for state means an exit from the current state, i.e. trace_cpu_idle(4, smp_processor_id()) means that the system enters the idle state 4, while trace_cpu_idle(PWR_EVENT_EXIT, smp_processor_id()) means that the system exits the previous idle state.

The event which has 'state=4294967295' in the trace is very important to the user space tools which are using it to detect the end of the current state, and so to correctly draw the states diagrams and to calculate accurate statistics etc.

11.2 2. Clocks events

The clock events are used for clock enable/disable and for clock rate change.

```
clock_enable            "%s state=%lu cpu_id=%lu"
clock_disable           "%s state=%lu cpu_id=%lu"
clock_set_rate                  "%s state=%lu cpu_id=%lu"
```

The first parameter gives the clock name (e.g. "gpio1_iclk"). The second parameter is '1' for enable, '0' for disable, the target clock rate for set_rate.

11.3 3. Power domains events

The power domain events are used for power domains transitions

```
power_domain_target   "%s state=%lu cpu_id=%lu"
```

The first parameter gives the power domain name (e.g. "mpu_pwrdm"). The second parameter is the power domain target state.

11.4 4. PM QoS events

The PM QoS events are used for QoS add/update/remove request and for target/flags update.

```
pm_qos_update_target                  "action=%s prev_value=%d curr_
↪value=%d"
pm_qos_update_flags                   "action=%s prev_value=0x%x curr_
↪value=0x%x"
```

The first parameter gives the QoS action name (e.g. "ADD_REQ"). The second parameter is the previous QoS value. The third parameter is the current QoS value to update.

There are also events used for device PM QoS add/update/remove request.

```
dev_pm_qos_add_request                "device=%s type=%s new_value=%d"
dev_pm_qos_update_request             "device=%s type=%s new_value=%d"
dev_pm_qos_remove_request             "device=%s type=%s new_value=%d"
```

The first parameter gives the device name which tries to add/update/remove QoS requests. The second parameter gives the request type (e.g. "DEV_PM_QOS_RESUME_LATENCY"). The third parameter is value to be added/updated/removed.

And, there are events used for CPU latency QoS add/update/remove request.

```
pm_qos_add_request         "value=%d"
pm_qos_update_request      "value=%d"
pm_qos_remove_request      "value=%d"
```

The parameter is the value to be added/updated/removed.

CHAPTER

TWELVE

NMI TRACE EVENTS

These events normally show up here:

/sys/kernel/debug/tracing/events/nmi

12.1 nmi_handler

You might want to use this tracepoint if you suspect that your NMI handlers are hogging large amounts of CPU time. The kernel will warn if it sees long-running handlers:

```
INFO: NMI handler took too long to run: 9.207 msecs
```

and this tracepoint will allow you to drill down and get some more details.

Let's say you suspect that perf_event_nmi_handler() is causing you some problems and you only want to trace that handler specifically. You need to find its address:

```
$ grep perf_event_nmi_handler /proc/kallsyms
ffffffff81625600 t perf_event_nmi_handler
```

Let's also say you are only interested in when that function is really hogging a lot of CPU time, like a millisecond at a time. Note that the kernel's output is in milliseconds, but the input to the filter is in nanoseconds! You can filter on 'delta_ns':

```
cd /sys/kernel/debug/tracing/events/nmi/nmi_handler
echo 'handler==0xffffffff81625600 && delta_ns>1000000' > filter
echo 1 > enable
```

Your output would then look like:

```
$ cat /sys/kernel/debug/tracing/trace_pipe
<idle>-0     [000] d.h3   505.397558: nmi_handler: perf_event_nmi_
↪handler() delta_ns: 3236765 handled: 1
<idle>-0     [000] d.h3   505.805893: nmi_handler: perf_event_nmi_
↪handler() delta_ns: 3174234 handled: 1
<idle>-0     [000] d.h3   506.158206: nmi_handler: perf_event_nmi_
↪handler() delta_ns: 3084642 handled: 1
<idle>-0     [000] d.h3   506.334346: nmi_handler: perf_event_nmi_
↪handler() delta_ns: 3080351 handled: 1
```

CHAPTER

THIRTEEN

MSR TRACE EVENTS

The x86 kernel supports tracing most MSR (Model Specific Register) accesses. To see the definition of the MSRs on Intel systems please see the SDM at https://www.intel.com/sdm (Volume 3)

Available trace points:

/sys/kernel/debug/tracing/events/msr/

Trace MSR reads:

read_msr

- msr: MSR number
- val: Value written
- failed: 1 if the access failed, otherwise 0

Trace MSR writes:

write_msr

- msr: MSR number
- val: Value written
- failed: 1 if the access failed, otherwise 0

Trace RDPMC in kernel:

rdpmc

The trace data can be post processed with the postprocess/decode_msr.py script:

```
cat /sys/kernel/debug/tracing/trace | decode_msr.py /usr/src/linux/
↪include/asm/msr-index.h
```

to add symbolic MSR names.

CHAPTER

FOURTEEN

IN-KERNEL MEMORY-MAPPED I/O TRACING

Home page and links to optional user space tools:

> https://nouveau.freedesktop.org/wiki/MmioTrace

MMIO tracing was originally developed by Intel around 2003 for their Fault Injection Test Harness. In Dec 2006 - Jan 2007, using the code from Intel, Jeff Muizelaar created a tool for tracing MMIO accesses with the Nouveau project in mind. Since then many people have contributed.

Mmiotrace was built for reverse engineering any memory-mapped IO device with the Nouveau project as the first real user. Only x86 and x86_64 architectures are supported.

Out-of-tree mmiotrace was originally modified for mainline inclusion and ftrace framework by Pekka Paalanen <pq@iki.fi>.

14.1 Preparation

Mmiotrace feature is compiled in by the CONFIG_MMIOTRACE option. Tracing is disabled by default, so it is safe to have this set to yes. SMP systems are supported, but tracing is unreliable and may miss events if more than one CPU is on-line, therefore mmiotrace takes all but one CPU off-line during run-time activation. You can re-enable CPUs by hand, but you have been warned, there is no way to automatically detect if you are losing events due to CPUs racing.

14.2 Usage Quick Reference

```
$ mount -t debugfs debugfs /sys/kernel/debug
$ echo mmiotrace > /sys/kernel/debug/tracing/current_tracer
$ cat /sys/kernel/debug/tracing/trace_pipe > mydump.txt &
Start X or whatever.
$ echo "X is up" > /sys/kernel/debug/tracing/trace_marker
$ echo nop > /sys/kernel/debug/tracing/current_tracer
Check for lost events.
```

14.3 Usage

Make sure debugfs is mounted to /sys/kernel/debug. If not (requires root privileges):

```
$ mount -t debugfs debugfs /sys/kernel/debug
```

Check that the driver you are about to trace is not loaded.

Activate mmiotrace (requires root privileges):

```
$ echo mmiotrace > /sys/kernel/debug/tracing/current_tracer
```

Start storing the trace:

```
$ cat /sys/kernel/debug/tracing/trace_pipe > mydump.txt &
```

The ‘cat’ process should stay running (sleeping) in the background.

Load the driver you want to trace and use it. Mmiotrace will only catch MMIO accesses to areas that are ioremapped while mmiotrace is active.

During tracing you can place comments (markers) into the trace by $ echo “X is up” > /sys/kernel/debug/tracing/trace_marker This makes it easier to see which part of the (huge) trace corresponds to which action. It is recommended to place descriptive markers about what you do.

Shut down mmiotrace (requires root privileges):

```
$ echo nop > /sys/kernel/debug/tracing/current_tracer
```

The ‘cat’ process exits. If it does not, kill it by issuing ‘fg’ command and pressing ctrl+c.

Check that mmiotrace did not lose events due to a buffer filling up. Either:

```
$ grep -i lost mydump.txt
```

which tells you exactly how many events were lost, or use:

```
$ dmesg
```

to view your kernel log and look for “mmiotrace has lost events” warning. If events were lost, the trace is incomplete. You should enlarge the buffers and try again. Buffers are enlarged by first seeing how large the current buffers are:

```
$ cat /sys/kernel/debug/tracing/buffer_size_kb
```

gives you a number. Approximately double this number and write it back, for instance:

```
$ echo 128000 > /sys/kernel/debug/tracing/buffer_size_kb
```

Then start again from the top.

If you are doing a trace for a driver project, e.g. Nouveau, you should also do the following before sending your results:

```
$ lspci -vvv > lspci.txt
$ dmesg > dmesg.txt
$ tar zcf pciid-nick-mmiotrace.tar.gz mydump.txt lspci.txt dmesg.txt
```

and then send the .tar.gz file. The trace compresses considerably. Replace "pciid" and "nick" with the PCI ID or model name of your piece of hardware under investigation and your nickname.

14.4 How Mmiotrace Works

Access to hardware IO-memory is gained by mapping addresses from PCI bus by calling one of the ioremap_*() functions. Mmiotrace is hooked into the __ioremap() function and gets called whenever a mapping is created. Mapping is an event that is recorded into the trace log. Note that ISA range mappings are not caught, since the mapping always exists and is returned directly.

MMIO accesses are recorded via page faults. Just before __ioremap() returns, the mapped pages are marked as not present. Any access to the pages causes a fault. The page fault handler calls mmiotrace to handle the fault. Mmiotrace marks the page present, sets TF flag to achieve single stepping and exits the fault handler. The instruction that faulted is executed and debug trap is entered. Here mmiotrace again marks the page as not present. The instruction is decoded to get the type of operation (read/write), data width and the value read or written. These are stored to the trace log.

Setting the page present in the page fault handler has a race condition on SMP machines. During the single stepping other CPUs may run freely on that page and events can be missed without a notice. Re-enabling other CPUs during tracing is discouraged.

14.5 Trace Log Format

The raw log is text and easily filtered with e.g. grep and awk. One record is one line in the log. A record starts with a keyword, followed by keyword- dependent arguments. Arguments are separated by a space, or continue until the end of line. The format for version 20070824 is as follows:

14.6 Explanation Keyword Space-separated arguments

read event R width, timestamp, map id, physical, value, PC, PID write event W width, timestamp, map id, physical, value, PC, PID ioremap event MAP timestamp, map id, physical, virtual, length, PC, PID iounmap event UNMAP timestamp, map id, PC, PID marker MARK timestamp, text version VERSION the string "20070824" info for reader LSPCI one line from lspci -v PCI address map PCIDEV space-separated /proc/bus/pci/devices data unk. opcode UNKNOWN timestamp, map id, physical, data, PC, PID

Timestamp is in seconds with decimals. Physical is a PCI bus address, virtual is a kernel virtual address. Width is the data width in bytes and value is the data value. Map id is an arbitrary id number identifying the mapping that was used in an operation. PC is the program counter and PID is process id. PC is zero if it is not recorded. PID is always zero as tracing MMIO accesses originating in user space memory is not yet supported.

For instance, the following awk filter will pass all 32-bit writes that target physical addresses in the range [0xfb73ce40, 0xfb800000]

```
$ awk '/W 4 / { adr=strtonum($5); if (adr >= 0xfb73ce40 &&
adr < 0xfb800000) print; }'
```

14.7 Tools for Developers

The user space tools include utilities for:

- replacing numeric addresses and values with hardware register names
- replaying MMIO logs, i.e., re-executing the recorded writes

CHAPTER

FIFTEEN

EVENT HISTOGRAMS

Documentation written by Tom Zanussi

15.1 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 *Event Tracing*.

15.2 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 three 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
.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.

15.2.1 '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:

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

15.2.2 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 `Documentation/trace/ftrace.rst` for details.

15.2.3 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' > \
        /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]
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
# 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_site: 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_
```

(continues on next page)

(continued from previous page)

```
↪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
```

(continues on next page)

(continued from previous page)

```
{ 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] ␣
```

(continues on next page)

(continued from previous page)

```
↪                    } 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                    ␣
```

(continues on next page)

(continued from previous page)

```
↪                  } 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:     ␣
```

(continues on next page)

(continued from previous page)

```
↪    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_
```

(continues on next page)

(continued from previous page)

```
↪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,
```

(continues on next page)

(continued from previous page)

```
↪bytes_alloc:sort=bytes_alloc.descending:size=2048 [active]

{ call_site: [ffffffffa046041c] i915_gem_execbuffer2 [i915]␣
↪                  } hitcount:       7403  bytes_req:   ␣
↪4084360  bytes_alloc:    5958016
{ call_site: [ffffffff811e2a1b] seq_buf_alloc              ␣
↪                  } hitcount:        541  bytes_req:   ␣
↪2213968  bytes_alloc:    2228224
{ call_site: [ffffffffa0489a66] intel_ring_begin [i915]    ␣
↪                  } hitcount:       7404  bytes_req:   ␣
↪1066176  bytes_alloc:    1421568
{ call_site: [ffffffffa045e7c4] i915_gem_do_execbuffer.isra.
↪23 [i915]         } hitcount:       1565  bytes_req:    ␣
↪557368  bytes_alloc:    1037760
{ call_site: [ffffffff8125847d] ext4_htree_store_dirent    ␣
↪                  } hitcount:       9557  bytes_req:    ␣
↪595778  bytes_alloc:     695744
{ call_site: [ffffffffa045e646] i915_gem_do_execbuffer.isra.
↪23 [i915]         } hitcount:       5839  bytes_req:    ␣
↪430680  bytes_alloc:     470400
{ call_site: [ffffffffa04c4a3c] intel_plane_duplicate_state␣
↪[i915]            } hitcount:       2388  bytes_req:    ␣
↪324768  bytes_alloc:     458496
{ call_site: [ffffffffa02911f2] drm_modeset_lock_crtc [drm]␣
↪                  } hitcount:       3911  bytes_req:    ␣
↪219016  bytes_alloc:     250304
{ call_site: [ffffffff815f8d7b] sk_prot_alloc              ␣
↪                  } hitcount:        235  bytes_req:    ␣
↪236880  bytes_alloc:     240640
{ call_site: [ffffffff8137e559] sg_kmalloc                 ␣
↪                  } hitcount:        557  bytes_req:    ␣
↪169024  bytes_alloc:     221760
{ call_site: [ffffffffa00b7c06] hid_report_raw_event [hid] ␣
↪                  } hitcount:       9378  bytes_req:    ␣
↪187548  bytes_alloc:     206312
{ call_site: [ffffffffa04a580c] intel_crtc_page_flip [i915]␣
↪                  } hitcount:       1519  bytes_req:    ␣
↪157976  bytes_alloc:     194432
.
.
.
{ call_site: [ffffffff8109bd3b] sched_autogroup_create_
↪attach                     } hitcount:          2  bytes_req:␣
↪       144  bytes_alloc:        192
{ call_site: [ffffffff81097ee8] alloc_rt_sched_group       ␣
↪                  } hitcount:          2  bytes_req:      ␣
↪  128  bytes_alloc:        128
{ call_site: [ffffffff8109524a] alloc_fair_sched_group     ␣
↪                  } hitcount:          2  bytes_req:      ␣
```

(continues on next page)

(continued from previous page)

```
↪  128  bytes_alloc:        128
{ call_site: [ffffffff81095225] alloc_fair_sched_group                 ␣
↪                   } hitcount:          2  bytes_req:                  ␣
↪  128  bytes_alloc:        128
{ call_site: [ffffffff81097ec2] alloc_rt_sched_group                   ␣
↪                   } hitcount:          2  bytes_req:                  ␣
↪  128  bytes_alloc:        128
{ call_site: [ffffffff81213e80] load_elf_binary                        ␣
↪                   } hitcount:          3  bytes_req:                  ␣
↪   84  bytes_alloc:         96
{ call_site: [ffffffff81079a2e] kthread_create_on_node                 ␣
↪                   } hitcount:          1  bytes_req:                  ␣
↪   56  bytes_alloc:         64
{ call_site: [ffffffffa00bf6fe] hidraw_send_report [hid]               ␣
↪                   } hitcount:          1  bytes_req:                  ␣
↪    7  bytes_alloc:          8
{ call_site: [ffffffff8154bc62] usb_control_msg                        ␣
↪                   } hitcount:          1  bytes_req:                  ␣
↪    8  bytes_alloc:          8
{ call_site: [ffffffffa00bf1ca] hidraw_report_event [hid]              ␣
↪                   } hitcount:          1  bytes_req:                  ␣
↪    7  bytes_alloc:          8

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]
```

(continues on next page)

(continued from previous page)

```
{ 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:
```

(continues on next page)

(continued from previous page)

```
     __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
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
{ 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:␣
```

(continues on next page)

(continued from previous page)

```
↪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:␣
```

(continues on next page)

(continued from previous page)

```
↪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_
```

(continues on next page)

(continued from previous page)

```
↪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: ␣
```

(continues on next page)

(continued from previous page)

```
↪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                            }␣
```

(continues on next page)

(continued from previous page)

```
↪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]
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
{ 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' >␣
↪\
```

(continues on next page)

(continued from previous page)

```
      /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]
```

(continues on next page)

(continued from previous page)

```
     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
.
.
.
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
.
.
.
{ 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
```

(continues on next page)

(continued from previous page)

```
{ 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' > \
```

(continues on next page)

(continued from previous page)

```
      /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:   ␣
```

(continues on next page)

(continued from previous page)

```
↪     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
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
{ 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:
```

(continues on next page)

(continued from previous page)

```
↪      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
```

(continues on next page)

(continued from previous page)

```
         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
```

(continues on next page)

(continued from previous page)

```
         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
```

15.2.4 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.

15.2.5 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
```

15.2.6 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.

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

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

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
```

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

15.2.7 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
```

(continues on next page)

(continued from previous page)

```
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/trigg

  # 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_comm=swapper/5 next_pid=0 next_prio=120
    <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=cyclictest next_pid=2102 next_prio=19
    <...>-2102  [005] d..3   309.873625: sched_switch: prev_
↪comm=cyclictest prev_pid=2102 prev_prio=19 prev_state=D ==>␣
↪next_comm=swapper/5 next_pid=0 next_prio=120
    <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=cyclictest next_pid=2102 next_prio=19
    <idle>-0     [004] d.h3   309.874757: sched_waking:␣
```

(continues on next page)

(continued from previous page)

```
↪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- next_pid=1699 next_prio=120
gnome-terminal--1699  [004] d.h2   309.874941: sched_stat_
↪runtime: comm=gnome-terminal- pid=1699 runtime=180706 [ns]␣
↪vruntime=1126870572 [ns]
   <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 next_pid=9 next_prio=120
 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=swapper/7 next_pid=0 next_prio=120
    <...>-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=cyclictest next_pid=2103 next_prio=19
```

- 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=1835730389 [ns]
 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+ ==> next_comm=kworker/3:2 next_pid=135 next_prio=120
   gnome-shell-1261  [006] d..3    49.823114: sched_switch:␣
↪prev_comm=gnome-shell prev_pid=1261 prev_prio=120 prev_
↪state=R+ ==> next_comm=kworker/6:2 next_pid=387 next_prio=120
   kworker/3:2-135   [003] d..3    49.823118: sched_stat_
↪runtime: comm=kworker/3:2 pid=135 runtime=5339 [ns]␣
↪vruntime=17815800388 [ns]
   kworker/6:2-387   [006] d..3    49.823120: sched_stat_
↪runtime: comm=kworker/6:2 pid=387 runtime=9594 [ns]␣
↪vruntime=14589605367 [ns]
   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+ ==> next_comm=gnome-shell next_pid=1261 next_prio=120
   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␣
↪==> next_comm=swapper/3 next_pid=0 next_prio=120
        <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=0xe3ae2ff5 snd_una=0xe3ae2ecd snd_cwnd=10␣
↪ssthresh=2147483647 snd_wnd=28960 srtt=19604 rcv_wnd=29312
```

15.2.8 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;
```

(continues on next page)

(continued from previous page)

```
        /* 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/trigger
# 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_
```

(continues on next page)

(continued from previous page)

```
↪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
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
{ 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/print/trigger
# 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
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
{ 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
```

(continues on next page)

(continued from previous page)

```
{ 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.

CHAPTER

SIXTEEN

HISTOGRAM DESIGN NOTES

Author

Tom Zanussi <zanussi@kernel.org>

This document attempts to provide a description of how the ftrace histograms work and how the individual pieces map to the data structures used to implement them in trace_events_hist.c and tracing_map.c.

Note: All the ftrace histogram command examples assume the working directory is the ftrace /tracing directory. For example:

```
# cd /sys/kernel/debug/tracing
```

Also, the histogram output displayed for those commands will be generally be truncated - only enough to make the point is displayed.

16.1 'hist_debug' trace event files

If the kernel is compiled with CONFIG_HIST_TRIGGERS_DEBUG set, an event file named 'hist_debug' will appear in each event's subdirectory. This file can be read at any time and will display some of the hist trigger internals described in this document. Specific examples and output will be described in test cases below.

16.2 Basic histograms

First, basic histograms. Below is pretty much the simplest thing you can do with histograms - create one with a single key on a single event and cat the output:

```
# echo 'hist:keys=pid' >> events/sched/sched_waking/trigger

# cat events/sched/sched_waking/hist

{ pid:      18249 } hitcount:          1
{ pid:      13399 } hitcount:          1
{ pid:      17973 } hitcount:          1
{ pid:      12572 } hitcount:          1
...
{ pid:         10 } hitcount:        921
```

(continues on next page)

(continued from previous page)

```
{ pid:      18255 } hitcount:       1444
{ pid:      25526 } hitcount:       2055
{ pid:       5257 } hitcount:       2055
{ pid:      27367 } hitcount:       2055
{ pid:       1728 } hitcount:       2161

Totals:
  Hits: 21305
  Entries: 183
  Dropped: 0
```

What this does is create a histogram on the sched_waking event using pid as a key and with a single value, hitcount, which even if not explicitly specified, exists for every histogram regardless.

The hitcount value is a per-bucket value that's automatically incremented on every hit for the given key, which in this case is the pid.

So in this histogram, there's a separate bucket for each pid, and each bucket contains a value for that bucket, counting the number of times sched_waking was called for that pid.

Each histogram is represented by a hist_data struct.

To keep track of each key and value field in the histogram, hist_data keeps an array of these fields named fields[]. The fields[] array is an array containing struct hist_field representations of each histogram val and key in the histogram (variables are also included here, but are discussed later). So for the above histogram we have one key and one value; in this case the one value is the hitcount value, which all histograms have, regardless of whether they define that value or not, which the above histogram does not.

Each struct hist_field contains a pointer to the ftrace_event_field from the event's trace_event_file along with various bits related to that such as the size, offset, type, and a hist_field_fn_t function, which is used to grab the field's data from the ftrace event buffer (in most cases - some hist_fields such as hitcount don't directly map to an event field in the trace buffer - in these cases the function implementation gets its value from somewhere else). The flags field indicates which type of field it is - key, value, variable, variable reference, etc., with value being the default.

The other important hist_data data structure in addition to the fields[] array is the tracing_map instance created for the histogram, which is held in the .map member. The tracing_map implements the lock-free hash table used to implement histograms (see kernel/trace/tracing_map.h for much more discussion about the low-level data structures implementing the tracing_map). For the purposes of this discussion, the tracing_map contains a number of buckets, each bucket corresponding to a particular tracing_map_elt object hashed by a given histogram key.

Below is a diagram the first part of which describes the hist_data and associated key and value fields for the histogram described above. As you can see, there are two fields in the fields array, one val field for the hitcount and one key field for the pid key.

Below that is a diagram of a run-time snapshot of what the tracing_map might look like for a given run. It attempts to show the relationships between the hist_data fields and the tracing_map elements for a couple hypothetical keys and values.:

```
+------------------+
| hist_data        |
+------------------+     +----------------+
  | .fields[]      |---->| val = hitcount |------------------------
↪---+
  +----------------+     +----------------+                         ␣
↪   |
  | .map           |       | .size        |                         ␣
↪   |
  +----------------+       +--------------+                         ␣
↪   |
                           | .offset      |                         ␣
↪   |
                           +--------------+                         ␣
↪   |
                           | .fn()        |                         ␣
↪   |
                           +--------------+                         ␣
↪   |
                                 .                                  ␣
↪   |
                                 .                                  ␣
↪   |
                                 .                                  ␣
↪   |
                         +----------------+ <--- n_vals             ␣
↪   |
                         | key = pid      |------------------------
↪---|--+
                         +----------------+                         ␣
↪   |  |
                           | .size        |                         ␣
↪   |  |
                           +--------------+                         ␣
↪   |  |
                           | .offset      |                         ␣
↪   |  |
                           +--------------+                         ␣
↪   |  |
                           | .fn()        |                         ␣
↪   |  |
                         +----------------+ <--- n_fields           ␣
↪   |  |
                         | unused         |                         ␣
↪   |  |
                         +----------------+                         ␣
↪   |  |
```

(continues on next page)

(continued from previous page)

```
                            |              |                        ␣
↪   |  |
                            +--------------+                        ␣
↪   |  |
                            |              |                        ␣
↪   |  |
                            +--------------+                        ␣
↪   |  |
                            |              |                        ␣
↪   |  |
                            +--------------+                        ␣
↪   |  |
                                          n_keys = n_fields - n_
↪vals   |  |
```

The hist_data n_vals and n_fields delineate the extent of the fields[] | | array and separate keys from values for the rest of the code. | |

Below is a run-time representation of the tracing_map part of the | | histogram, with pointers from various parts of the fields[] array | | to corresponding parts of the tracing_map. | |

The tracing_map consists of an array of tracing_map_entrys and a set | | of preallocated tracing_map_elts (abbreviated below as map_entry and | | map_elt). The total number of map_entrys in the hist_data.map array = | | map->max_elts (actually map->map_size but only max_elts of those are | | used. This is a property required by the map_insert() algorithm). | |

If a map_entry is unused, meaning no key has yet hashed into it, its | | .key value is 0 and its .val pointer is NULL. Once a map_entry 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:

```
+-----------+                                                       ␣
↪   |  |
| hist_data |                                                       ␣
↪   |  |
+-----------+                                                       ␣
↪   |  |
  | .fields |                                                       ␣
↪   |  |
  +---------+     +-----------+                                     ␣
↪   |  |
  | .map    |---->| map_entry |                                     ␣
↪   |  |
```

(continues on next page)

(continued from previous page)

```
+---------+     +-----------+
↪   |  |
                  | .key    |---> 0
↪   |  |
                  +---------+
↪   |  |
                  | .val    |---> NULL
↪   |  |
                +-----------+
↪   |  |
                | map_entry |
↪   |  |
                +-----------+
↪   |  |
                  | .key    |---> pid = 999
↪   |  |
                  +---------+    +-----------+
↪   |  |
                  | .val    |--->| map_elt   |
↪   |  |
                  +---------+    +-----------+
↪   |  |
                       .           | .key    |---> full key *
↪   |  |
                       .           +---------+    +--------------
↪+  |  |
                       .           | .fields |--->| .sum (val)
↪|<-+  |
                +-----------+      +---------+    | 2345
↪|  |  |
                | map_entry |                     +--------------
↪+  |  |
                +-----------+                     | .offset (key)
↪|<----+
                  | .key    |---> 0              | 0
↪|  |  |
                  +---------+                     +--------------
↪+  |  |
                  | .val    |---> NULL                    .
↪   |  |
                +-----------+                             .
↪   |  |
                | map_entry |                             .
↪   |  |
                +-----------+                     +--------------
↪+  |  |
                  | .key    |                     | .sum (val) or
↪|  |  |
                  +---------+    +---------+      | .offset (key)
```

(continues on next page)

(continued from previous page)

```
↪|   |   |
                  | .val     |--->| map_elt |       +---------------
↪+   |   |
                +-----------+    +---------+       | .sum (val) or␣
↪|   |   |
                | map_entry |                      | .offset (key)␣
↪|   |   |
                +-----------+                      +---------------
↪+   |   |
                  | .key     |---> pid = 4444                      ␣
↪    |   |
                  +---------+    +-----------+                     ␣
↪    |   |
                  | .val     |    | map_elt   |                    ␣
↪    |   |
                  +---------+    +-----------+                     ␣
↪    |   |
                                  | .key     |---> full key *      ␣
↪    |   |
                                  +---------+    +---------------
↪+   |   |
                                  | .fields |--->| .sum (val)      ␣
↪|<-+   |
                                  +---------+    | 65523           ␣
↪|       |
                                                 +---------------
↪+       |
                                                 | .offset (key)␣
↪|<----+
                                                 | 0               ␣
↪|
                                                 +---------------
↪+
                                                         .
                                                         .
                                                         .
                                                 +---------------
↪+
                                                 | .sum (val) or␣
↪|
                                                 | .offset (key)␣
↪|
                                                 +---------------
↪+
                                                 | .sum (val) or␣
↪|
                                                 | .offset (key)␣
↪|
                                                 +---------------
```

(continues on next page)

(continued from previous page)

```
↪+
```

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.

16.2.1 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: 000000005e48c9a5

n_vals: 3
n_keys: 2
n_fields: 5

val fields:

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

  hist_data->fields[1]:
    flags:
      VAL: normal u64 value
    ftrace_event_field name: bytes_req
    type: size_t
    size: 8
    is_signed: 0

  hist_data->fields[2]:
    flags:
      VAL: normal u64 value
    ftrace_event_field name: bytes_alloc
    type: size_t
    size: 8
    is_signed: 0

key fields:
```

(continues on next page)

(continued from previous page)

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

  hist_data->fields[4]:
    flags:
      HIST_FIELD_FL_KEY
    ftrace_event_field name: call_site
    type: unsigned long
    size: 8
    is_signed: 0
```

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

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

16.3 Variables

Variables allow data from one hist trigger to be saved by one hist trigger and retrieved by another hist trigger. For example, a trigger on the sched_waking event can capture a timestamp for a particular pid, and later a sched_switch event that switches to that pid event can grab the timestamp and use it to calculate a time delta between the two events:

```
# 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
```

In terms of the histogram data structures, variables are implemented as another type of hist_field and for a given hist trigger are added to the hist_data.fields[] array just after all the val fields. To distinguish them from the existing key and val fields, they' re given a new flag type, HIST_FIELD_FL_VAR (abbreviated FL_VAR) and they also make use of a new .var.idx field member in struct hist_field, which maps them to an index in a new map_elt.vars[] array added to the map_elt specifically designed to store and retrieve variable values. The diagram below shows those new elements and adds a new variable entry, ts0, corresponding to the ts0 variable in the sched_waking trigger above.

sched_waking histogram ——————-:

```
+------------------+
| hist_data        |<---------------------------------------------
↪--------+
+------------------+   +-------------------+                       ␣
↪        |
  | .fields[]      |-->| val = hitcount    |                       ␣
↪        |
  +----------------+   +-------------------+                       ␣
↪        |
  | .map           |     | .size           |                       ␣
↪        |
  +----------------+     +-----------------+                       ␣
↪        |
                         | .offset         |                       ␣
↪        |
                         +-----------------+                       ␣
↪        |
                         | .fn()           |                       ␣
↪        |
                         +-----------------+                       ␣
↪        |
                         | .flags          |                       ␣
↪        |
                         +-----------------+                       ␣
↪        |
                         | .var.idx        |                       ␣
↪        |
                       +-------------------+                       ␣
↪        |
                       | var = ts0         |                       ␣
↪        |
                       +-------------------+                       ␣
↪        |
                         | .size           |                       ␣
↪        |
                         +-----------------+                       ␣
↪        |
                         | .offset         |                       ␣
↪        |
                         +-----------------+                       ␣
↪        |
                         | .fn()           |                       ␣
↪        |
                         +-----------------+                       ␣
↪        |
                         | .flags & FL_VAR |                       ␣
↪        |
                         +-----------------+                       ␣
↪        |
                         | .var.idx        |-----------------------
```

(continues on next page)

(continued from previous page)

```
↪----+-+ |                  +----------------+                            ␣
↪    | | |                           .                                   ␣
↪    | | |                           .                                   ␣
↪    | | |                           .                                   ␣
↪    | | |                +------------------+ <--- n_vals               ␣
↪    | | |                | key = pid        |                           ␣
↪    | | |                +------------------+                           ␣
↪    | | |                  | .size          |                           ␣
↪    | | |                  +----------------+                           ␣
↪    | | |                  | .offset        |                           ␣
↪    | | |                  +----------------+                           ␣
↪    | | |                  | .fn()          |                           ␣
↪    | | |                  +----------------+                           ␣
↪    | | |                  | .flags & FL_KEY |                          ␣
↪    | | |                  +----------------+                           ␣
↪    | | |                  | .var.idx       |                           ␣
↪    | | |                +------------------+ <--- n_fields             ␣
↪    | | |                | unused           |                           ␣
↪    | | |                +------------------+                           ␣
↪    | | |                  |                |                           ␣
↪    | | |                  +----------------+                           ␣
↪    | | |                  |                |                           ␣
↪    | | |                  +----------------+                           ␣
↪    | | |                  |                |                           ␣
↪    | | |
```

(continues on next page)

(continued from previous page)

```
                        +-----------------+                          ␣
↪     | | |
                        |                 |                          ␣
↪     | | |
                        +-----------------+                          ␣
↪     | | |
                        |                 |                          ␣
↪     | | |
                        +-----------------+                          ␣
↪     | | |
                                           n_keys = n_fields - n_
↪vals    | | |
                                                                     ␣
↪     | | |
```

This is very similar to the basic case. In the above diagram, we can | | | see a new .flags member has been added to the struct hist_field | | | struct, and a new entry added to hist_data.fields representing the ts0 | | | variable. For a normal val hist_field, .flags is just 0 (modulo | | | modifier flags), but if the value is defined as a variable, the .flags | | | contains a set FL_VAR bit. | | |

As you can see, the ts0 entry's .var.idx member contains the index | | | into the tracing_map_elts' .vars[] array containing variable values. | | | This idx is used whenever the value of the variable is set or read. | | | The map_elt.vars idx assigned to the given variable is assigned and | | | saved in .var.idx by create_tracing_map_fields() after it calls | | | tracing_map_add_var(). | | |

Below is a representation of the histogram at run-time, which | | | populates the map, along with correspondence to the above hist_data and | | | hist_field data structures. | | |

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.:

```
+-----------+                                                        ␣
↪     | | |
| hist_data |                                                        ␣
↪     | | |
+-----------+                                                        ␣
↪     | | |
  | .fields |                                                        ␣
↪     | | |
  +---------+     +-----------+                                      ␣
↪     | | |
  | .map    |---->| map_entry |                                      ␣
↪     | | |
```

(continues on next page)

(continued from previous page)

```
 +---------+     +-----------+                                                    ␣
↪    | | |
                   | .key    |---> 0                                              ␣
↪    | | |
                   +---------+                                                    ␣
↪    | | |
                   | .val    |---> NULL                                           ␣
↪    | | |
                 +-----------+                                                    ␣
↪    | | |
                 | map_entry |                                                    ␣
↪    | | |
                 +-----------+                                                    ␣
↪    | | |
                   | .key    |---> pid = 999                                      ␣
↪    | | |
                   +---------+    +-----------+                                   ␣
↪    | | |
                   | .val    |--->| map_elt   |                                   ␣
↪    | | |
                   +---------+    +-----------+                                   ␣
↪    | | |
                        .           | .key    |---> full key *                    ␣
↪    | | |
                        .           +---------+    +---------------
↪+   | | |
                        .           | .fields |--->| .sum (val)                   ␣
↪|   | | |
                        .           +---------+    | 2345                         ␣
↪|   | | |
                        .        +--| .vars   |    +---------------
↪+   | | |
                        .        |  +---------+    | .offset (key)␣
↪|   | | |
                        .        |                 | 0                            ␣
↪|   | | |
                        .        |                 +---------------
↪+   | | |
                        .        |                        .                       ␣
↪    | | |
                        .        |                        .                       ␣
↪    | | |
                        .        |                        .                       ␣
↪    | | |
                        .        |                 +---------------
↪+   | | |
                        .        |                 | .sum (val) or␣
↪|   | | |
                        .        |                 | .offset (key)␣
```

(continues on next page)

(continued from previous page)

```
↪|   | | |
                  .       |               +--------------
↪+   | | |
                  .       |               | .sum (val) or␣
↪|   | | |
                  .       |               | .offset (key)␣
↪|   | | |
                  .       |               +--------------
↪+   | | |
                  .       |                             ␣
↪    | | |
                  .       +--------------->+--------------
↪+   | | |
                  .                       | ts0         ␣
↪|<--+ | |
                  .                       | 113345679876 ␣
↪|   | | |
                  .                       +--------------
↪+   | | |
                  .                       | unused      ␣
↪|   | | |
                  .                       |             ␣
↪|   | | |
                  .                       +--------------
↪+   | | |
                  .                               .     ␣
↪    | | |
                  .                               .     ␣
↪    | | |
                  .                               .     ␣
↪    | | |
                  .                       +--------------
↪+   | | |
                  .                       | unused      ␣
↪|   | | |
                  .                       |             ␣
↪|   | | |
                  .                       +--------------
↪+   | | |
                  .                       | unused      ␣
↪|   | | |
                  .                       |             ␣
↪|   | | |
                  .                       +--------------
↪+   | | |
                  .                                     ␣
↪    | | |
              +-----------+                             ␣
↪    | | |
```

(continues on next page)

(continued from previous page)

```
                  | map_entry |
↪     | | |
                  +-----------+
↪     | | |
                    | .key    |---> pid = 4444
↪     | | |
                    +---------+    +-----------+
↪     | | |
                    | .val    |--->| map_elt   |
↪     | | |
                    +---------+    +-----------+
↪     | | |
                         .           | .key    |---> full key *
↪     | | |
                         .           +---------+    +---------------
↪+    | | |
                         .           | .fields |--->| .sum (val)
↪|    | | |
                                     +---------+    | 2345
↪|    | | |
                                  +--| .vars   |    +---------------
↪+    | | |
                                  |  +---------+    | .offset (key)
↪|    | | |
                                  |                 | 0
↪|    | | |
                                  |                 +---------------
↪+    | | |
                                  |                        .
↪     | | |
                                  |                        .
↪     | | |
                                  |                        .
↪     | | |
                                  |                 +---------------
↪+    | | |
                                  |                 | .sum (val) or
↪|    | | |
                                  |                 | .offset (key)
↪|    | | |
                                  |                 +---------------
↪+    | | |
                                  |                 | .sum (val) or
↪|    | | |
                                  |                 | .offset (key)
↪|    | | |
                                  |                 +---------------
↪+    | | |
                                  |
```

(continues on next page)

(continued from previous page)

```
↪     | | |
                                             |                 +--------------
↪+    | | |
                                             +----------------->| ts0          ␣
↪|<--+ | |
                                                                | 213499240729 ␣
↪|      | |
                                                               +--------------
↪+      | |
                                                                | unused       ␣
↪|      | |
                                                                |              ␣
↪|      | |
                                                               +--------------
↪+      | |
                                                                        .      ␣
↪       | |
                                                                        .      ␣
↪       | |
                                                                        .      ␣
↪       | |
                                                               +--------------
↪+      | |
                                                                | unused       ␣
↪|      | |
                                                                |              ␣
↪|      | |
                                                               +--------------
↪+      | |
                                                                | unused       ␣
↪|      | |
                                                                |              ␣
↪|      | |
                                                               +--------------
↪+      | |
```

For each used map entry, there’ s a map_elt pointing to an array of | | .vars containing the current value of the variables associated with | | that histogram entry. So in the above, the timestamp associated with | | pid 999 is 113345679876, and the timestamp variable in the same | | .var.idx for pid 4444 is 213499240729. | |

sched_switch histogram | | ———————- | |

The sched_switch histogram paired with the above sched_waking | | histogram is shown below. The most important aspect of the | | sched_switch histogram is that it references a variable on the | | sched_waking histogram above. | |

The histogram diagram is very similar to the others so far displayed, | | but it adds variable references. You can see the normal hitcount and | | key fields along with a new wakeup_lat variable implemented in the | | same way as the sched_waking ts0 variable, but in addition there’ s an | | entry with the new FL_VAR_REF (short for HIST_FIELD_FL_VAR_REF) flag. | |

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. | |

The diagram below describes the situation for the sched_switch | | histogram referred to before:

```
 # echo 'hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0'␣
↪>>      | |
        events/sched/sched_switch/trigger                              ␣
↪       | |
                                                                       ␣
↪       | |
 +------------------+                                                  ␣
↪       | |
 | hist_data        |                                                  ␣
↪       | |
 +------------------+   +-----------------------+                      ␣
↪       | |
   | .fields[]      |-->| val = hitcount        |                      ␣
↪       | |
   +----------------+   +-----------------------+                      ␣
↪       | |
   | .map           |     | .size               |                      ␣
↪       | |
   +----------------+     +---------------------+                      ␣
↪       | |
+--| .var_refs[]    |     | .offset             |                      ␣
↪       | |
|  +----------------+     +---------------------+                      ␣
↪       | |
|                         | .fn()               |                      ␣
↪       | |
|   var_ref_vals[]        +---------------------+                      ␣
↪       | |
|  +-------------+        | .flags              |                      ␣
↪       | |
|  | $ts0        |<---+   +---------------------+                      ␣
↪       | |
|  +-------------+    |   | .var.idx            |                      ␣
↪       | |
|  |             |    |   +---------------------+                      ␣
↪       | |
|  +-------------+    |   | .var.hist_data      |                      ␣
↪       | |
|  |             |    |   +---------------------+                      ␣
```

(continues on next page)

(continued from previous page)

```
↪        | |
|  +-------------+    |   | .var_ref_idx        |                ␣
↪        | |
|  |             |    | +----------------------+                ␣
↪        | |
|  +-------------+    | | var = wakeup_lat     |                ␣
↪        | |
|         .           | +----------------------+                ␣
↪        | |
|         .           |   | .size               |                ␣
↪        | |
|         .           |   +---------------------+                ␣
↪        | |
|  +-------------+    |   | .offset             |                ␣
↪        | |
|  |             |    |   +---------------------+                ␣
↪        | |
|  +-------------+    |   | .fn()               |                ␣
↪        | |
|  |             |    |   +---------------------+                ␣
↪        | |
|  +-------------+    |   | .flags & FL_VAR     |                ␣
↪        | |
|                     |   +---------------------+                ␣
↪        | |
|                     |   | .var.idx            |                ␣
↪        | |
|                     |   +---------------------+                ␣
↪        | |
|                     |   | .var.hist_data      |                ␣
↪        | |
|                     |   +---------------------+                ␣
↪        | |
|                     |   | .var_ref_idx        |                ␣
↪        | |
|                     |   +---------------------+                ␣
↪        | |
|                     |             .                            ␣
↪        | |
|                     |             .                            ␣
↪        | |
|                     |             .                            ␣
↪        | |
|                     | +----------------------+ <--- n_vals     ␣
↪        | |
|                     | | key = pid            |                ␣
↪        | |
|                     | +----------------------+                ␣
↪        | |
```

(continues on next page)

(continued from previous page)

```
|                      |   | .size                |                        ␣
↪        | |
|                      |   +--------------------+                        ␣
↪        | |
|                      |   | .offset              |                        ␣
↪        | |
|                      |   +--------------------+                        ␣
↪        | |
|                      |   | .fn()                |                        ␣
↪        | |
|                      |   +--------------------+                        ␣
↪        | |
|                      |   | .flags               |                        ␣
↪        | |
|                      |   +--------------------+                        ␣
↪        | |
|                      |   | .var.idx             |                        ␣
↪        | |
|                      | +----------------------+ <--- n_fields        ␣
↪        | |
|                      | | unused               |                        ␣
↪        | |
|                      | +----------------------+                        ␣
↪        | |
|                      |   |                    |                        ␣
↪        | |
|                      |   +--------------------+                        ␣
↪        | |
|                      |   |                    |                        ␣
↪        | |
|                      |   +--------------------+                        ␣
↪        | |
|                      |   |                    |                        ␣
↪        | |
|                      |   +--------------------+                        ␣
↪        | |
|                      |   |                    |                        ␣
↪        | |
|                      |   +--------------------+                        ␣
↪        | |
|                      |   |                    |                        ␣
↪        | |
|                      |   +--------------------+                        ␣
↪        | |
|                      |                          n_keys = n_fields -␣
↪n_vals | |
|                      |                                                 ␣
↪        | |
|                      |                                                 ␣
```

(continues on next page)

(continued from previous page)

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

Abbreviations used in the diagrams:

```
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
```

When a hist trigger makes use of a variable, a new hist_field is created with flag HIST_FIELD_FL_VAR_REF. For a VAR_REF field, the var.idx and var.hist_data take the same values as the referenced variable, as well as the referenced variable's size, type, and is_signed values. The VAR_REF field's .name is set to the name of the variable it references. If a variable reference was created using the explicit system.event.$var_ref notation, the hist_field's system and event_name variables are also set.

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_refs() 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.

16.3.1 Variable and variable reference test

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
```

Looking at the sched_waking 'hist_debug' output, in addition to the normal key and value hist_fields, in the val fields section we see a field with the HIST_FIELD_FL_VAR flag, which indicates that that field represents a variable. Note that in addition to the variable name, contained in the var.name field, it includes the var.idx, which is the index into the tracing_map_elt.vars[] array of the actual variable location. Note also that the output shows that variables live in the same part of the hist_data->fields[] array as normal values:

```
# 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
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
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
```

16.4 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.

The diagram below illustrates the new elements described above in the context of the sched_switch histogram using the onmatch() handler and the trace() action.

First, we define the wakeup_latency synthetic event:

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

Next, the sched_waking hist trigger as before:

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

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:

```
# echo 'hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0: \
        onmatch(sched.sched_waking).wakeup_latency($wakeup_lat,next_
↪pid)' >>
        /sys/kernel/debug/tracing/events/sched/sched_switch/trigger
```

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           |     | .size              |
    +----------------+     +--------------------+
+---| .field_vars[]  |     | .offset            |
|   +----------------+     +--------------------+
|+--| .var_refs[]    |     | .offset            |
||  +----------------+     +--------------------+
||                         | .fn()              |
||   var_ref_vals[]        +--------------------+
||  +-------------+        | .flags             |
||  | $ts0        |<---+   +--------------------+
||  +-------------+    |   | .var.idx           |
||  | $next_pid   |<-+ |   +--------------------+
||  +-------------+  | |   | .var.hist_data     |
||+>| $wakeup_lat |  | |   +--------------------+
||| +-------------+  | |   | .var_ref_idx       |
||| |             |  | | +----------------------+
||| +-------------+  | | | var = wakeup_lat     |
|||        .         | | +----------------------+
|||        .         | |   | .size              |
|||        .         | |   +--------------------+
||| +-------------+  | |   | .offset            |
||| |             |  | |   +--------------------+
||| +-------------+  | |   | .fn()              |
||| |             |  | |   +--------------------+
||| +-------------+  | |   | .flags & FL_VAR    |
|||                  | |   +--------------------+
|||                  | |   | .var.idx           |
|||                  | |   +--------------------+
|||                  | |   | .var.hist_data     |
|||                  | |   +--------------------+
|||                  | |   | .var_ref_idx       |
|||                  | |   +--------------------+
|||                  | |              .
|||                  | |              .
|||                  | |              .
|||                  | |              .
||| +--------------+ | |              .
+-->| field_var    | | |              .
 || +--------------+ | |              .
 ||   | var        | | |              .
 ||   +------------+ | |              .
 ||   | val        | | |              .
 || +--------------+ | |              .
 || | field_var    | | |              .
 || +--------------+ | |              .
```

(continues on next page)

(continued from previous page)

```
||    | var          | | |              .
||    +------------+ | |              .
||    | val          | | |              .
||    +------------+ | |              .
||          .         | |              .
||          .         | |              .
||          .         | | +----------------------+ <--- n_vals
|| +--------------+ | | | key = pid            |
|| | 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             |
||                   | |   +--------------------+
||                   | |   |                    |
||                   | |   +--------------------+
||                   | |
||                   | |
```

(continues on next page)

(continued from previous page)

```
 ||                  | | +-----------------------+
 +|------------------|-|>| var_ref = $ts0        |
  |                  | | +-----------------------+
  |                  | |   | .size               |
  |                  | |   +---------------------+
  |                  | |   | .offset             |
  |                  | |   +---------------------+
  |                  | |   | .fn()               |
  |                  | |   +---------------------+
  |                  | |   | .flags & FL_VAR_REF |
  |                  | |   +---------------------+
  |                  | +---| .var_ref_idx        |
  |                  |   +-----------------------+
  |                  |   | var_ref = $next_pid   |
  |                  |   +-----------------------+
  |                  |     | .size               |
  |                  |     +---------------------+
  |                  |     | .offset             |
  |                  |     +---------------------+
  |                  |     | .fn()               |
  |                  |     +---------------------+
  |                  |     | .flags & FL_VAR_REF |
  |                  |     +---------------------+
  |                  +-----| .var_ref_idx        |
  |                      +-----------------------+
  |                      | var_ref = $wakeup_lat |
  |                      +-----------------------+
  |                        | .size               |
  |                        +---------------------+
  |                        | .offset             |
  |                        +---------------------+
  |                        | .fn()               |
  |                        +---------------------+
  |                        | .flags & FL_VAR_REF |
  |                        +---------------------+
  +------------------------| .var_ref_idx        |
                           +---------------------+
```

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
```

16.4.1 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)' >> /sys/kernel/debug/tracing/events/sched/sched_
↪switch/trigger
```

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-
↪$ts0:sort=hitcount:size=2048:clock=global:onmatch(sched.sched_
↪waking).wakeup_latency($wakeup_lat,next_pid,next_comm) [active]
#

hist_data: 0000000008f551b7

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:
```

(continues on next page)

(continued from previous page)

```
      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: 0000000008f551b7
    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: 0000000008f551b7
    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: 0000000008f551b7
    var_ref_idx (into hist_data->var_refs[]): 3
    type: char[16]
```

(continues on next page)

(continued from previous page)

```
    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:

```
# echo '!hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-
↪$ts0:onmatch(sched.sched_waking).wakeup_latency($wakeup_lat,next_
↪pid,next_comm)' >> /sys/kernel/debug/tracing/events/sched/sched_
↪switch/trigger

# 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
```

16.4.2 action_data and the trace() action

As mentioned above, when the trace() action generates a synthetic event, all the parameters to the synthetic event either already are variables or are converted into variables (via field variables), and finally all those variable values are collected via references to them into a var_ref_vals[] array.

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:

```
+------------------+     wakeup_latency()
| action_data      |       event params               var_ref_vals[]
+------------------+    +-----------------+        +-----------------+
  | .var_ref_idx[] |--->| $wakeup_lat idx |---+    |                 |
  +----------------+    +-----------------+   |    +-----------------+
  | .synth_event   |    | $next_pid idx   |---|-+  | $wakeup_lat val |
  +----------------+    +-----------------+   | |  +-----------------+
                                 .            | +->| $next_pid val   |
                                 .            |    +-----------------+
                                 .            |           .
                        +-----------------+   |           .
                        |                 |   |           .
                        +-----------------+   |    +-----------------+
                                              +--->| $wakeup_lat val |
                                                   +-----------------+
```

Basically, how this ends up getting used in the synthetic event probe function, trace_event_raw_event_synth(), is as follows:

```
for each field i in .synth_event
  val_idx = .var_ref_idx[i]
  val = var_ref_vals[val_idx]
```

16.4.3 action_data and the onXXX() handlers

The hist trigger onXXX() actions other than onmatch(), such as onmax() and onchange(), also make use of and internally create hidden variables. This information is contained in the action_data.track_data struct, and is also visible in the hist_debug output as will be described in the example below.

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

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

or:

```
# echo 'hist:keys=next_pid:wakeup_lat=common_timestamp.usecs-$ts0: \
        onmax($wakeup_lat).snapshot()' >>
        /sys/kernel/debug/tracing/events/sched/sched_switch/trigger
```

16.4.4 save() action field variable test

For this example, instead of generating a synthetic event, the save() action is used to save field values whenever an onmax() handler detects that a new max latency has been hit. As in the previous example, the values being saved are also field values, but in this case, are kept in a separate hist_data array named save_vars[].

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
```

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:

```
# 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_switch/trigger
```

The sched_waking hist_debug output shows the same data as in the previous test examples:

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

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

(continues on next page)

(continued from previous page)

```
hist_data: 00000000e6290f48

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

First, the action tracking variables section now shows the actions[].track_data information describing the special tracking variables and references used to track, in this case, the running maximum value. The actions[].track_data.var_ref member contains the reference to the variable being tracked, in this case the $wakeup_lat variable. In order to perform the onmax() handler function, there also needs to be a variable that tracks the current maximum by getting updated whenever a new maximum is hit. In this case, we can see that an auto-generated variable named '__max' has been created and is visible in the actions[].track_data.track_var variable.

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:

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

# event histogram
#
# trigger info: hist:keys=next_pid:vals=hitcount:wakeup_lat=common_
↪timestamp.usecs-$ts0:sort=hitcount:size=2048:clock=global:onmax(
↪$wakeup_lat).save(next_comm,prev_pid,prev_prio,prev_comm) [active]
#

hist_data: 0000000057bcd28d

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
```

(continues on next page)

(continued from previous page)

```
    name: ts0
    var.idx (into tracing_map_elt.vars[]): 0
    var.hist_data: 00000000e6290f48
    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: 0000000057bcd28d
    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: 0000000057bcd28d
    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
```

(continues on next page)

(continued from previous page)

```
    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_switch/trigger

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

16.5 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.

16.5.1 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
```

(continues on next page)

(continued from previous page)

```
    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-
↪$ts0:sort=hitcount:size=2048:clock=global:onmatch(sched.sched_
↪waking).wakeup_latency($wakeup_lat,next_pid,prio) [active]
#

hist_data: 00000000a73b67df

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
```

(continues on next page)

(continued from previous page)

```
    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: 00000000a73b67df
    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: 00000000a73b67df
    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
```

(continues on next page)

(continued from previous page)

```
    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)' >> events/sched/sched_switch/trigger

# 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
```

16.5.2 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:onmatch(sched.sched_waking).wakeup_latency(
↪$wakeup_lat,$woken_pid,next_comm)' >> 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
```

(continues on next page)

(continued from previous page)

```
    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:onmatch(sched.sched_
↪waking).wakeup_latency($wakeup_lat,$woken_pid,next_comm) [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
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
    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: 0000000055d65ed0
    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: 0000000055d65ed0
    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: 0000000055d65ed0
    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
```

(continues on next page)

(continued from previous page)

```
    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(
↪$wakeup_lat,$woken_pid,next_comm)' >> events/sched/sched_switch/
↪trigger

# 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
```

CHAPTER

SEVENTEEN

BOOT-TIME TRACING

Author
Masami Hiramatsu <mhiramat@kernel.org>

17.1 Overview

Boot-time tracing allows users to trace boot-time process including device initialization with full features of ftrace including per-event filter and actions, histograms, kprobe-events and synthetic-events, and trace instances. Since kernel command line is not enough to control these complex features, this uses bootconfig file to describe tracing feature programming.

17.2 Options in the Boot Config

Here is the list of available options list for boot time tracing in boot config file[1]. All options are under "ftrace." or "kernel." prefix. See kernel parameters for the options which starts with "kernel." prefix[2].

17.2.1 Ftrace Global Options

Ftrace global options have "kernel." prefix in boot config, which means these options are passed as a part of kernel legacy command line.

kernel.tp_printk
Output trace-event data on printk buffer too.

kernel.dump_on_oops [= MODE]
Dump ftrace on Oops. If MODE = 1 or omitted, dump trace buffer on all CPUs. If MODE = 2, dump a buffer on a CPU which kicks Oops.

kernel.traceoff_on_warning
Stop tracing if WARN_ON() occurs.

kernel.fgraph_max_depth = MAX_DEPTH
Set MAX_DEPTH to maximum depth of fgraph tracer.

[1] See Documentation/admin-guide/bootconfig.rst

[2] See Documentation/admin-guide/kernel-parameters.rst

kernel.fgraph_filters = FILTER[, FILTER2…]
Add fgraph tracing function filters.

kernel.fgraph_notraces = FILTER[, FILTER2…]
Add fgraph non-tracing function filters.

17.2.2 Ftrace Per-instance Options

These options can be used for each instance including global ftrace node.

ftrace.[instance.INSTANCE.]options = OPT1[, OPT2[…]]
Enable given ftrace options.

ftrace.[instance.INSTANCE.]tracing_on = 0|1
Enable/Disable tracing on this instance when starting boot-time tracing. (you can enable it by the "traceon" event trigger action)

ftrace.[instance.INSTANCE.]trace_clock = CLOCK
Set given CLOCK to ftrace's trace_clock.

ftrace.[instance.INSTANCE.]buffer_size = SIZE
Configure ftrace buffer size to SIZE. You can use "KB" or "MB" for that SIZE.

ftrace.[instance.INSTANCE.]alloc_snapshot
Allocate snapshot buffer.

ftrace.[instance.INSTANCE.]cpumask = CPUMASK
Set CPUMASK as trace cpu-mask.

ftrace.[instance.INSTANCE.]events = EVENT[, EVENT2[…]]
Enable given events on boot. You can use a wild card in EVENT.

ftrace.[instance.INSTANCE.]tracer = TRACER
Set TRACER to current tracer on boot. (e.g. function)

ftrace.[instance.INSTANCE.]ftrace.filters
This will take an array of tracing function filter rules.

ftrace.[instance.INSTANCE.]ftrace.notraces
This will take an array of NON-tracing function filter rules.

17.2.3 Ftrace Per-Event Options

These options are setting per-event options.

ftrace.[instance.INSTANCE.]event.GROUP.EVENT.enable
Enable GROUP:EVENT tracing.

ftrace.[instance.INSTANCE.]event.GROUP.EVENT.filter = FILTER
Set FILTER rule to the GROUP:EVENT.

ftrace.[instance.INSTANCE.]event.GROUP.EVENT.actions = ACTION[, ACTION2[…]]
Set ACTIONs to the GROUP:EVENT.

ftrace.[instance.INSTANCE.]event.kprobes.EVENT.probes = PROBE[, PROBE2[…]]
Defines new kprobe event based on PROBEs. It is able to define multiple

probes on one event, but those must have same type of arguments. This option is available only for the event which group name is "kprobes".

ftrace.[instance.INSTANCE.]event.synthetic.EVENT.fields = FIELD[, FIELD2[…]]
Defines new synthetic event with FIELDs. Each field should be "type varname".

Note that kprobe and synthetic event definitions can be written under instance node, but those are also visible from other instances. So please take care for event name conflict.

17.3 When to Start

All boot-time tracing options starting with `ftrace` will be enabled at the end of core_initcall. This means you can trace the events from postcore_initcall. Most of the subsystems and architecture dependent drivers will be initialized after that (arch_initcall or subsys_initcall). Thus, you can trace those with boot-time tracing. If you want to trace events before core_initcall, you can use the options starting with `kernel`. Some of them will be enabled eariler than the initcall processing (for example,. `kernel.ftrace=function` and `kernel.trace_event` will start before the initcall.)

17.4 Examples

For example, to add filter and actions for each event, define kprobe events, and synthetic events with histogram, write a boot config like below:

```
ftrace.event {
      task.task_newtask {
              filter = "pid < 128"
              enable
      }
      kprobes.vfs_read {
              probes = "vfs_read $arg1 $arg2"
              filter = "common_pid < 200"
              enable
      }
      synthetic.initcall_latency {
              fields = "unsigned long func", "u64 lat"
              actions = "hist:keys=func.sym,lat:vals=lat:sort=lat"
      }
      initcall.initcall_start {
              actions = "hist:keys=func:ts0=common_timestamp.usecs"
      }
      initcall.initcall_finish {
              actions = "hist:keys=func:lat=common_timestamp.usecs-
→$ts0:onmatch(initcall.initcall_start).initcall_latency(func,$lat)"
```

(continues on next page)

(continued from previous page)

```
	}
}
```

Also, boot-time tracing supports "instance" node, which allows us to run several tracers for different purpose at once. For example, one tracer is for tracing functions starting with "user_", and others tracing "kernel_" functions, you can write boot config as below:

```
ftrace.instance {
	foo {
		tracer = "function"
		ftrace.filters = "user_*"
	}
	bar {
		tracer = "function"
		ftrace.filters = "kernel_*"
	}
}
```

The instance node also accepts event nodes so that each instance can customize its event tracing.

With the trigger action and kprobes, you can trace function-graph while a function is called. For example, this will trace all function calls in the pci_proc_init():

```
ftrace {
	tracing_on = 0
	tracer = function_graph
	event.kprobes {
		start_event {
			probes = "pci_proc_init"
			actions = "traceon"
		}
		end_event {
			probes = "pci_proc_init%return"
			actions = "traceoff"
		}
	}
}
```

This boot-time tracing also supports ftrace kernel parameters via boot config. For example, following kernel parameters:

```
trace_options=sym-addr trace_event=initcall:* tp_printk trace_buf_
↪size=1M ftrace=function ftrace_filter="vfs*"
```

This can be written in boot config like below:

```
kernel {
	trace_options = sym-addr
	trace_event = "initcall:*"
```

(continues on next page)

(continued from previous page)

```
        tp_printk
        trace_buf_size = 1M
        ftrace = function
        ftrace_filter = "vfs*"
}
```

Note that parameters start with “kernel” prefix instead of “ftrace”.

CHAPTER

EIGHTEEN

HARDWARE LATENCY DETECTOR

18.1 Introduction

The tracer hwlat_detector is a special purpose tracer that is used to detect large system latencies induced by the behavior of certain underlying hardware or firmware, independent of Linux itself. The code was developed originally to detect SMIs (System Management Interrupts) on x86 systems, however there is nothing x86 specific about this patchset. It was originally written for use by the "RT" patch since the Real Time kernel is highly latency sensitive.

SMIs are not serviced by the Linux kernel, which means that it does not even know that they are occuring. SMIs are instead set up by BIOS code and are serviced by BIOS code, usually for "critical"events such as management of thermal sensors and fans. Sometimes though, SMIs are used for other tasks and those tasks can spend an inordinate amount of time in the handler (sometimes measured in milliseconds). Obviously this is a problem if you are trying to keep event service latencies down in the microsecond range.

The hardware latency detector works by hogging one of the cpus for configurable amounts of time (with interrupts disabled), polling the CPU Time Stamp Counter for some period, then looking for gaps in the TSC data. Any gap indicates a time when the polling was interrupted and since the interrupts are disabled, the only thing that could do that would be an SMI or other hardware hiccup (or an NMI, but those can be tracked).

Note that the hwlat detector should *NEVER* be used in a production environment. It is intended to be run manually to determine if the hardware platform has a problem with long system firmware service routines.

18.2 Usage

Write the ASCII text "hwlat" into the current_tracer file of the tracing system (mounted at /sys/kernel/tracing or /sys/kernel/tracing). It is possible to redefine the threshold in microseconds (us) above which latency spikes will be taken into account.

Example:

```
# echo hwlat > /sys/kernel/tracing/current_tracer
# echo 100 > /sys/kernel/tracing/tracing_thresh
```

The /sys/kernel/tracing/hwlat_detector interface contains the following files:

- **width - time period to sample with CPUs held (usecs)**
  must be less than the total window size (enforced)
- window - total period of sampling, width being inside (usecs)

By default the width is set to 500,000 and window to 1,000,000, meaning that for every 1,000,000 usecs (1s) the hwlat detector will spin for 500,000 usecs (0.5s). If tracing_thresh contains zero when hwlat tracer is enabled, it will change to a default of 10 usecs. If any latencies that exceed the threshold is observed then the data will be written to the tracing ring buffer.

The minimum sleep time between periods is 1 millisecond. Even if width is less than 1 millisecond apart from window, to allow the system to not be totally starved.

If tracing_thresh was zero when hwlat detector was started, it will be set back to zero if another tracer is loaded. Note, the last value in tracing_thresh that hwlat detector had will be saved and this value will be restored in tracing_thresh if it is still zero when hwlat detector is started again.

The following tracing directory files are used by the hwlat_detector:

in /sys/kernel/tracing:

- tracing_threshold - minimum latency value to be considered (usecs)
- tracing_max_latency - maximum hardware latency actually observed (usecs)
- tracing_cpumask - the CPUs to move the hwlat thread across
- hwlat_detector/width - specified amount of time to spin within window (usecs)
- hwlat_detector/window - amount of time between (width) runs (usecs)

The hwlat detector' s kernel thread will migrate across each CPU specified in tracing_cpumask between each window. To limit the migration, either modify tracing_cpumask, or modify the hwlat kernel thread (named [hwlatd]) CPU affinity directly, and the migration will stop.

CHAPTER

NINETEEN

INTEL(R) TRACE HUB (TH)

19.1 Overview

Intel(R) Trace Hub (TH) is a set of hardware blocks that produce, switch and output trace data from multiple hardware and software sources over several types of trace output ports encoded in System Trace Protocol (MIPI STPv2) and is intended to perform full system debugging. For more information on the hardware, see Intel(R) Trace Hub developer's manual [1].

It consists of trace sources, trace destinations (outputs) and a switch (Global Trace Hub, GTH). These devices are placed on a bus of their own ("intel_th"), where they can be discovered and configured via sysfs attributes.

Currently, the following Intel TH subdevices (blocks) are supported:

- Software Trace Hub (STH), trace source, which is a System Trace Module (STM) device,
- Memory Storage Unit (MSU), trace output, which allows storing trace hub output in system memory,
- Parallel Trace Interface output (PTI), trace output to an external debug host via a PTI port,
- Global Trace Hub (GTH), which is a switch and a central component of Intel(R) Trace Hub architecture.

Common attributes for output devices are described in Documentation/ABI/testing/sysfs-bus-intel_th-output-devices, the most notable of them is "active", which enables or disables trace output into that particular output device.

GTH allows directing different STP masters into different output ports via its "masters" attribute group. More detailed GTH interface description is at Documentation/ABI/testing/sysfs-bus-intel_th-devices-gth.

STH registers an stm class device, through which it provides interface to userspace and kernelspace software trace sources. See *System Trace Module* for more information on that.

MSU can be configured to collect trace data into a system memory buffer, which can later on be read from its device nodes via read() or mmap() interface and directed to a "software sink" driver that will consume the data and/or relay it further.

On the whole, Intel(R) Trace Hub does not require any special userspace software to function; everything can be configured, started and collected via sysfs attributes, and device nodes.

[1] https://software.intel.com/sites/default/files/managed/d3/3c/intel-th-developer-manual.pdf

19.2 Bus and Subdevices

For each Intel TH device in the system a bus of its own is created and assigned an id number that reflects the order in which TH devices were enumerated. All TH subdevices (devices on intel_th bus) begin with this id: 0-gth, 0-msc0, 0-msc1, 0-pti, 0-sth, which is followed by device's name and an optional index.

Output devices also get a device node in /dev/intel_thN, where N is the Intel TH device id. For example, MSU's memory buffers, when allocated, are accessible via /dev/intel_th0/msc{0,1}.

19.3 Quick example

figure out which GTH port is the first memory controller:

```
$ cat /sys/bus/intel_th/devices/0-msc0/port
0
```

looks like it's port 0, configure master 33 to send data to port 0:

```
$ echo 0 > /sys/bus/intel_th/devices/0-gth/masters/33
```

allocate a 2-windowed multiblock buffer on the first memory # controller, each with 64 pages:

```
$ echo multi > /sys/bus/intel_th/devices/0-msc0/mode
$ echo 64,64 > /sys/bus/intel_th/devices/0-msc0/nr_pages
```

enable wrapping for this controller, too:

```
$ echo 1 > /sys/bus/intel_th/devices/0-msc0/wrap
```

and enable tracing into this port:

```
$ echo 1 > /sys/bus/intel_th/devices/0-msc0/active
```

.. send data to master 33, see stm.txt for more details .. # .. wait for traces to pile up .. # .. and stop the trace:

```
$ echo 0 > /sys/bus/intel_th/devices/0-msc0/active
```

and now you can collect the trace from the device node:

```
$ cat /dev/intel_th0/msc0 > my_stp_trace
```

19.4 Host Debugger Mode

It is possible to configure the Trace Hub and control its trace capture from a remote debug host, which should be connected via one of the hardware debugging interfaces, which will then be used to both control Intel Trace Hub and transfer its trace data to the debug host.

The driver needs to be told that such an arrangement is taking place so that it does not touch any capture/port configuration and avoids conflicting with the debug host' s configuration accesses. The only activity that the driver will perform in this mode is collecting software traces to the Software Trace Hub (an stm class device). The user is still responsible for setting up adequate master/channel mappings that the decoder on the receiving end would recognize.

In order to enable the host mode, set the 'host_mode' parameter of the 'intel_th' kernel module to 'y'. None of the virtual output devices will show up on the intel_th bus. Also, trace configuration and capture controlling attribute groups of the 'gth' device will not be exposed. The 'sth' device will operate as usual.

19.5 Software Sinks

The Memory Storage Unit (MSU) driver provides an in-kernel API for drivers to register themselves as software sinks for the trace data. Such drivers can further export the data via other devices, such as USB device controllers or network cards.

The API has two main parts::

- notifying the software sink that a particular window is full, and "locking" that window, that is, making it unavailable for the trace collection; when this happens, the MSU driver will automatically switch to the next window in the buffer if it is unlocked, or stop the trace capture if it' s not;
- tracking the "locked" state of windows and providing a way for the software sink driver to notify the MSU driver when a window is unlocked and can be used again to collect trace data.

An example sink driver, msu-sink illustrates the implementation of a software sink. Functionally, it simply unlocks windows as soon as they are full, keeping the MSU running in a circular buffer mode. Unlike the "multi" mode, it will fill out all the windows in the buffer as opposed to just the first one. It can be enabled by writing "sink" to the "mode" file (assuming msu-sink.ko is loaded).

CHAPTER

TWENTY

LOCKLESS RING BUFFER DESIGN

Author
Steven Rostedt <srostedt@redhat.com>

Reviewers
Mathieu Desnoyers, Huang Ying, Hidetoshi Seto, and Frederic Weisbecker.

Written for: 2.6.31

20.1 Terminology used in this Document

tail

- where new writes happen in the ring buffer.

head

- where new reads happen in the ring buffer.

producer

- the task that writes into the ring buffer (same as writer)

writer

- same as producer

consumer

- the task that reads from the buffer (same as reader)

reader

- same as consumer.

reader_page

- A page outside the ring buffer used solely (for the most part) by the reader.

head_page

- a pointer to the page that the reader will use next

tail_page

- a pointer to the page that will be written to next

commit_page

- a pointer to the page with the last finished non-nested write.

cmpxchg

- hardware-assisted atomic transaction that performs the following:

  ```
  A = B if previous A == C

  R = cmpxchg(A, C, B) is saying that we replace A with B if␣
  ↪and only
      if current A is equal to C, and we put the old (current)
      A into R

  R gets the previous A regardless if A is updated with B or␣
  ↪not.
  ```

  To see if the update was successful a compare of `R == C` may be used.

20.2 The Generic Ring Buffer

The ring buffer can be used in either an overwrite mode or in producer/consumer mode.

Producer/consumer mode is where if the producer were to fill up the buffer before the consumer could free up anything, the producer will stop writing to the buffer. This will lose most recent events.

Overwrite mode is where if the producer were to fill up the buffer before the consumer could free up anything, the producer will overwrite the older data. This will lose the oldest events.

No two writers can write at the same time (on the same per-cpu buffer), but a writer may interrupt another writer, but it must finish writing before the previous writer may continue. This is very important to the algorithm. The writers act like a "stack". The way interrupts works enforces this behavior:

```
writer1 start
   <preempted> writer2 start
       <preempted> writer3 start
                   writer3 finishes
               writer2 finishes
writer1 finishes
```

This is very much like a writer being preempted by an interrupt and the interrupt doing a write as well.

Readers can happen at any time. But no two readers may run at the same time, nor can a reader preempt/interrupt another reader. A reader cannot preempt/interrupt a writer, but it may read/consume from the buffer at the same time as a writer is writing, but the reader must be on another processor to do so. A reader may read on its own processor and can be preempted by a writer.

A writer can preempt a reader, but a reader cannot preempt a writer. But a reader can read the buffer at the same time (on another processor) as a writer.

The ring buffer is made up of a list of pages held together by a linked list.

At initialization a reader page is allocated for the reader that is not part of the ring buffer.

The head_page, tail_page and commit_page are all initialized to point to the same page.

The reader page is initialized to have its next pointer pointing to the head page, and its previous pointer pointing to a page before the head page.

The reader has its own page to use. At start up time, this page is allocated but is not attached to the list. When the reader wants to read from the buffer, if its page is empty (like it is on start-up), it will swap its page with the head_page. The old reader page will become part of the ring buffer and the head_page will be removed. The page after the inserted page (old reader_page) will become the new head page.

Once the new page is given to the reader, the reader could do what it wants with it, as long as a writer has left that page.

A sample of how the reader page is swapped: Note this does not show the head page in the buffer, it is for demonstrating a swap only.

```
+------+
|reader|          RING BUFFER
|page  |
+------+
                +---+   +---+   +---+
                |   |-->|   |-->|   |
                |   |<--|   |<--|   |
                +---+   +---+   +---+
                 ^ |             ^ |
                 | +-------------+ |
                 +-----------------+


+------+
|reader|          RING BUFFER
|page  |-------------------+
+------+                   v
  |             +---+   +---+   +---+
  |             |   |-->|   |-->|   |
  |             |   |<--|   |<--|   |<-+
  |             +---+   +---+   +---+  |
  |              ^ |             ^ |   |
```

(continues on next page)

(continued from previous page)

```
  |              | +-------------+ |   |
  |              +-----------------+   |
  +------------------------------------+

+------+
|reader|          RING BUFFER
|page  |------------------+
+------+ <---------------+ v
  |  ^          +---+   +---+   +---+
  |  |          |   |-->|   |-->|   |
  |  |          |   |   |   |<--|   |<-+
  |  |          +---+   +---+   +---+  |
  |  |             |            ^ |   |
  |  |             +-------------+ |   |
  |  +-----------------------------+   |
  +------------------------------------+

+------+
|buffer|          RING BUFFER
|page  |------------------+
+------+ <---------------+ v
  |  ^          +---+   +---+   +---+
  |  |          |   |   |   |-->|   |
  |  |  New     |   |   |   |<--|   |<-+
  |  | Reader   +---+   +---+   +---+  |
  |  |  page ----^                 |   |
  |  |                             |   |
  |  +-----------------------------+   |
  +------------------------------------+
```

It is possible that the page swapped is the commit page and the tail page, if what is in the ring buffer is less than what is held in a buffer page.

```
          reader page    commit page   tail page
              |              |             |
              v              |             |
             +---+           |             |
             |   |<----------+             |
             |   |<------------------------+
             |   |------+
             +---+      |
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

This case is still valid for this algorithm. When the writer leaves the page, it simply goes into the ring buffer since the reader page still points to the next location in the ring buffer.

The main pointers:

reader page

- The page used solely by the reader and is not part of the ring buffer (may be swapped in)

head page

- the next page in the ring buffer that will be swapped with the reader page.

tail page

- the page where the next write will take place.

commit page

- the page that last finished a write.

The commit page only is updated by the outermost writer in the writer stack. A writer that preempts another writer will not move the commit page.

When data is written into the ring buffer, a position is reserved in the ring buffer and passed back to the writer. When the writer is finished writing data into that position, it commits the write.

Another write (or a read) may take place at anytime during this transaction. If another write happens it must finish before continuing with the previous write.

Write reserve:

```
 Buffer page
+---------+
|written  |
+---------+  <--- given back to writer (current␣
↪commit)
|reserved |
+---------+ <--- tail pointer
| empty   |
+---------+
```

Write commit:

```
 Buffer page
+---------+
|written  |
+---------+
|written  |
+---------+  <--- next position for write (current␣
↪commit)
| empty   |
+---------+
```

If a write happens after the first reserve:

```
     Buffer page
    +---------+
    |written  |
    +---------+  <-- current commit
    |reserved |
    +---------+  <--- given back to second writer
    |reserved |
    +---------+ <--- tail pointer

After second writer commits::


     Buffer page
    +---------+
    |written  |
    +---------+  <--(last full commit)
    |reserved |
    +---------+
    |pending  |
    |commit   |
    +---------+ <--- tail pointer

When the first writer commits::

     Buffer page
    +---------+
    |written  |
    +---------+
    |written  |
    +---------+
    |written  |
    +---------+  <--(last full commit and tail pointer)
```

The commit pointer points to the last write location that was committed without preempting another write. When a write that preempted another write is committed, it only becomes a pending commit and will not be a full commit until all writes have been committed.

The commit page points to the page that has the last full commit. The tail page points to the page with the last write (before committing).

The tail page is always equal to or after the commit page. It may be several pages ahead. If the tail page catches up to the commit page then no more writes may take place (regardless of the mode of the ring buffer: overwrite and produce/consumer).

The order of pages is:

```
head page
commit page
tail page
```

Possible scenario:

```
                             tail page
  head page         commit page  |
      |                 |        |
      v                 v        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

There is a special case that the head page is after either the commit page and possibly the tail page. That is when the commit (and tail) page has been swapped with the reader page. This is because the head page is always part of the ring buffer, but the reader page is not. Whenever there has been less than a full page that has been committed inside the ring buffer, and a reader swaps out a page, it will be swapping out the commit page.

```
          reader page    commit page   tail page
              |              |             |
              v              |             |
             +---+           |             |
             |   |<----------+             |
             |   |<------------------------+
             |   |------+
             +---+      |
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
                        ^
                        |
                    head page
```

In this case, the head page will not move when the tail and commit move back into the ring buffer.

The reader cannot swap a page into the ring buffer if the commit page is still on that page. If the read meets the last commit (real commit not pending or reserved), then there is nothing more to read. The buffer is considered empty until another full commit finishes.

When the tail meets the head page, if the buffer is in overwrite mode, the head page will be pushed ahead one. If the buffer is in producer/consumer mode, the write will fail.

Overwrite mode:

```
            tail page
               |
               v
```

(continues on next page)

(continued from previous page)

```
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
                        ^
                        |
                    head page


            tail page
               |
               v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
                                 ^
                                 |
                             head page


                    tail page
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
                                 ^
                                 |
                             head page
```

Note, the reader page will still point to the previous head page. But when a swap takes place, it will use the most recent head page.

20.3 Making the Ring Buffer Lockless:

The main idea behind the lockless algorithm is to combine the moving of the head_page pointer with the swapping of pages with the reader. State flags are placed inside the pointer to the page. To do this, each page must be aligned in memory by 4 bytes. This will allow the 2 least significant bits of the address to be used as flags, since they will always be zero for the address. To get the address, simply mask out the flags:

```
MASK = ~3

address & MASK
```

Two flags will be kept by these two bits:

HEADER

- the page being pointed to is a head page

UPDATE

- the page being pointed to is being updated by a writer and was or is about to be a head page.

```
          reader page
              |
              v
            +---+
            |   |------+
            +---+      |
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-H->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

The above pointer "-H->" would have the HEADER flag set. That is the next page is the next page to be swapped out by the reader. This pointer means the next page is the head page.

When the tail page meets the head pointer, it will use cmpxchg to change the pointer to the UPDATE state:

```
            tail page
               |
               v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-H->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+

            tail page
               |
               v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

"-U->" represents a pointer in the UPDATE state.

Any access to the reader will need to take some sort of lock to serialize the readers. But the writers will never take a lock to write to the ring buffer. This means we only need to worry about a single reader, and writes only preempt in "stack" formation.

When the reader tries to swap the page with the ring buffer, it will also use cmpxchg. If the flag bit in the pointer to the head page does not have the HEADER flag set, the compare will fail and the reader will need to look for the new head page and try again. Note, the flags UPDATE and HEADER are never set at the same time.

The reader swaps the reader page as follows:

```
+------+
|reader|          RING BUFFER
|page  |
+------+
                +---+    +---+    +---+
                |   |--->|   |--->|   |
                |   |<---|   |<---|   |
                +---+    +---+    +---+
                 ^ |               ^ |
                 | +---------------+ |
                 +-----H-------------+
```

The reader sets the reader page next pointer as HEADER to the page after the head page:

```
+------+
|reader|          RING BUFFER
|page  |-------H-----------+
+------+                   v
  |             +---+    +---+    +---+
  |             |   |--->|   |--->|   |
  |             |   |<---|   |<---|   |<-+
  |             +---+    +---+    +---+  |
  |              ^ |               ^ |   |
  |              | +---------------+ |   |
  |              +-----H-------------+   |
  +--------------------------------------+
```

It does a cmpxchg with the pointer to the previous head page to make it point to the reader page. Note that the new pointer does not have the HEADER flag set. This action atomically moves the head page forward:

```
+------+
|reader|          RING BUFFER
|page  |-------H-----------+
+------+                   v
  |  ^          +---+   +---+   +---+
  |  |          |   |-->|   |-->|   |
  |  |          |   |<--|   |<--|   |<-+
  |  |          +---+   +---+   +---+  |
  |  |             |             ^ |   |
  |  |             +-------------+ |   |
  |  +-----------------------------+   |
  +------------------------------------+
```

After the new head page is set, the previous pointer of the head page is updated to the reader page:

```
+------+
|reader|          RING BUFFER
|page  |-------H-----------+
+------+ <---------------+ v
  |  ^          +---+   +---+   +---+
  |  |          |   |-->|   |-->|   |
  |  |          |   |   |   |<--|   |<-+
  |  |          +---+   +---+   +---+  |
  |  |             |             ^ |   |
  |  |             +-------------+ |   |
  |  +-----------------------------+   |
  +------------------------------------+

+------+
|buffer|          RING BUFFER
|page  |-------H-----------+  <--- New head page
+------+ <---------------+ v
  |  ^          +---+   +---+   +---+
  |  |          |   |   |   |-->|   |
  |  |  New     |   |   |   |<--|   |<-+
  |  | Reader   +---+   +---+   +---+  |
  |  |  page ----^                 |   |
  |  |                             |   |
  |  +-----------------------------+   |
  +------------------------------------+
```

Another important point: The page that the reader page points back to by its previous pointer (the one that now points to the new head page) never points back to the reader page. That is because the reader page is not part of the ring buffer. Traversing the ring buffer via the next pointers will always stay in the ring buffer. Traversing the ring buffer via the prev pointers may not.

Note, the way to determine a reader page is simply by examining the previous pointer of the page. If the next pointer of the previous page does not point back to the original page, then the original page is a reader page:

```
 +--------+
 | reader |  next   +----+
 |  page  |-------->|    |<====== (buffer page)
 +--------+         +----+
     |                | ^
     |                v | next
prev |              +----+
     +------------->|    |
                    +----+
```

The way the head page moves forward:

When the tail page meets the head page and the buffer is in overwrite mode and more writes take place, the head page must be moved forward before the writer may move the tail page. The way this is done is that the writer performs a cmpxchg to convert the pointer to the head page from the HEADER flag to have the UPDATE flag set. Once this is done, the reader will not be able to swap the head page from the buffer, nor will it be able to move the head page, until the writer is finished with the move.

This eliminates any races that the reader can have on the writer. The reader must spin, and this is why the reader cannot preempt the writer:

```
           tail page
              |
              v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-H->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+

           tail page
              |
              v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

The following page will be made into the new head page:

```
          tail page
              |
              v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |-H->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

After the new head page has been set, we can set the old head page pointer back to NORMAL:

```
          tail page
              |
              v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |-H->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

After the head page has been moved, the tail page may now move forward:

```
                   tail page
                       |
                       v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |-H->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
```

(continues on next page)

(continued from previous page)

```
    +---+    +---+    +---+    +---+
```

The above are the trivial updates. Now for the more complex scenarios.

As stated before, if enough writes preempt the first write, the tail page may make it all the way around the buffer and meet the commit page. At this time, we must start dropping writes (usually with some kind of warning to the user). But what happens if the commit was still on the reader page? The commit page is not part of the ring buffer. The tail page must account for this:

```
          reader page    commit page
              |              |
              v              |
             +---+           |
             |   |<----------+
             |   |
             |   |------+
             +---+      |
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-H->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
               ^
               |
           tail page
```

If the tail page were to simply push the head page forward, the commit when leaving the reader page would not be pointing to the correct page.

The solution to this is to test if the commit page is on the reader page before pushing the head page. If it is, then it can be assumed that the tail page wrapped the buffer, and we must drop new writes.

This is not a race condition, because the commit page can only be moved by the outermost writer (the writer that was preempted). This means that the commit will not move while a writer is moving the tail page. The reader cannot swap the reader page if it is also being used as the commit page. The reader can simply check that the commit is off the reader page. Once the commit page leaves the reader page it will never go back on it unless a reader does another swap with the buffer page that is also the commit page.

20.4 Nested writes

In the pushing forward of the tail page we must first push forward the head page if the head page is the next page. If the head page is not the next page, the tail page is simply updated with a cmpxchg.

Only writers move the tail page. This must be done atomically to protect against nested writers:

```
temp_page = tail_page
next_page = temp_page->next
cmpxchg(tail_page, temp_page, next_page)
```

The above will update the tail page if it is still pointing to the expected page. If this fails, a nested write pushed it forward, the current write does not need to push it:

```
           temp page
               |
               v
            tail page
               |
               v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

Nested write comes in and moves the tail page forward:

```
                    tail page (moved by nested writer)
            temp page   |
               |        |
               v        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

The above would fail the cmpxchg, but since the tail page has already been moved forward, the writer will just try again to reserve storage on the new tail page.

But the moving of the head page is a bit more complex:

```
            tail page
               |
               v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-H->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

The write converts the head page pointer to UPDATE:

```
            tail page
               |
               v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

But if a nested writer preempts here, it will see that the next page is a head page, but it is also nested. It will detect that it is nested and will save that information. The detection is the fact that it sees the UPDATE flag instead of a HEADER or NORMAL pointer.

The nested writer will set the new head page pointer:

```
           tail page
               |
               v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |-H->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

But it will not reset the update back to normal. Only the writer that converted a pointer from HEAD to UPDATE will convert it back to NORMAL:

```
                    tail page
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |-H->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

After the nested writer finishes, the outermost writer will convert the UPDATE pointer to NORMAL:

```
                    tail page
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |-H->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

It can be even more complex if several nested writes came in and moved the tail page ahead several pages:

```
(first writer)

            tail page
```

(continues on next page)

(continued from previous page)

```
               |
               v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-H->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

The write converts the head page pointer to UPDATE:

```
            tail page
               |
               v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |--->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

Next writer comes in, and sees the update and sets up the new head page:

```
(second writer)

           tail page
               |
               v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |-H->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

The nested writer moves the tail page forward. But does not set the old update page to NORMAL because it is not the outermost writer:

```
                    tail page
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |-H->|   |--->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

Another writer preempts and sees the page after the tail page is a head page. It changes it from HEAD to UPDATE:

```
(third writer)

                    tail page
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |-U->|   |--->
```

(continues on next page)

(continued from previous page)

```
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

The writer will move the head page forward:

```
(third writer)

                    tail page
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |-U->|   |-H->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

But now that the third writer did change the HEAD flag to UPDATE it will convert it to normal:

```
(third writer)

                    tail page
                        |
                        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |--->|   |-H->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

Then it will move the tail page, and return back to the second writer:

```
(second writer)

                             tail page
                                 |
                                 v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |--->|   |-H->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

The second writer will fail to move the tail page because it was already moved, so it will try again and add its data to the new tail page. It will return to the first writer:

```
(first writer)

                             tail page
                                 |
                                 v
    +---+    +---+    +---+    +---+
```

(continues on next page)

(continued from previous page)

```
<---|   |--->|   |-U->|   |--->|   |-H->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

The first writer cannot know atomically if the tail page moved while it updates the HEAD page. It will then update the head page to what it thinks is the new head page:

```
(first writer)

                             tail page
                                 |
                                 v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |-H->|   |-H->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

Since the cmpxchg returns the old value of the pointer the first writer will see it succeeded in updating the pointer from NORMAL to HEAD. But as we can see, this is not good enough. It must also check to see if the tail page is either where it use to be or on the next page:

```
(first writer)

               A        B    tail page
               |        |        |
               v        v        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |-H->|   |-H->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

If tail page != A and tail page != B, then it must reset the pointer back to NORMAL. The fact that it only needs to worry about nested writers means that it only needs to check this after setting the HEAD page:

```
(first writer)

               A        B    tail page
               |        |        |
               v        v        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |-U->|   |--->|   |-H->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

Now the writer can update the head page. This is also why the head page must remain in UPDATE and only reset by the outermost writer. This prevents the reader from seeing the incorrect head page:

```
(first writer)

              A         B    tail page
              |         |        |
              v         v        v
    +---+    +---+    +---+    +---+
<---|   |--->|   |--->|   |--->|   |-H->
--->|   |<---|   |<---|   |<---|   |<---
    +---+    +---+    +---+    +---+
```

CHAPTER

TWENTYONE

SYSTEM TRACE MODULE

System Trace Module (STM) is a device described in MIPI STP specs as STP trace stream generator. STP (System Trace Protocol) is a trace protocol multiplexing data from multiple trace sources, each one of which is assigned a unique pair of master and channel. While some of these masters and channels are statically allocated to certain hardware trace sources, others are available to software. Software trace sources are usually free to pick for themselves any master/channel combination from this pool.

On the receiving end of this STP stream (the decoder side), trace sources can only be identified by master/channel combination, so in order for the decoder to be able to make sense of the trace that involves multiple trace sources, it needs to be able to map those master/channel pairs to the trace sources that it understands.

For instance, it is helpful to know that syslog messages come on master 7 channel 15, while arbitrary user applications can use masters 48 to 63 and channels 0 to 127.

To solve this mapping problem, stm class provides a policy management mechanism via configfs, that allows defining rules that map string identifiers to ranges of masters and channels. If these rules (policy) are consistent with what decoder expects, it will be able to properly process the trace data.

This policy is a tree structure containing rules (policy_node) that have a name (string identifier) and a range of masters and channels associated with it, located in "stp-policy" subsystem directory in configfs. The topmost directory's name (the policy) is formatted as the STM device name to which this policy applies and an arbitrary string identifier separated by a stop. From the example above, a rule may look like this:

```
$ ls /config/stp-policy/dummy_stm.my-policy/user
channels masters
$ cat /config/stp-policy/dummy_stm.my-policy/user/masters
48 63
$ cat /config/stp-policy/dummy_stm.my-policy/user/channels
0 127
```

which means that the master allocation pool for this rule consists of masters 48 through 63 and channel allocation pool has channels 0 through 127 in it. Now, any producer (trace source) identifying itself with "user" identification string will be allocated a master and channel from within these ranges.

These rules can be nested, for example, one can define a rule "dummy" under "user" directory from the example above and this new rule will be used for trace sources with the id string of "user/dummy".

Trace sources have to open the stm class device's node and write their trace data into its file descriptor.

In order to find an appropriate policy node for a given trace source, several mechanisms can be used. First, a trace source can explicitly identify itself by calling an STP_POLICY_ID_SET ioctl on the character device's file descriptor, providing their id string, before they write any data there. Secondly, if they chose not to perform the explicit identification (because you may not want to patch existing software to do this), they can just start writing the data, at which point the stm core will try to find a policy node with the name matching the task's name (e.g., "syslogd") and if one exists, it will be used. Thirdly, if the task name can't be found among the policy nodes, the catch-all entry "default" will be used, if it exists. This entry also needs to be created and configured by the system administrator or whatever tools are taking care of the policy configuration. Finally, if all the above steps failed, the write() to an stm file descriptor will return a error (EINVAL).

Previously, if no policy nodes were found for a trace source, the stm class would silently fall back to allocating the first available contiguous range of master/channels from the beginning of the device's master/channel range. The new requirement for a policy node to exist will help programmers and sysadmins identify gaps in configuration and have better control over the un-identified sources.

Some STM devices may allow direct mapping of the channel mmio regions to userspace for zero-copy writing. One mappable page (in terms of mmu) will usually contain multiple channels' mmios, so the user will need to allocate that many channels to themselves (via the aforementioned ioctl() call) to be able to do this. That is, if your stm device's channel mmio region is 64 bytes and hardware page size is 4096 bytes, after a successful STP_POLICY_ID_SET ioctl() call with width==64, you should be able to mmap() one page on this file descriptor and obtain direct access to an mmio region for 64 channels.

Examples of STM devices are Intel(R) Trace Hub [1] and Coresight STM [2].

21.1 stm_source

For kernel-based trace sources, there is "stm_source" device class. Devices of this class can be connected and disconnected to/from stm devices at runtime via a sysfs attribute called "stm_source_link" by writing the name of the desired stm device there, for example:

```
$ echo dummy_stm.0 > /sys/class/stm_source/console/stm_source_link
```

For examples on how to use stm_source interface in the kernel, refer to stm_console, stm_heartbeat or stm_ftrace drivers.

Each stm_source device will need to assume a master and a range of channels, depending on how many channels it requires. These are allocated for the device according to the policy configuration. If there's a node in the root of the policy directory that matches the stm_source device's name (for example, "console"), this node will be used to allocate master and channel numbers. If there's no such policy node, the stm core will use the catch-all entry "default", if one exists. If neither policy nodes exist, the write() to stm_source_link will return an error.

21.2 stm_console

One implementation of this interface also used in the example above is the "stm_console" driver, which basically provides a one-way console for kernel messages over an stm device.

To configure the master/channel pair that will be assigned to this console in the STP stream, create a "console" policy entry (see the beginning of this text on how to do that). When initialized, it will consume one channel.

21.3 stm_ftrace

This is another "stm_source" device, once the stm_ftrace has been linked with an stm device, and if "function" tracer is enabled, function address and parent function address which Ftrace subsystem would store into ring buffer will be exported via the stm device at the same time.

Currently only Ftrace "function" tracer is supported.

- [1] https://software.intel.com/sites/default/files/managed/d3/3c/intel-th-developer-manual.pdf
- [2] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0444b/index.html

CHAPTER

TWENTYTWO

MIPI SYS-T OVER STP

The MIPI SyS-T protocol driver can be used with STM class devices to generate standardized trace stream. Aside from being a standard, it provides better trace source identification and timestamp correlation.

In order to use the MIPI SyS-T protocol driver with your STM device, first, you'll need CONFIG_STM_PROTO_SYS_T.

Now, you can select which protocol driver you want to use when you create a policy for your STM device, by specifying it in the policy name:

mkdir /config/stp-policy/dummy_stm.0:p_sys-t.my-policy/

In other words, the policy name format is extended like this:

<device_name>:<protocol_name>.<policy_name>

With Intel TH, therefore it can look like "0-sth:p_sys-t.my-policy".

If the protocol name is omitted, the STM class will chose whichever protocol driver was loaded first.

You can also double check that everything is working as expected by

cat /config/stp-policy/dummy_stm.0:p_sys-t.my-policy/protocol p_sys-t

Now, with the MIPI SyS-T protocol driver, each policy node in the configfs gets a few additional attributes, which determine per-source parameters specific to the protocol:

mkdir /config/stp-policy/dummy_stm.0:p_sys-t.my-policy/default # ls /config/stp-policy/dummy_stm.0:p_sys-t.my-policy/default channels clocksync_interval do_len masters ts_interval uuid

The most important one here is the "uuid", which determines the UUID that will be used to tag all data coming from this source. It is automatically generated when a new node is created, but it is likely that you would want to change it.

do_len switches on/off the additional "payload length" field in the MIPI SyS-T message header. It is off by default as the STP already marks message boundaries.

ts_interval and clocksync_interval determine how much time in milliseconds can pass before we need to include a protocol (not transport, aka STP) timestamp in a message header or send a CLOCKSYNC packet, respectively.

See Documentation/ABI/testing/configfs-stp-policy-p_sys-t for more details.

- [1] https://www.mipi.org/specifications/sys-t

CHAPTER TWENTYTHREE

CORESIGHT - ARM HARDWARE TRACE

23.1 Coresight - HW Assisted Tracing on ARM

Author
Mathieu Poirier <mathieu.poirier@linaro.org>

Date
September 11th, 2014

23.1.1 Introduction

Coresight is an umbrella of technologies allowing for the debugging of ARM based SoC. It includes solutions for JTAG and HW assisted tracing. This document is concerned with the latter.

HW assisted tracing is becoming increasingly useful when dealing with systems that have many SoCs and other components like GPU and DMA engines. ARM has developed a HW assisted tracing solution by means of different components, each being added to a design at synthesis time to cater to specific tracing needs. Components are generally categorised as source, link and sinks and are (usually) discovered using the AMBA bus.

"Sources" generate a compressed stream representing the processor instruction path based on tracing scenarios as configured by users. From there the stream flows through the coresight system (via ATB bus) using links that are connecting the emanating source to a sink(s). Sinks serve as endpoints to the coresight implementation, either storing the compressed stream in a memory buffer or creating an interface to the outside world where data can be transferred to a host without fear of filling up the onboard coresight memory buffer.

At typical coresight system would look like this:

```
 ******************************************************************
**************************** AMBA AXI
****************************===||
 ******************************************************************
  ||
      ^                    ^                            |
  ||
      |                    |                            *
```

(continues on next page)

(continued from previous page)

```
↪  **
   0000000    :::::     0000000    :::::    :::::    @@@@@@@
↪||||||||||||
   0 CPU 0<-->: C :     0 CPU 0<-->: C :    : C :    @ STM @    ||
↪System ||
 |->0000000    : T :  |->0000000    : T :    : T :<--->@@@@@     ||
↪Memory ||
 |  #######<-->: I :  |  #######<-->: I :    : I :      @@@<-|
↪||||||||||||
 |  # ETM #    :::::  |  # PTM #    :::::    :::::       @   |
 |   #####      ^ ^   |   #####      ^ !      ^ !        .   |
↪|||||||||
 | |->###       | !   | |->###       | !      | !        .   |   ||
↪DAP ||
 | |   #        | !   | |   #        | !      | !        .   |
↪|||||||||
 | |   .        | !   | |   .        | !      | !        .   |
↪|  |
 | |   .        | !   | |   .        | !      | !        .   |
↪|  *
 | |   .        | !   | |   .        | !      | !        .   |
↪| SWD/
 | |   .        | !   | |   .        | !      | !        .   |
↪| JTAG
 *****************************************************************<-
↪|
*************************** AMBA Debug APB ************************
 *****************************************************************
  |    .          !         .          !        !        .    |
  |    .          *         .          *        *        .    |
 *****************************************************************
******************** Cross Trigger Matrix (CTM) ******************
 *****************************************************************
  |    .     ^              .                            .    |
  |    *     !              *                            *    |
 *****************************************************************
****************** AMBA Advanced Trace Bus (ATB) ******************
 *****************************************************************
  |          !                        ===============         |
  |          *                         ===== F =====<---------|
  |   :::::::::                         ==== U ====
  |-->:: CTI ::<!!                       === N ===
  |   :::::::::  !                        == N ==
  |    ^         *                        == E ==
  |    !  &&&&&&&&&       IIIIIII         == L ==
  |------>&& ETB &&<......II     I        =======
  |    !  &&&&&&&&&       II     I           .
  |    !                    I     I          .
  |    !                    I REP I<.........
```

(continues on next page)

(continued from previous page)

```
  |    !                    I     I
  |    !!>&&&&&&&&&       II     I          *Source: ARM ltd.
  |------>& TPIU  &<......II    I           DAP = Debug Access Port
          &&&&&&&&&       IIIIIII           ETM = Embedded Trace␣
↪Macrocell
                ;                           PTM = Program Trace␣
↪Macrocell
                ;                           CTI = Cross Trigger␣
↪Interface
                *                           ETB = Embedded Trace␣
↪Buffer
             To trace port                  TPIU= Trace Port␣
↪Interface Unit
                                            SWD = Serial Wire Debug
```

While on target configuration of the components is done via the APB bus, all trace data are carried out-of-band on the ATB bus. The CTM provides a way to aggregate and distribute signals between CoreSight components.

The coresight framework provides a central point to represent, configure and manage coresight devices on a platform. This first implementation centers on the basic tracing functionality, enabling components such ETM/PTM, funnel, replicator, TMC, TPIU and ETB. Future work will enable more intricate IP blocks such as STM and CTI.

23.1.2 Acronyms and Classification

Acronyms:

PTM:
    Program Trace Macrocell

ETM:
    Embedded Trace Macrocell

STM:
    System trace Macrocell

ETB:
    Embedded Trace Buffer

ITM:
    Instrumentation Trace Macrocell

TPIU:
    Trace Port Interface Unit

TMC-ETR:
    Trace Memory Controller, configured as Embedded Trace Router

TMC-ETF:
    Trace Memory Controller, configured as Embedded Trace FIFO

CTI:
Cross Trigger Interface

Classification:

Source:
ETMv3.x ETMv4, PTMv1.0, PTMv1.1, STM, STM500, ITM

Link:
Funnel, replicator (intelligent or not), TMC-ETR

Sinks:
ETBv1.0, ETB1.1, TPIU, TMC-ETF

Misc:
CTI

23.1.3 Device Tree Bindings

See Documentation/devicetree/bindings/arm/coresight.txt for details.

As of this writing drivers for ITM, STMs and CTIs are not provided but are expected to be added as the solution matures.

23.1.4 Framework and implementation

The coresight framework provides a central point to represent, configure and manage coresight devices on a platform. Any coresight compliant device can register with the framework for as long as they use the right APIs:

struct coresight_device ***coresight_register**(struct coresight_desc *desc);

void **coresight_unregister**(struct coresight_device *csdev);

The registering function is taking a `struct coresight_desc *desc` and register the device with the core framework. The unregister function takes a reference to a `struct coresight_device *csdev` obtained at registration time.

If everything goes well during the registration process the new devices will show up under /sys/bus/coresight/devices, as showns here for a TC2 platform:

```
root:~# ls /sys/bus/coresight/devices/
replicator  20030000.tpiu    2201c000.ptm  2203c000.etm  2203e000.
↪etm
20010000.etb         20040000.funnel  2201d000.ptm  2203d000.etm
root:~#
```

The functions take a `struct coresight_device`, which looks like this:

```
struct coresight_desc {
        enum coresight_dev_type type;
        struct coresight_dev_subtype subtype;
        const struct coresight_ops *ops;
        struct coresight_platform_data *pdata;
```

(continues on next page)

(continued from previous page)

```
        struct device *dev;
        const struct attribute_group **groups;
};
```

The "coresight_dev_type" identifies what the device is, i.e, source link or sink while the "coresight_dev_subtype" will characterise that type further.

The struct coresight_ops is mandatory and will tell the framework how to perform base operations related to the components, each component having a different set of requirement. For that struct coresight_ops_sink, struct coresight_ops_link and struct coresight_ops_source have been provided.

The next field struct coresight_platform_data *pdata is acquired by calling of_get_coresight_platform_data(), as part of the driver's _probe routine and struct device *dev gets the device reference embedded in the amba_device:

```
static int etm_probe(struct amba_device *adev, const struct amba_id␣
↪*id)
{
 ...
 ...
 drvdata->dev = &adev->dev;
 ...
}
```

Specific class of device (source, link, or sink) have generic operations that can be performed on them (see struct coresight_ops). The **groups is a list of sysfs entries pertaining to operations specific to that component only. "Implementation defined" customisations are expected to be accessed and controlled using those entries.

23.1.5 Device Naming scheme

The devices that appear on the "coresight" bus were named the same as their parent devices, i.e, the real devices that appears on AMBA bus or the platform bus. Thus the names were based on the Linux Open Firmware layer naming convention, which follows the base physical address of the device followed by the device type. e.g:

```
root:~# ls /sys/bus/coresight/devices/
 20010000.etf  20040000.funnel      20100000.stm     22040000.etm
 22140000.etm  230c0000.funnel      23240000.etm     20030000.tpiu
 20070000.etr  20120000.replicator  220c0000.funnel
 23040000.etm  23140000.etm         23340000.etm
```

However, with the introduction of ACPI support, the names of the real devices are a bit cryptic and non-obvious. Thus, a new naming scheme was introduced to use more generic names based on the type of the device. The following rules apply:

```
1) Devices that are bound to CPUs, are named based on the CPU␣
↪logical
   number.

   e.g, ETM bound to CPU0 is named "etm0"

2) All other devices follow a pattern, "<device_type_prefix>N",␣
↪where :

      <device_type_prefix>    - A prefix specific to the type of␣
↪the device
      N                       - a sequential number assigned based␣
↪on the order
                                of probing.

      e.g, tmc_etf0, tmc_etr0, funnel0, funnel1
```

Thus, with the new scheme the devices could appear as

```
root:~# ls /sys/bus/coresight/devices/
 etm0     etm1     etm2         etm3  etm4      etm5      funnel0
 funnel1  funnel2  replicator0  stm0  tmc_etf0  tmc_etr0  tpiu0
```

Some of the examples below might refer to old naming scheme and some to the newer scheme, to give a confirmation that what you see on your system is not unexpected. One must use the "names" as they appear on the system under specified locations.

23.1.6 Topology Representation

Each CoreSight component has a `connections` directory which will contain links to other CoreSight components. This allows the user to explore the trace topology and for larger systems, determine the most appropriate sink for a given source. The connection information can also be used to establish which CTI devices are connected to a given component. This directory contains a `nr_links` attribute detailing the number of links in the directory.

For an ETM source, in this case `etm0` on a Juno platform, a typical arrangement will be:

```
linaro-developer:~# ls - l /sys/bus/coresight/devices/etm0/
↪connections
<file details>  cti_cpu0 -> ../../../23020000.cti/cti_cpu0
<file details>  nr_links
<file details>  out:0 -> ../../../230c0000.funnel/funnel2
```

Following the out port to `funnel2`:

```
linaro-developer:~# ls -l /sys/bus/coresight/devices/funnel2/
↪connections
```

(continues on next page)

(continued from previous page)

```
<file details> in:0 -> ../../../23040000.etm/etm0
<file details> in:1 -> ../../../23140000.etm/etm3
<file details> in:2 -> ../../../23240000.etm/etm4
<file details> in:3 -> ../../../23340000.etm/etm5
<file details> nr_links
<file details> out:0 -> ../../../20040000.funnel/funnel0
```

And again to `funnel0`:

```
linaro-developer:~# ls -l /sys/bus/coresight/devices/funnel0/
↪connections
<file details> in:0 -> ../../../220c0000.funnel/funnel1
<file details> in:1 -> ../../../230c0000.funnel/funnel2
<file details> nr_links
<file details> out:0 -> ../../../20010000.etf/tmc_etf0
```

Finding the first sink `tmc_etf0`. This can be used to collect data as a sink, or as a link to propagate further along the chain:

```
linaro-developer:~# ls -l /sys/bus/coresight/devices/tmc_etf0/
↪connections
<file details> cti_sys0 -> ../../../20020000.cti/cti_sys0
<file details> in:0 -> ../../../20040000.funnel/funnel0
<file details> nr_links
<file details> out:0 -> ../../../20150000.funnel/funnel4
```

via `funnel4`:

```
linaro-developer:~# ls -l /sys/bus/coresight/devices/funnel4/
↪connections
<file details> in:0 -> ../../../20010000.etf/tmc_etf0
<file details> in:1 -> ../../../20140000.etf/tmc_etf1
<file details> nr_links
<file details> out:0 -> ../../../20120000.replicator/replicator0
```

and a `replicator0`:

```
linaro-developer:~# ls -l /sys/bus/coresight/devices/replicator0/
↪connections
<file details> in:0 -> ../../../20150000.funnel/funnel4
<file details> nr_links
<file details> out:0 -> ../../../20030000.tpiu/tpiu0
<file details> out:1 -> ../../../20070000.etr/tmc_etr0
```

Arriving at the final sink in the chain, `tmc_etr0`:

```
linaro-developer:~# ls -l /sys/bus/coresight/devices/tmc_etr0/
↪connections
<file details> cti_sys0 -> ../../../20020000.cti/cti_sys0
<file details> in:0 -> ../../../20120000.replicator/replicator0
<file details> nr_links
```

As described below, when using sysfs it is sufficient to enable a sink and a source for successful trace. The framework will correctly enable all intermediate links as required.

Note: `cti_sys0` appears in two of the connections lists above. CTIs can connect to multiple devices and are arranged in a star topology via the CTM. See (*CoreSight Embedded Cross Trigger (CTI & CTM).*)[4] for further details. Looking at this device we see 4 connections:

```
linaro-developer:~# ls -l /sys/bus/coresight/devices/cti_sys0/
↪connections
<file details> nr_links
<file details> stm0 -> ../../../20100000.stm/stm0
<file details> tmc_etf0 -> ../../../20010000.etf/tmc_etf0
<file details> tmc_etr0 -> ../../../20070000.etr/tmc_etr0
<file details> tpiu0 -> ../../../20030000.tpiu/tpiu0
```

23.1.7 How to use the tracer modules

There are two ways to use the Coresight framework:

1. using the perf cmd line tools.
2. interacting directly with the Coresight devices using the sysFS interface.

Preference is given to the former as using the sysFS interface requires a deep understanding of the Coresight HW. The following sections provide details on using both methods.

1) Using the sysFS interface:

Before trace collection can start, a coresight sink needs to be identified. There is no limit on the amount of sinks (nor sources) that can be enabled at any given moment. As a generic operation, all device pertaining to the sink class will have an “active” entry in sysfs:

```
root:/sys/bus/coresight/devices# ls
replicator  20030000.tpiu    2201c000.ptm  2203c000.etm  2203e000.
↪etm
20010000.etb         20040000.funnel  2201d000.ptm  2203d000.etm
root:/sys/bus/coresight/devices# ls 20010000.etb
enable_sink  status  trigger_cntr
root:/sys/bus/coresight/devices# echo 1 > 20010000.etb/enable_sink
root:/sys/bus/coresight/devices# cat 20010000.etb/enable_sink
1
root:/sys/bus/coresight/devices#
```

At boot time the current etm3x driver will configure the first address comparator with “_stext” and “_etext”, essentially tracing any instruction that falls within that range. As such “enabling” a source will immediately trigger a trace capture:

[4] *CoreSight Embedded Cross Trigger (CTI & CTM).*

```
root:/sys/bus/coresight/devices# echo 1 > 2201c000.ptm/enable_source
root:/sys/bus/coresight/devices# cat 2201c000.ptm/enable_source
1
root:/sys/bus/coresight/devices# cat 20010000.etb/status
Depth:          0x2000
Status:         0x1
RAM read ptr:   0x0
RAM wrt ptr:    0x19d3   <----- The write pointer is moving
Trigger cnt:    0x0
Control:        0x1
Flush status:   0x0
Flush ctrl:     0x2001
root:/sys/bus/coresight/devices#
```

Trace collection is stopped the same way:

```
root:/sys/bus/coresight/devices# echo 0 > 2201c000.ptm/enable_source
root:/sys/bus/coresight/devices#
```

The content of the ETB buffer can be harvested directly from /dev:

```
root:/sys/bus/coresight/devices# dd if=/dev/20010000.etb \
of=~/cstrace.bin
64+0 records in
64+0 records out
32768 bytes (33 kB) copied, 0.00125258 s, 26.2 MB/s
root:/sys/bus/coresight/devices#
```

The file cstrace.bin can be decompressed using "ptm2human", DS-5 or Trace32.

Following is a DS-5 output of an experimental loop that increments a variable up to a certain value. The example is simple and yet provides a glimpse of the wealth of possibilities that coresight provides.

```
Info                                    Tracing enabled
Instruction     106378866       0x8026B53C      E52DE004        ␣
↪false   PUSH     {lr}
Instruction     0       0x8026B540      E24DD00C        false   SUB␣
↪       sp,sp,#0xc
Instruction     0       0x8026B544      E3A03000        false   MOV␣
↪       r3,#0
Instruction     0       0x8026B548      E58D3004        false   STR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B54C      E59D3004        false   LDR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B550      E3530004        false   CMP␣
↪       r3,#4
Instruction     0       0x8026B554      E2833001        false   ADD␣
↪       r3,r3,#1
Instruction     0       0x8026B558      E58D3004        false   STR␣
↪       r3,[sp,#4]
```

(continues on next page)

(continued from previous page)

```
Instruction     0       0x8026B55C      DAFFFFFA        true    BLE␣
↪       {pc}-0x10 ; 0x8026b54c
Timestamp                                       Timestamp:␣
↪17106715833
Instruction     319     0x8026B54C      E59D3004        false   LDR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B550      E3530004        false   CMP␣
↪       r3,#4
Instruction     0       0x8026B554      E2833001        false   ADD␣
↪       r3,r3,#1
Instruction     0       0x8026B558      E58D3004        false   STR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B55C      DAFFFFFA        true    BLE␣
↪       {pc}-0x10 ; 0x8026b54c
Instruction     9       0x8026B54C      E59D3004        false   LDR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B550      E3530004        false   CMP␣
↪       r3,#4
Instruction     0       0x8026B554      E2833001        false   ADD␣
↪       r3,r3,#1
Instruction     0       0x8026B558      E58D3004        false   STR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B55C      DAFFFFFA        true    BLE␣
↪       {pc}-0x10 ; 0x8026b54c
Instruction     7       0x8026B54C      E59D3004        false   LDR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B550      E3530004        false   CMP␣
↪       r3,#4
Instruction     0       0x8026B554      E2833001        false   ADD␣
↪       r3,r3,#1
Instruction     0       0x8026B558      E58D3004        false   STR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B55C      DAFFFFFA        true    BLE␣
↪       {pc}-0x10 ; 0x8026b54c
Instruction     7       0x8026B54C      E59D3004        false   LDR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B550      E3530004        false   CMP␣
↪       r3,#4
Instruction     0       0x8026B554      E2833001        false   ADD␣
↪       r3,r3,#1
Instruction     0       0x8026B558      E58D3004        false   STR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B55C      DAFFFFFA        true    BLE␣
↪       {pc}-0x10 ; 0x8026b54c
Instruction     10      0x8026B54C      E59D3004        false   LDR␣
↪       r3,[sp,#4]
Instruction     0       0x8026B550      E3530004        false   CMP␣
↪       r3,#4
Instruction     0       0x8026B554      E2833001        false   ADD␣
```

(continues on next page)

(continued from previous page)

```
↪     r3,r3,#1
Instruction     0       0x8026B558      E58D3004        false   STR␣
↪     r3,[sp,#4]
Instruction     0       0x8026B55C      DAFFFFFA        true    BLE␣
↪     {pc}-0x10 ; 0x8026b54c
Instruction     6       0x8026B560      EE1D3F30        false   MRC␣
↪     p15,#0x0,r3,c13,c0,#1
Instruction     0       0x8026B564      E1A0100D        false   MOV␣
↪     r1,sp
Instruction     0       0x8026B568      E3C12D7F        false   BIC␣
↪     r2,r1,#0x1fc0
Instruction     0       0x8026B56C      E3C2203F        false   BIC␣
↪     r2,r2,#0x3f
Instruction     0       0x8026B570      E59D1004        false   LDR␣
↪     r1,[sp,#4]
Instruction     0       0x8026B574      E59F0010        false   LDR␣
↪     r0,[pc,#16] ; [0x8026B58C] = 0x80550368
Instruction     0       0x8026B578      E592200C        false   LDR␣
↪     r2,[r2,#0xc]
Instruction     0       0x8026B57C      E59221D0        false   LDR␣
↪     r2,[r2,#0x1d0]
Instruction     0       0x8026B580      EB07A4CF        true    BL ␣
↪     {pc}+0x1e9344 ; 0x804548c4
Info                                    Tracing enabled
Instruction     13570831        0x8026B584      E28DD00C        ␣
↪false   ADD      sp,sp,#0xc
Instruction     0       0x8026B588      E8BD8000        true    LDM␣
↪     sp!,{pc}
Timestamp                                       Timestamp:␣
↪17107041535
```

2) Using perf framework:

Coresight tracers are represented using the Perf framework’ s Performance Monitoring Unit (PMU) abstraction. As such the perf framework takes charge of controlling when tracing gets enabled based on when the process of interest is scheduled. When configured in a system, Coresight PMUs will be listed when queried by the perf command line tool:

> linaro@linaro-nano:~$ ./perf list pmu
>
> > List of pre-defined events (to be used in -e):
> >
> > cs_etm// [Kernel PMU event]
>
> linaro@linaro-nano:~$

Regardless of the number of tracers available in a system (usually equal to the amount of processor cores), the “cs_etm” PMU will be listed only once.

A Coresight PMU works the same way as any other PMU, i.e the name of the PMU is listed along with configuration options within forward slashes ‘/’ . Since a Coresight system will typically have more than one sink, the name of the sink to work with needs to be specified as an event option. On newer kernels the available sinks are listed in sysFS under ($SYSFS)/bus/event_source/devices/cs_etm/sinks/:

```
root@localhost:/sys/bus/event_source/devices/cs_etm/sinks# ls
tmc_etf0  tmc_etr0  tpiu0
```

On older kernels, this may need to be found from the list of coresight devices, available under ($SYSFS)/bus/coresight/devices/:

```
root:~# ls /sys/bus/coresight/devices/
 etm0     etm1     etm2         etm3  etm4      etm5      funnel0
 funnel1  funnel2  replicator0  stm0  tmc_etf0  tmc_etr0  tpiu0
root@linaro-nano:~# perf record -e cs_etm/@tmc_etr0/u --per-thread␣
↪program
```

As mentioned above in section "Device Naming scheme", the names of the devices could look different from what is used in the example above. One must use the device names as it appears under the sysFS.

The syntax within the forward slashes '/' is important. The '@' character tells the parser that a sink is about to be specified and that this is the sink to use for the trace session.

More information on the above and other example on how to use Coresight with the perf tools can be found in the "HOWTO.md" file of the openCSD gitHub repository[3].

2.1) AutoFDO analysis using the perf tools:

perf can be used to record and analyze trace of programs.

Execution can be recorded using 'perf record' with the cs_etm event, specifying the name of the sink to record to, e.g:

```
perf record -e cs_etm/@tmc_etr0/u --per-thread
```

The 'perf report' and 'perf script' commands can be used to analyze execution, synthesizing instruction and branch events from the instruction trace. 'perf inject' can be used to replace the trace data with the synthesized events. The –itrace option controls the type and frequency of synthesized events (see perf documentation).

Note that only 64-bit programs are currently supported - further work is required to support instruction decode of 32-bit Arm programs.

23.1.8 Generating coverage files for Feedback Directed Optimization: AutoFDO

'perf inject' accepts the –itrace option in which case tracing data is removed and replaced with the synthesized events. e.g.

```
perf inject --itrace --strip -i perf.data -o perf.data.new
```

[3] https://github.com/Linaro/perf-opencsd

Below is an example of using ARM ETM for autoFDO. It requires autofdo (https://github.com/google/autofdo) and gcc version 5. The bubble sort example is from the AutoFDO tutorial (https://gcc.gnu.org/wiki/AutoFDO/Tutorial).

```
$ gcc-5 -O3 sort.c -o sort
$ taskset -c 2 ./sort
Bubble sorting array of 30000 elements
5910 ms

$ perf record -e cs_etm/@tmc_etr0/u --per-thread taskset -c 2 ./sort
Bubble sorting array of 30000 elements
12543 ms
[ perf record: Woken up 35 times to write data ]
[ perf record: Captured and wrote 69.640 MB perf.data ]

$ perf inject -i perf.data -o inj.data --itrace=il64 --strip
$ create_gcov --binary=./sort --profile=inj.data --gcov=sort.gcov -
↪gcov_version=1
$ gcc-5 -O3 -fauto-profile=sort.gcov sort.c -o sort_autofdo
$ taskset -c 2 ./sort_autofdo
Bubble sorting array of 30000 elements
5806 ms
```

23.1.9 How to use the STM module

Using the System Trace Macrocell module is the same as the tracers - the only difference is that clients are driving the trace capture rather than the program flow through the code.

As with any other CoreSight component, specifics about the STM tracer can be found in sysfs with more information on each entry being found in[1]:

```
root@genericarmv8:~# ls /sys/bus/coresight/devices/stm0
enable_source   hwevent_select  port_enable     subsystem      ␣
↪uevent
hwevent_enable  mgmt            port_select     traceid
root@genericarmv8:~#
```

Like any other source a sink needs to be identified and the STM enabled before being used:

```
root@genericarmv8:~# echo 1 > /sys/bus/coresight/devices/tmc_etf0/
↪enable_sink
root@genericarmv8:~# echo 1 > /sys/bus/coresight/devices/stm0/
↪enable_source
```

From there user space applications can request and use channels using the devfs interface provided for that purpose by the generic STM API:

[1] Documentation/ABI/testing/sysfs-bus-coresight-devices-stm

```
root@genericarmv8:~# ls -l /dev/stm0
crw-------    1 root     root       10,  61 Jan  3 18:11 /dev/stm0
root@genericarmv8:~#
```

Details on how to use the generic STM API can be found here:- *System Trace Module*[2].

23.1.10 The CTI & CTM Modules

The CTI (Cross Trigger Interface) provides a set of trigger signals between individual CTIs and components, and can propagate these between all CTIs via channels on the CTM (Cross Trigger Matrix).

A separate documentation file is provided to explain the use of these devices. (*CoreSight Embedded Cross Trigger (CTI & CTM).*)[Page 330, 4].

23.2 Coresight CPU Debug Module

Author
Leo Yan <leo.yan@linaro.org>

Date
April 5th, 2017

23.2.1 Introduction

Coresight CPU debug module is defined in ARMv8-a architecture reference manual (ARM DDI 0487A.k) Chapter 'Part H: External debug', the CPU can integrate debug module and it is mainly used for two modes: self-hosted debug and external debug. Usually the external debug mode is well known as the external debugger connects with SoC from JTAG port; on the other hand the program can explore debugging method which rely on self-hosted debug mode, this document is to focus on this part.

The debug module provides sample-based profiling extension, which can be used to sample CPU program counter, secure state and exception level, etc; usually every CPU has one dedicated debug module to be connected. Based on self-hosted debug mechanism, Linux kernel can access these related registers from mmio region when the kernel panic happens. The callback notifier for kernel panic will dump related registers for every CPU; finally this is good for assistant analysis for panic.

[2] *System Trace Module*

23.2.2 Implementation

- During driver registration, it uses EDDEVID and EDDEVID1 - two device ID registers to decide if sample-based profiling is implemented or not. On some platforms this hardware feature is fully or partially implemented; and if this feature is not supported then registration will fail.
- At the time this documentation was written, the debug driver mainly relies on information gathered by the kernel panic callback notifier from three sampling registers: EDPCSR, EDVIDSR and EDCIDSR: from EDPCSR we can get program counter; EDVIDSR has information for secure state, exception level, bit width, etc; EDCIDSR is context ID value which contains the sampled value of CONTEXTIDR_EL1.
- The driver supports a CPU running in either AArch64 or AArch32 mode. The registers naming convention is a bit different between them, AArch64 uses 'ED' for register prefix (ARM DDI 0487A.k, chapter H9.1) and AArch32 uses 'DBG' as prefix (ARM DDI 0487A.k, chapter G5.1). The driver is unified to use AArch64 naming convention.
- ARMv8-a (ARM DDI 0487A.k) and ARMv7-a (ARM DDI 0406C.b) have different register bits definition. So the driver consolidates two difference:

  If PCSROffset=0b0000, on ARMv8-a the feature of EDPCSR is not implemented; but ARMv7-a defines "PCSR samples are offset by a value that depends on the instruction set state". For ARMv7-a, the driver checks furthermore if CPU runs with ARM or thumb instruction set and calibrate PCSR value, the detailed description for offset is in ARMv7-a ARM (ARM DDI 0406C.b) chapter C11.11.34 "DBGPCSR, Program Counter Sampling Register".

  If PCSROffset=0b0010, ARMv8-a defines "EDPCSR implemented, and samples have no offset applied and do not sample the instruction set state in AArch32 state". So on ARMv8 if EDDEVID1.PCSROffset is 0b0010 and the CPU operates in AArch32 state, EDPCSR is not sampled; when the CPU operates in AArch64 state EDPCSR is sampled and no offset are applied.

23.2.3 Clock and power domain

Before accessing debug registers, we should ensure the clock and power domain have been enabled properly. In ARMv8-a ARM (ARM DDI 0487A.k) chapter 'H9.1 Debug registers', the debug registers are spread into two domains: the debug domain and the CPU domain.

```
                              +---------------+
                              |               |
                              |               |
                   +----------+--+            |
      dbg_clock -->|          |**|            |<-- cpu_clock
                   |    Debug |**|   CPU      |
dbg_power_domain -->|         |**|            |<-- cpu_power_domain
                   +----------+--+            |
```

(continues on next page)

(continued from previous page)

```
                                         |                   |
                                         |                   |
                                         +---------------+
```

For debug domain, the user uses DT binding "clocks" and "power-domains" to specify the corresponding clock source and power supply for the debug logic. The driver calls the pm_runtime_{put|get} operations as needed to handle the debug power domain.

For CPU domain, the different SoC designs have different power management schemes and finally this heavily impacts external debug module. So we can divide into below cases:

- On systems with a sane power controller which can behave correctly with respect to CPU power domain, the CPU power domain can be controlled by register EDPRCR in driver. The driver firstly writes bit EDPRCR.COREPURQ to power up the CPU, and then writes bit EDPRCR.CORENPDRQ for emulation of CPU power down. As result, this can ensure the CPU power domain is powered on properly during the period when access debug related registers;
- Some designs will power down an entire cluster if all CPUs on the cluster are powered down - including the parts of the debug registers that should remain powered in the debug power domain. The bits in EDPRCR are not respected in these cases, so these designs do not support debug over power down in the way that the CoreSight / Debug designers anticipated. This means that even checking EDPRSR has the potential to cause a bus hang if the target register is unpowered.

  In this case, accessing to the debug registers while they are not powered is a recipe for disaster; so we need preventing CPU low power states at boot time or when user enable module at the run time. Please see chapter "How to use the module" for detailed usage info for this.

23.2.4 Device Tree Bindings

See Documentation/devicetree/bindings/arm/coresight-cpu-debug.txt for details.

23.2.5 How to use the module

If you want to enable debugging functionality at boot time, you can add "coresight_cpu_debug.enable=1" to the kernel command line parameter.

The driver also can work as module, so can enable the debugging when insmod module:

```
# insmod coresight_cpu_debug.ko debug=1
```

When boot time or insmod module you have not enabled the debugging, the driver uses the debugfs file system to provide a knob to dynamically enable or disable debugging:

To enable it, write a '1' into /sys/kernel/debug/coresight_cpu_debug/enable:

```
# echo 1 > /sys/kernel/debug/coresight_cpu_debug/enable
```

To disable it, write a '0' into /sys/kernel/debug/coresight_cpu_debug/enable:

```
# echo 0 > /sys/kernel/debug/coresight_cpu_debug/enable
```

As explained in chapter "Clock and power domain", if you are working on one platform which has idle states to power off debug logic and the power controller cannot work well for the request from EDPRCR, then you should firstly constraint CPU idle states before enable CPU debugging feature; so can ensure the accessing to debug logic.

If you want to limit idle states at boot time, you can use "nohlt" or "cpuidle.off=1" in the kernel command line.

At the runtime you can disable idle states with below methods:

It is possible to disable CPU idle states by way of the PM QoS subsystem, more specifically by using the "/dev/cpu_dma_latency" interface (see Documentation/power/pm_qos_interface.rst for more details). As specified in the PM QoS documentation the requested parameter will stay in effect until the file descriptor is released. For example:

```
# exec 3<> /dev/cpu_dma_latency; echo 0 >&3
...
Do some work...
...
# exec 3<>-
```

The same can also be done from an application program.

Disable specific CPU's specific idle state from cpuidle sysfs (see Documentation/admin-guide/pm/cpuidle.rst):

```
# echo 1 > /sys/devices/system/cpu/cpu$cpu/cpuidle/state$state/
↪disable
```

23.2.6 Output format

Here is an example of the debugging output format:

```
ARM external debug module:
coresight-cpu-debug 850000.debug: CPU[0]:
coresight-cpu-debug 850000.debug:  EDPRSR:  00000001 (Power:On␣
↪DLK:Unlock)
coresight-cpu-debug 850000.debug:  EDPCSR:  handle_IPI+0x174/0x1d8
coresight-cpu-debug 850000.debug:  EDCIDSR: 00000000
coresight-cpu-debug 850000.debug:  EDVIDSR: 90000000 (State:Non-
↪secure Mode:EL1/0 Width:64bits VMID:0)
coresight-cpu-debug 852000.debug: CPU[1]:
coresight-cpu-debug 852000.debug:  EDPRSR:  00000001 (Power:On␣
↪DLK:Unlock)
```

(continues on next page)

(continued from previous page)

```
coresight-cpu-debug 852000.debug:  EDPCSR:  debug_notifier_
↪call+0x23c/0x358
coresight-cpu-debug 852000.debug:  EDCIDSR: 00000000
coresight-cpu-debug 852000.debug:  EDVIDSR: 90000000 (State:Non-
↪secure Mode:EL1/0 Width:64bits VMID:0)
```

23.3 CoreSight Embedded Cross Trigger (CTI & CTM).

Author
Mike Leach <mike.leach@linaro.org>

Date
November 2019

23.3.1 Hardware Description

The CoreSight Cross Trigger Interface (CTI) is a hardware device that takes individual input and output hardware signals known as triggers to and from devices and interconnects them via the Cross Trigger Matrix (CTM) to other devices via numbered channels, in order to propagate events between devices.

e.g.:

```
0000000  in_trigs  :::::::
0 C   0----------->:     :            +======>(other CTI channel␣
↪IO)
0  P  0<-----------:     :            v
0   U 0  out_trigs :     : Channels  *****      :::::::
0000000            : CTI :<=========>*CTM*<====>: CTI :---+
#######  in_trigs  :     : (id 0-3)  *****      :::::::   v
# ETM #----------->:     :                         ^   #######
#     #<-----------:     :                         +---# ETR #
####### out_trigs  :::::::                             #######
```

The CTI driver enables the programming of the CTI to attach triggers to channels. When an input trigger becomes active, the attached channel will become active. Any output trigger attached to that channel will also become active. The active channel is propagated to other CTIs via the CTM, activating connected output triggers there, unless filtered by the CTI channel gate.

It is also possible to activate a channel using system software directly programming registers in the CTI.

The CTIs are registered by the system to be associated with CPUs and/or other CoreSight devices on the trace data path. When these devices are enabled the attached CTIs will also be enabled. By default/on power up the CTIs have no programmed trigger/channel attachments, so will not affect the system until explicitly programmed.

The hardware trigger connections between CTIs and devices is implementation defined, unless the CPU/ETM combination is a v8 architecture, in which case the connections have an architecturally defined standard layout.

The hardware trigger signals can also be connected to non-CoreSight devices (e.g. UART), or be propagated off chip as hardware IO lines.

All the CTI devices are associated with a CTM. On many systems there will be a single effective CTM (one CTM, or multiple CTMs all interconnected), but it is possible that systems can have nets of CTIs+CTM that are not interconnected by a CTM to each other. On these systems a CTM index is declared to associate CTI devices that are interconnected via a given CTM.

23.3.2 Sysfs files and directories

The CTI devices appear on the existing CoreSight bus alongside the other CoreSight devices:

```
>$ ls /sys/bus/coresight/devices
 cti_cpu0  cti_cpu2  cti_sys0  etm0  etm2  funnel0  replicator0 ␣
↪tmc_etr0
 cti_cpu1  cti_cpu3  cti_sys1  etm1  etm3  funnel1  tmc_etf0    ␣
↪tpiu0
```

The `cti_cpu<N>` named CTIs are associated with a CPU, and any ETM used by that core. The `cti_sys<N>` CTIs are general system infrastructure CTIs that can be associated with other CoreSight devices, or other system hardware capable of generating or using trigger signals.:

```
>$ ls /sys/bus/coresight/devices/etm0/cti_cpu0
channels  ctmid  enable  nr_trigger_cons mgmt  power powered  regs
connections subsystem triggers0 triggers1  uevent
```

Key file items are:-

- `enable`: enables/disables the CTI. Read to determine current state. If this shows as enabled (1), but `powered` shows unpowered (0), then the enable indicates a request to enabled when the device is powered.
- `ctmid` : associated CTM - only relevant if system has multiple CTI+CTM clusters that are not interconnected.
- `nr_trigger_cons` : total connections - triggers<N> directories.
- `powered` : Read to determine if the CTI is currently powered.

Sub-directories:-

- `triggers<N>`: contains list of triggers for an individual connection.
- `channels`: Contains the channel API - CTI main programming interface.
- `regs`: Gives access to the raw programmable CTI regs.
- `mgmt`: the standard CoreSight management registers.

- `connections`: Links to connected *CoreSight* devices. The number of links can be 0 to `nr_trigger_cons`. Actual number given by `nr_links` in this directory.

triggers<N> directories

Individual trigger connection information. This describes trigger signals for CoreSight and non-CoreSight connections.

Each triggers directory has a set of parameters describing the triggers for the connection.

- `name` : name of connection
- `in_signals` : input trigger signal indexes used in this connection.
- `in_types` : functional types for in signals.
- `out_signals` : output trigger signals for this connection.
- `out_types` : functional types for out signals.

e.g:

```
>$ ls ./cti_cpu0/triggers0/
in_signals  in_types  name  out_signals  out_types
>$ cat ./cti_cpu0/triggers0/name
cpu0
>$ cat ./cti_cpu0/triggers0/out_signals
0-2
>$ cat ./cti_cpu0/triggers0/out_types
pe_edbgreq pe_dbgrestart pe_ctiirq
>$ cat ./cti_cpu0/triggers0/in_signals
0-1
>$ cat ./cti_cpu0/triggers0/in_types
pe_dbgtrigger pe_pmuirq
```

If a connection has zero signals in either the 'in' or 'out' triggers then those parameters will be omitted.

Channels API Directory

This provides an easy way to attach triggers to channels, without needing the multiple register operations that are required if manipulating the 'regs' subdirectory elements directly.

A number of files provide this API:

```
>$ ls ./cti_sys0/channels/
chan_clear          chan_inuse      chan_xtrigs_out     trigin_attach
chan_free           chan_pulse      chan_xtrigs_reset   trigin_detach
chan_gate_disable   chan_set        chan_xtrigs_sel     trigout_
↪attach
chan_gate_enable    chan_xtrigs_in  trig_filter_enable  trigout_
```

(continues on next page)

(continued from previous page)

```
↪detach
trigout_filtered
```

Most access to these elements take the form:

```
echo <chan> [<trigger>] > /<device_path>/<operation>
```

where the optional <trigger> is only needed for trigXX_attach | detach operations.

e.g.:

```
>$ echo 0 1 > ./cti_sys0/channels/trigout_attach
>$ echo 0 > ./cti_sys0/channels/chan_set
```

Attaches trigout(1) to channel(0), then activates channel(0) generating a set state on cti_sys0.trigout(1)

API operations

- `trigin_attach, trigout_attach`: Attach a channel to a trigger signal.
- `trigin_detach, trigout_detach`: Detach a channel from a trigger signal.
- `chan_set`: Set the channel - the set state will be propagated around the CTM to other connected devices.
- `chan_clear`: Clear the channel.
- `chan_pulse`: Set the channel for a single CoreSight clock cycle.
- `chan_gate_enable`: Write operation sets the CTI gate to propagate (enable) the channel to other devices. This operation takes a channel number. CTI gate is enabled for all channels by default at power up. Read to list the currently enabled channels on the gate.
- `chan_gate_disable`: Write channel number to disable gate for that channel.
- `chan_inuse`: Show the current channels attached to any signal
- `chan_free`: Show channels with no attached signals.
- `chan_xtrigs_sel`: write a channel number to select a channel to view, read to show the selected channel number.
- `chan_xtrigs_in`: Read to show the input triggers attached to the selected view channel.
- `chan_xtrigs_out`:Read to show the output triggers attached to the selected view channel.
- `trig_filter_enable`: Defaults to enabled, disable to allow potentially dangerous output signals to be set.
- `trigout_filtered`: Trigger out signals that are prevented from being set if filtering `trig_filter_enable` is enabled. One use is to prevent accidental `EDBGREQ` signals stopping a core.
- `chan_xtrigs_reset`: Write 1 to clear all channel / trigger programming. Resets device hardware to default state.

The example below attaches input trigger index 1 to channel 2, and output trigger index 6 to the same channel. It then examines the state of the channel / trigger connections using the appropriate sysfs attributes.

The settings mean that if either input trigger 1, or channel 2 go active then trigger out 6 will go active. We then enable the CTI, and use the software channel control to activate channel 2. We see the active channel on the `choutstatus` register and the active signal on the `trigoutstatus` register. Finally clearing the channel removes this.

e.g.:

```
.../cti_sys0/channels# echo 2 1 > trigin_attach
.../cti_sys0/channels# echo 2 6 > trigout_attach
.../cti_sys0/channels# cat chan_free
0-1,3
.../cti_sys0/channels# cat chan_inuse
2
.../cti_sys0/channels# echo 2 > chan_xtrigs_sel
.../cti_sys0/channels# cat chan_xtrigs_trigin
1
.../cti_sys0/channels# cat chan_xtrigs_trigout
6
.../cti_sys0/# echo 1 > enable
.../cti_sys0/channels# echo 2 > chan_set
.../cti_sys0/channels# cat ../regs/choutstatus
0x4
.../cti_sys0/channels# cat ../regs/trigoutstatus
0x40
.../cti_sys0/channels# echo 2 > chan_clear
.../cti_sys0/channels# cat ../regs/trigoutstatus
0x0
.../cti_sys0/channels# cat ../regs/choutstatus
0x0
```

23.4 ETMv4 sysfs linux driver programming reference.

Author
Mike Leach <mike.leach@linaro.org>

Date
October 11th, 2019

Supplement to existing ETMv4 driver documentation.

23.4.1 Sysfs files and directories

Root: `/sys/bus/coresight/devices/etm<N>`

The following paragraphs explain the association between sysfs files and the ETMv4 registers that they effect. Note the register names are given without the 'TRC' prefix.

File
`mode` (rw)

Trace Registers
{CONFIGR + others}

Notes
Bit select trace features. See 'mode' section below. Bits in this will cause equivalent programming of trace config and other registers to enable the features requested.

Syntax & eg
`echo bitfield > mode`

bitfield up to 32 bits setting trace features.

Example
`$> echo 0x012 > mode`

File
`reset` (wo)

Trace Registers
All

Notes
Reset all programming to trace nothing / no logic programmed.

Syntax
`echo 1 > reset`

File
`enable_source` (wo)

Trace Registers
PRGCTLR, All hardware regs.

Notes

- > 0 : Programs up the hardware with the current values held in the driver and enables trace.
- = 0 : disable trace hardware.

Syntax
`echo 1 > enable_source`

File
`cpu` (ro)

Trace Registers
None.

Notes
CPU ID that this ETM is attached to.

Example
`$> cat cpu`

`$> 0`

File
`addr_idx` (rw)

Trace Registers
None.

Notes
Virtual register to index address comparator and range features. Set index for first of the pair in a range.

Syntax
`echo idx > addr_idx`

Where idx < nr_addr_cmp x 2

File
`addr_range` (rw)

Trace Registers
ACVR[idx, idx+1], VIIECTLR

Notes
Pair of addresses for a range selected by addr_idx. Include / exclude according to the optional parameter, or if omitted uses the current 'mode' setting. Select comparator range in control register. Error if index is odd value.

Depends
`mode, addr_idx`

Syntax
`echo addr1 addr2 [exclude] > addr_range`

Where addr1 and addr2 define the range and addr1 < addr2.

Optional exclude value:-

- 0 for include
- 1 for exclude.

Example
`$> echo 0x0000 0x2000 0 > addr_range`

File
`addr_single` (rw)

Trace Registers
ACVR[idx]

Notes
Set a single address comparator according to addr_idx. This is used if the address comparator is used as part of event generation logic etc.

Depends
`addr_idx`

Syntax
`echo addr1 > addr_single`

File
`addr_start` (rw)

Trace Registers
ACVR[idx], VISSCTLR

Notes
Set a trace start address comparator according to addr_idx. Select comparator in control register.

Depends
`addr_idx`

Syntax
`echo addr1 > addr_start`

File
`addr_stop` (rw)

Trace Registers
ACVR[idx], VISSCTLR

Notes
Set a trace stop address comparator according to addr_idx. Select comparator in control register.

Depends
`addr_idx`

Syntax
`echo addr1 > addr_stop`

File
`addr_context` (rw)

Trace Registers
ACATR[idx,{6:4}]

Notes
Link context ID comparator to address comparator addr_idx

Depends
`addr_idx`

Syntax
`echo ctxt_idx > addr_context`

Where ctxt_idx is the index of the linked context id / vmid comparator.

File
`addr_ctxtype` (rw)

Trace Registers
ACATR[idx,{3:2}]

Notes
Input value string. Set type for linked context ID comparator

Depends
`addr_idx`

Syntax
`echo type > addr_ctxtype`

Type one of {all, vmid, ctxid, none}

Example
`$> echo ctxid > addr_ctxtype`

File
`addr_exlevel_s_ns` (rw)

Trace Registers
ACATR[idx,{14:8}]

Notes
Set the ELx secure and non-secure matching bits for the selected address comparator

Depends
`addr_idx`

Syntax
`echo val > addr_exlevel_s_ns`

val is a 7 bit value for exception levels to exclude. Input value shifted to correct bits in register.

Example
`$> echo 0x4F > addr_exlevel_s_ns`

File
`addr_instdatatype` (rw)

Trace Registers
ACATR[idx,{1:0}]

Notes
Set the comparator address type for matching. Driver only supports setting instruction address type.

Depends
`addr_idx`

File
`addr_cmp_view` (ro)

Trace Registers
ACVR[idx, idx+1], ACATR[idx], VIIECTLR

Notes
Read the currently selected address comparator. If part of address range then display both addresses.

Depends
`addr_idx`

Syntax
`cat addr_cmp_view`

Example

```
$> cat addr_cmp_view
```

```
addr_cmp[0] range 0x0 0xffffffffffffffff include
ctrl(0x4b00)
```

File
`nr_addr_cmp` (ro)

Trace Registers
From IDR4

Notes
Number of address comparator pairs

File
`sshot_idx` (rw)

Trace Registers
None

Notes
Select single shot register set.

File
`sshot_ctrl` (rw)

Trace Registers
SSCCR[idx]

Notes
Access a single shot comparator control register.

Depends
`sshot_idx`

Syntax
`echo val > sshot_ctrl`

Writes val into the selected control register.

File
`sshot_status` (ro)

Trace Registers
SSCSR[idx]

Notes
Read a single shot comparator status register

Depends
`sshot_idx`

Syntax
`cat sshot_status`

Read status.

Example
`$> cat sshot_status`

`0x1`

File
`sshot_pe_ctrl` (rw)

Trace Registers
SSPCICR[idx]

Notes
Access a single shot PE comparator input control register.

Depends
`sshot_idx`

Syntax
`echo val > sshot_pe_ctrl`

Writes val into the selected control register.

File
`ns_exlevel_vinst` (rw)

Trace Registers
VICTLR{23:20}

Notes
Program non-secure exception level filters. Set / clear NS exception filter bits. Setting '1' excludes trace from the exception level.

Syntax
`echo bitfield > ns_exlevel_viinst`

Where bitfield contains bits to set clear for EL0 to EL2

Example
`%> echo 0x4 > ns_exlevel_viinst`

Excludes EL2 NS trace.

File
`vinst_pe_cmp_start_stop` (rw)

Trace Registers
VIPCSSCTLR

Notes
Access PE start stop comparator input control registers

File
`bb_ctrl` (rw)

Trace Registers
BBCTLR

Notes
Define ranges that Branch Broadcast will operate in. Default (0x0) is all addresses.

Depends
BB enabled.

File
`cyc_threshold` (rw)

Trace Registers
CCCTLR

Notes
Set the threshold for which cycle counts will be emitted. Error if attempt to set below minimum defined in IDR3, masked to width of valid bits.

Depends
CC enabled.

File
`syncfreq` (rw)

Trace Registers
SYNCPR

Notes
Set trace synchronisation period. Power of 2 value, 0 (off) or 8-20. Driver defaults to 12 (every 4096 bytes).

File
`cntr_idx` (rw)

Trace Registers
none

Notes
Select the counter to access

Syntax
`echo idx > cntr_idx`

Where idx < nr_cntr

File
`cntr_ctrl` (rw)

Trace Registers
CNTCTLR[idx]

Notes
Set counter control value.

Depends
`cntr_idx`

Syntax
`echo val > cntr_ctrl`

Where val is per ETMv4 spec.

File
`cntrldvr` (rw)

Trace Registers
CNTRLDVR[idx]

Notes
Set counter reload value.

Depends
`cntr_idx`

Syntax
`echo val > cntrldvr`

Where val is per ETMv4 spec.

File
`nr_cntr` (ro)

Trace Registers
From IDR5

Notes
Number of counters implemented.

File
`ctxid_idx` (rw)

Trace Registers
None

Notes
Select the context ID comparator to access

Syntax
`echo idx > ctxid_idx`

Where idx < numcidc

File
`ctxid_pid` (rw)

Trace Registers
CIDCVR[idx]

Notes
Set the context ID comparator value

Depends
`ctxid_idx`

File
`ctxid_masks` (rw)

Trace Registers
CIDCCTLR0, CIDCCTLR1, CIDCVR<0-7>

Notes
Pair of values to set the byte masks for 1-8 context ID comparators. Automatically clears masked bytes to 0 in CID value registers.

Syntax
`echo m3m2m1m0 [m7m6m5m4] > ctxid_masks`

32 bit values made up of mask bytes, where mN represents a byte mask value for Context ID comparator N.

Second value not required on systems that have fewer than 4 context ID comparators

---

File
`numcidc` (ro)

Trace Registers
From IDR4

Notes
Number of Context ID comparators

File
`vmid_idx` (rw)

Trace Registers
None

Notes
Select the VM ID comparator to access.

Syntax
`echo idx > vmid_idx`

Where idx < numvmidc

File
`vmid_val` (rw)

Trace Registers
VMIDCVR[idx]

Notes
Set the VM ID comparator value

Depends
`vmid_idx`

File
`vmid_masks` (rw)

Trace Registers
VMIDCCTLR0, VMIDCCTLR1, VMIDCVR<0-7>

Notes
Pair of values to set the byte masks for 1-8 VM ID comparators. Automatically clears masked bytes to 0 in VMID value registers.

Syntax
`echo m3m2m1m0 [m7m6m5m4] > vmid_masks`

Where mN represents a byte mask value for VMID comparator N. Second value not required on systems that have fewer than 4 VMID comparators.

File
`numvmidc` (ro)

Trace Registers
From IDR4

Notes
Number of VMID comparators

File
`res_idx` (rw)

Trace Registers
None.

Notes
Select the resource selector control to access. Must be 2 or higher as selectors 0 and 1 are hardwired.

Syntax
`echo idx > res_idx`

Where 2 <= idx < nr_resource x 2

File
`res_ctrl` (rw)

Trace Registers
RSCTLR[idx]

Notes
Set resource selector control value. Value per ETMv4 spec.

Depends
`res_idx`

Syntax
`echo val > res_cntr`

Where val is per ETMv4 spec.

File
`nr_resource` (ro)

Trace Registers
From IDR4

Notes
Number of resource selector pairs

File
`event` (rw)

Trace Registers
EVENTCTRL0R

Notes
Set up to 4 implemented event fields.

Syntax
`echo ev3ev2ev1ev0 > event`

Where evN is an 8 bit event field. Up to 4 event fields make up the 32-bit input value. Number of valid fields is implementation dependent, defined in IDR0.

File
`event_instren` (rw)

Trace Registers
EVENTCTRL1R

Notes
Choose events which insert event packets into trace stream.

Depends
EVENTCTRL0R

Syntax
`echo bitfield > event_instren`

Where bitfield is up to 4 bits according to number of event fields.

File
`event_ts` (rw)

Trace Registers
TSCTLR

Notes
Set the event that will generate timestamp requests.

Depends
`TS activated`

Syntax
`echo evfield > event_ts`

Where evfield is an 8 bit event selector.

File
`seq_idx` (rw)

Trace Registers
None

Notes
Sequencer event register select - 0 to 2

File
`seq_state` (rw)

Trace Registers
SEQSTR

Notes
Sequencer current state - 0 to 3.

File
`seq_event` (rw)

Trace Registers
SEQEVR[idx]

Notes
State transition event registers

Depends
`seq_idx`

Syntax
`echo evBevF > seq_event`

Where evBevF is a 16 bit value made up of two event selectors,

- evB : back
- evF : forwards.

File
`seq_reset_event` (rw)

Trace Registers
SEQRSTEVR

Notes
Sequencer reset event

Syntax
`echo evfield > seq_reset_event`

Where evfield is an 8 bit event selector.

File
`nrseqstate` (ro)

Trace Registers
From IDR5

Notes
Number of sequencer states (0 or 4)

File
nr_pe_cmp (ro)

Trace Registers
From IDR4

Notes
Number of PE comparator inputs

File
nr_ext_inp (ro)

Trace Registers
From IDR5

Notes
Number of external inputs

File
nr_ss_cmp (ro)

Trace Registers
From IDR4

Notes
Number of Single Shot control registers

Note: When programming any address comparator the driver will tag the comparator with a type used - i.e. RANGE, SINGLE, START, STOP. Once this tag is set, then only the values can be changed using the same sysfs file / type used to program it.

Thus:

```
% echo 0 > addr_idx              ; select address comparator 0
% echo 0x1000 0x5000 0 > addr_range ; set address range on
↪comparators 0, 1.
% echo 0x2000 > addr_start    ; error as comparator 0 is a range
↪comparator
% echo 2 > addr_idx              ; select address comparator 2
% echo 0x2000 > addr_start    ; this is OK as comparator 2 is
↪unused.
% echo 0x3000 > addr_stop     ; error as comparator 2 set as start
↪address.
% echo 2 > addr_idx              ; select address comparator 3
% echo 0x3000 > addr_stop     ; this is OK
```

To remove programming on all the comparators (and all the other hardware) use the reset parameter:

```
% echo 1 > reset
```

23.4.2 The 'mode' sysfs parameter.

This is a bitfield selection parameter that sets the overall trace mode for the ETM. The table below describes the bits, using the defines from the driver source file, along with a description of the feature these represent. Many features are optional and therefore dependent on implementation in the hardware.

Bit assignments shown below:-

---

bit (0):
ETM_MODE_EXCLUDE

description:
This is the default value for the include / exclude function when setting address ranges. Set 1 for exclude range. When the mode parameter is set this value is applied to the currently indexed address range.

bit (4):
ETM_MODE_BB

description:
Set to enable branch broadcast if supported in hardware [IDR0].

bit (5):
ETMv4_MODE_CYCACC

description:
Set to enable cycle accurate trace if supported [IDR0].

bit (6):
ETMv4_MODE_CTXID

description:
Set to enable context ID tracing if supported in hardware [IDR2].

bit (7):
ETM_MODE_VMID

description:
Set to enable virtual machine ID tracing if supported [IDR2].

bit (11):
ETMv4_MODE_TIMESTAMP

description:
Set to enable timestamp generation if supported [IDR0].

bit (12):
ETM_MODE_RETURNSTACK

description:
Set to enable trace return stack use if supported [IDR0].

bit (13-14):
ETM_MODE_QELEM(val)

description:
'val' determines level of Q element support enabled if implemented by the ETM [IDR0]

bit (19):
ETM_MODE_ATB_TRIGGER

description:
Set to enable the ATBTRIGGER bit in the event control register [EVENTCTLR1] if supported [IDR5].

bit (20):
ETM_MODE_LPOVERRIDE

description:
Set to enable the LPOVERRIDE bit in the event control register [EVENTCTLR1], if supported [IDR5].

bit (21):
ETM_MODE_ISTALL_EN

description:
Set to enable the ISTALL bit in the stall control register [STALLCTLR]

bit (23):
ETM_MODE_INSTPRIO

description:
Set to enable the INSTPRIORITY bit in the stall control register [STALLCTLR] , if supported [IDR0].

bit (24):
ETM_MODE_NOOVERFLOW

description:
Set to enable the NOOVERFLOW bit in the stall control register [STALLCTLR], if supported [IDR3].

bit (25):
ETM_MODE_TRACE_RESET

description:
Set to enable the TRCRESET bit in the viewinst control register [VICTLR] , if supported [IDR3].

bit (26):
ETM_MODE_TRACE_ERR

description:
Set to enable the TRCCTRL bit in the viewinst control register [VICTLR].

bit (27):
ETM_MODE_VIEWINST_STARTSTOP

description:
Set the initial state value of the ViewInst start / stop logic in the viewinst control register [VICTLR]

bit (30):
ETM_MODE_EXCL_KERN

description:
Set default trace setup to exclude kernel mode trace (see note a)

bit (31):
ETM_MODE_EXCL_USER

description:
Set default trace setup to exclude user space trace (see note a)

Note a) On startup the ETM is programmed to trace the complete address space using address range comparator 0. 'mode' bits 30 / 31 modify this setting to set EL exclude bits for NS state in either user space (EL0) or kernel space (EL1) in the address range comparator. (the default setting excludes all secure EL, and NS EL2)

Once the reset parameter has been used, and/or custom programming has been implemented - using these bits will result in the EL bits for address comparator 0 being set in the same way.

Note b) Bits 2-3, 8-10, 15-16, 18, 22, control features that only work with data trace. As A-profile data trace is architecturally prohibited in ETMv4, these have been omitted here. Possible uses could be where a kernel has support for control of R or M profile infrastructure as part of a heterogeneous system.

Bits 17, 28-29 are unused.