What is the kernel upto? Powerful tracing techniques

Joel Fernandes
<joel@linuxinternals.org>

Concepts: Tracing: Static vs Dynamic tracepoints

- Static: Tracepoint events
- Dynamic: kprobes

Static tracepoints: Methodology

- Define what your trace point looks like (what name, args etc)
- Define a trace function that accepts arguments from tracepoint users
- Find where to insert the tracepoinin your code
- Call the tracepoint function you just defined to broadcast the event

Static tracepoints

Other points:

- Code location is fixed at compiled time, can’t change it
- Better than printk:
  - Dynamically enabled
  - Lower overhead than printk, stored in fast ring-buffer
- No overhead when disabled (defaults to a NOP)

Static tracepoints: Real example - block layer

What sectors did I just write to? (block-sectors-demo)

Use the block:* static tracepoints

First record:

trace-cmd record -e “block:block_rq_insert” -F dd if=/dev/zero of=tmp

Demo.. trace-blk.sh demo script

Static tracepoints: Real example – block layer

What sectors did I just write to?

Use the block:* static tracepoints

Then report: “trace-cmd report”

```
dd-2791  [001]  1663.322775: block_rq_insert:     8,16 W 0 () 495704 + 24 [dd]
dd-2791  [001]  1663.322780: block_rq_issue:      8,16 W 0 () 495704 + 24 [dd]
dd-2791  [001]  1663.323027: block_rq_complete:   8,16 W () 495704 + 24 [0]
```

495704 is the sector number

24 is the number of sectors (we wrote 12k, that’s 24 sectors)

Static tracepoints: Real example - scheduler

- You can even write kernel modules to install your own probes on static tracepoints !!

Static tracepoints: Real example - scheduler

- Scheduler has a static TP called **sched_switch**, gives you information about what was switched in and was switched out.
- Imagine writing code that monitors a context-switch, the world is in your hands.
- Code and Demo (repo/cpuhists demo)

Static tracepoints: Real example - scheduler

- Very powerful feature
- Use it to write your own tools to understand kernel internals
- Very low-overhead technique (as its in-kernel)

Dynamic tracepoints: Methodology

- Find instruction address to instrument
- Save the instruction and install a jump in place
- Execute some code
- Then execute the instruction
- Then execute some more code and jump back

Dynamic tracepoints: Mechanism

- All Linux kernel based, even user-level probes
- Under the hood, uses kprobes (for kernel dyn. probes) and uprobes (for user)
- ‘perf’ tool provides easy access to create and record these probes instead of having to poke debugfs entries.

Dive into a example! - kprobes

Find out where you want to insert your probe (perf probe -L)

```
A.   ## perf probe  -k <path-to-vmlinux> -s <path-kernel-sources> -L tcp_sendmsg
B.   <tcp_sendmsg@.//net/ipv4/tcp.c:0>
C.         0  int tcp_sendmsg(struct sock *sk, struct msghdr *msg, size_t size)
D.         1  {
E.                   struct tcp_sock *tp = tcp_sk(sk);
F.                   struct sk_buff *skb;
G.                   int flags, err, copied = 0;
H.         5         int mss_now = 0, size_goal, copied_syn = 0;
I.                   bool sg;
J.                   long timeo;
K.
L.         9         lock_sock(sk);
M.            ….
```

Dive into a example! - kprobes

Find which variables are available in the function

```
A.   ## perf probe  -k <path-to-vmlinux> -s <path-kernel-sources> -V tcp_sendmsg

Available variables at tcp_sendmsg

     @<tcp_sendmsg+0>

               size_t  size

               struct msghdr*  msg

               struct sock*   sk
```

Dive into examples! - kprobes

Lets insert probe on Line 9 of tcp_sendmsg and have the probe fetch variable size.
This will help record sizes of all TCP messages being sent!

```
A.   ## perf probe -a ‘tcp_sendmsg:9 size`   -s <...> -k <...>

Added new event:  probe:tcp_sendmsg    (on tcp_sendmsg with size)

A.   ## perf record -e probe:tcp_sendmsg -a -- sleep 5
B.   root@ubuntu:/mnt/sdb/linux-4.4.2# perf script
C.      Socket Thread 127991 [002] 486500.563947: probe:tcp_sendmsg: (ffffffff81739ad5) size=0x50
D.      Socket Thread 127991 [002] 486502.535738: probe:tcp_sendmsg: (ffffffff81739ad5) size=0x205
E.      Socket Thread 127991 [002] 486502.538618: probe:tcp_sendmsg: (ffffffff81739ad5) size=0x205
F.      Socket Thread 127991 [002] 486502.540033: probe:tcp_sendmsg: (ffffffff81739ad5) size=0x205
G.      Socket Thread 127991 [002] 486502.540369: probe:tcp_sendmsg: (ffffffff81739ad5) size=0x205
H.      Socket Thread 127991 [002] 486502.543893: probe:tcp_sendmsg: (ffffffff81739ad5) size=0x205
I.      Socket Thread 127991 [002] 486502.545757: probe:tcp_sendmsg: (ffffffff81739ad5) size=0x205

PID 127991 is Firefox browser’s Socket Thread
```

Dive into examples! - uprobes

Find out how ‘malloc’ C library function is being called globally

```
## perf probe -x /lib/x86_64-linux-gnu/libc.so.6 -a 'malloc bytes'

Added new events:

  probe_libc:malloc     (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so with bytes)

  probe_libc:malloc_1   (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so with bytes)

  probe_libc:malloc_2   (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so with bytes)


Multiple probe points are because malloc gets inlined at a couple of places
```

Dive into examples! - uprobes

```
## perf record -e "probe_libc:malloc*" -a -- sleep 1

## perf script

...

       vmtoolsd  1977 [000] 487072.439953: probe_libc:malloc_1: (7f8a18a324a0) bytes=0x64

        vmtoolsd  1977 [000] 487072.439956: probe_libc:malloc_1: (7f8a18a324a0) bytes=0x27

        vmtoolsd  1977 [000] 487072.439960: probe_libc:malloc_1: (7f8a18a324a0) bytes=0x64

 gnome-terminal-  2253 [001] 487072.570703: probe_libc:malloc_1: (7f27c71504a0) bytes=0x22

 gnome-terminal-  2253 [001] 487072.570710: probe_libc:malloc_1: (7f27c71504a0) bytes=0x20
```

Pros/Cons

Advantages:

- Dynamic, no need of recompiling any code
- When probe point is hit, stack traces can also be collected
- Very quickly able to understand system/code behavior

Disadvantages:

- Requires symbol information present in executables (compiled with -g)
- Insert probes anywhere but the beginning of the function requires DWARF information present in executable (which comes with -g but for kprobes, vmlinux can be 100s of MB)

Timing functions

Methodology:

- Take a time stamp at start
- Take a time stamp at end
- Take the difference
- Report difference (if you want)

Timing functions: Using Ftrace

Use trace-cmd and function_graph plugin to get function times. Set max_depth to 1 or 2

trace-cmd record -p function_graph -g sys_write

echo 2 > /sys/kernel/debug/tracing/max_graph_depth

```
gdbus-2341  [001]  8364.697427: funcgraph_entry:                   |  sys_write() {
gdbus-2341  [001]  8364.697431: funcgraph_entry:         0.137 us   |     __fdget_pos();
gdbus-2341  [001]  8364.697431: funcgraph_entry:         0.371 us   |     vfs_write();
gdbus-2341  [001]  8364.697432: funcgraph_entry:         0.036 us   |     fput();
gdbus-2341  [001]  8364.697432: funcgraph_exit:          1.458 us   |   }
gdbus-2341  [001]  8364.697434: funcgraph_entry:                   |  sys_write() {
gdbus-2341  [001]  8364.697434: funcgraph_entry:         0.111 us   |     __fdget_pos();
gdbus-2341  [001]  8364.697435: funcgraph_entry:         0.243 us   |     vfs_write();
```

Timing functions: Using kretprobes

- Install a kretprobe for a function
- 2 handlers involved, one at beginning and one at end
- Take time stamps and find difference
- Much more powerful than function graph tracer
  - Dump function arguments (shown in thardirq example)
  - Execute kernel code in handlers, store and aggregate data
  - Use custom criteria to report timing (shown in tirqthread example)
- Much less clunkier than instrumenting code

Demo: thardirq (code + run)

Timing functions: Advanced usage

- Install a kprobe at function entry and kretprobe at end of function
- Determine time at function beginning and function end
- Take the difference
- More powerful, use criteria to determine if the time difference should be reported (such as context switching)

Example tsoftirq (just showing code..)

What kernel code is executed for graphics?

Demo:

ps ax|grep X

Find pid

trace-cmd record -p function -P <pid>

Found an interesting function what args are passed to: `vmw_validate_single_buffer`

perf+kprobe example: vmwgfx driver

(repo/perf-probe-vmwgfx demo)

Find out what the lines of code are in the vmw_validate_single_buffer function

```
# perf probe -L vmw_validate_single_buffer -k ./vmlinux -m vmwgfx

<vmw_validate_single_buffer@/mnt/sdb/linux-4.5.2/drivers/gpu/drm/vmwgfx/vmwgfx_execbuf.c:0>
      0  int vmw_validate_single_buffer(struct vmw_private *dev_priv,
                                        struct ttm_buffer_object *bo,
                                        bool interruptible,
                                        bool validate_as_mob)
      4  {
                struct vmw_dma_buffer *vbo = container_of(bo, struct vmw_dma_buffer,
                                                          base);
                int ret;

      9         if (vbo->pin_count > 0)
     10                 return 0;

     12         if (validate_as_mob)
     13                 return ttm_bo_validate(bo, &vmw_mob_placement, interruptible,
                                        false);
```

perf+kprobe example: vmwgfx driver

Create probe point:

```
# perf probe -k ./vmlinux -m vmwgfx --add 'vmw_validate_single_buffer:10 interruptible'

 probe:vmw_validate_single_buffer (on vmw_validate_single_buffer:10 in vmwgfx with
interruptible)

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

     perf record -e probe:vmw_validate_single_buffer -aR sleep 1
```

perf+kprobe example: vmwgfx driver

Start recording

```
# perf record -e probe:vmw_validate_single_buffer -aR glxgears

<vmw_validate_single_buffer@/mnt/sdb/linux-4.5.2/drivers/gpu/drm/vmwgfx/vmwgfx_execbuf.c:0>
      0  int vmw_validate_single_buffer(struct vmw_private *dev_priv,
                                       struct ttm_buffer_object *bo,
                                       bool interruptible,
                                       bool validate_as_mob)
      4  {
                struct vmw_dma_buffer *vbo = container_of(bo, struct vmw_dma_buffer,
                                                          base);
                int ret;

      9         if (vbo->pin_count > 0)
     10                 return 0;

     12         if (validate_as_mob)
     13                 return ttm_bo_validate(bo, &vmw_mob_placement, interruptible,
                                              false);
```

perf+kprobe example: vmwgfx driver

Output of finding function arguments of vmw_validate_single_buffer:

```
   glxgears 12104 [003]  3161.546611: probe:vmw_validate_single_buffer: (ffffffffc01c60bb) interruptible=0x1
         glxgears 12104 [003]  3161.546612: probe:vmw_validate_single_buffer: (ffffffffc01c60bb)
interruptible=0x1
         glxgears 12104 [003]  3161.546616: probe:vmw_validate_single_buffer: (ffffffffc01c60bb)
interruptible=0x1
         glxgears 12104 [003]  3161.546688: probe:vmw_validate_single_buffer: (ffffffffc01c60bb)
interruptible=0x1
         glxgears 12104 [003]  3161.546759: probe:vmw_validate_single_buffer: (ffffffffc01c60bb)
interruptible=0x1
         glxgears 12104 [003]  3161.546764: probe:vmw_validate_single_buffer: (ffffffffc01c60bb)
interruptible=0x1
         glxgears 12104 [003]  3161.546766: probe:vmw_validate_single_buffer: (ffffffffc01c60bb)
interruptible=0x1
         glxgears 12104 [003]  3161.546768: probe:vmw_validate_single_buffer: (ffffffffc01c60bb)
interruptible=0x1
         glxgears 12104 [003]  3161.546770: probe:vmw_validate_single_buffer: (ffffffffc01c60bb)
interruptible=0x1
```

Careful with overhead

In the worst case, assuming you’re hosing the tracing system as fast as you can, the over head can be as much as 50% or more..

Example:

```
# dd if=/dev/zero of=/dev/null bs=4k count=10000000 conv=fdatasync 2>&1

40960000000 bytes (41 GB) copied, 3.8354 s, 10.7 GB/s

# trace-cmd record -e syscalls:sys_enter_write

40960000000 bytes (41 GB) copied, 7.97696 s, 5.1 GB/s
```

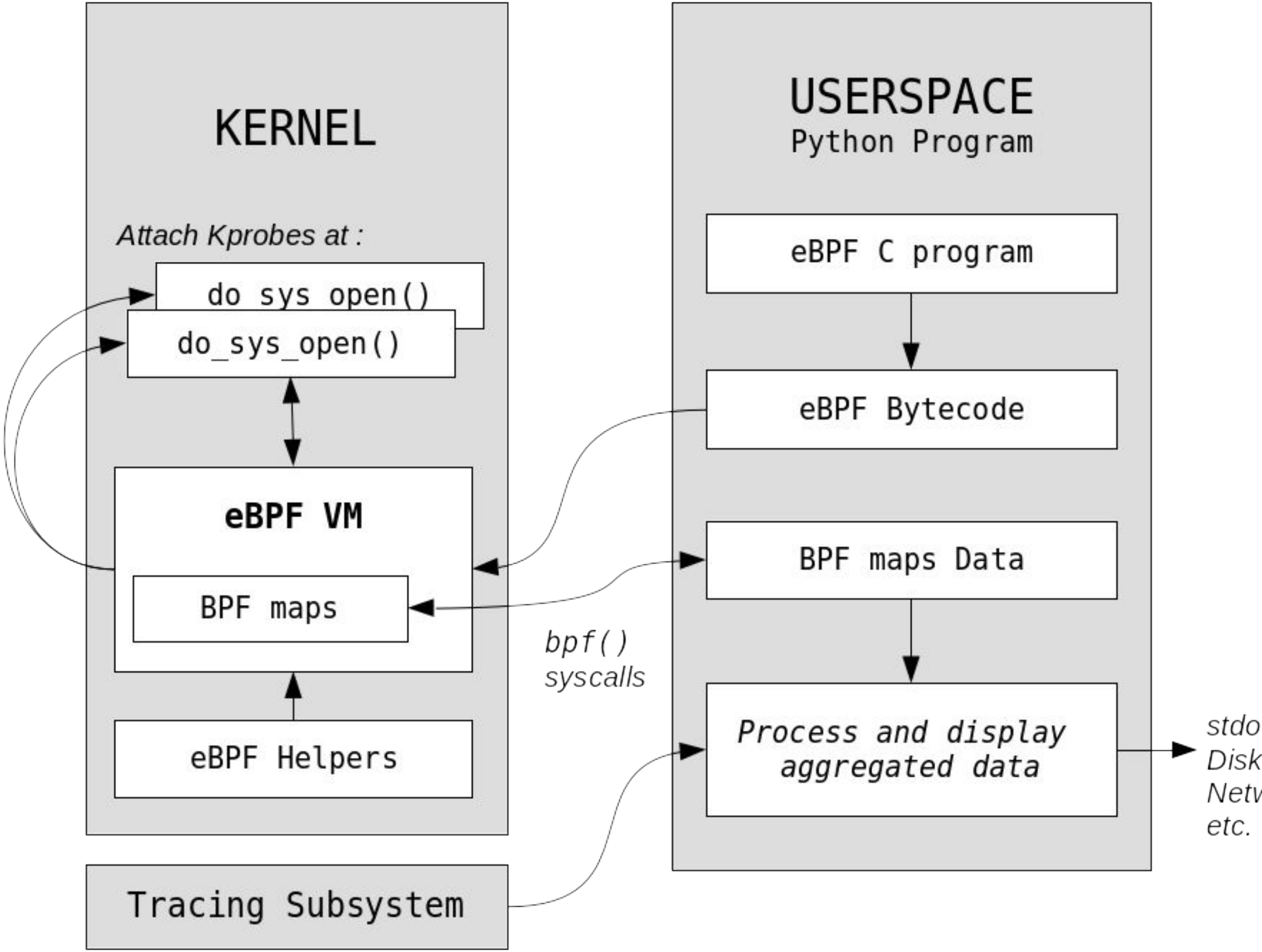
A 50% slow down!

Careful with overhead

Solutions:

- Only enable events of interest
- Use event filters, both ftrace and perf have these. Perf is lesser in overhead
- Do in-kernel tracing using eBPF, systemtap or your own kernel modules

BPF: Scripting to safely instrument kernel code



BPF: Scripting to safely instrument kernel code

Architecture

- Byte code generated by CLANG that can be interpretted by kernel

SAFE:

- Write code that exposes all the things we just saw writing kernel modules
- Safe to run in kernel (memory accesses are checked, no faults)
- Cannot crash the kernel

More pros:

- Powerful datastructures that reduce boiler plate (such as hashtables)

BPF: Scripting to safely instrument kernel code

- Demo - tracex5: Trace system calls in a process

How it works:

- bpf program is loaded from userspace
- bpf byte code interpretted by kernel
- Kernel inserts kprobes for different system calls
- Kprobe handlers are hit and print stuff to output

Profiling to explore and understand code: how it works

Methodology:

- Interrupt the CPU periodically (100 or 200 times a second)
- See what's running on CPU
- Low overhead, unlike tracing
- Doesn't catch everything, some of it bound to be missed

Profiling to explore and understand code: how it works

Mechanism:

- Performance Monitoring Unit (PMU) in CPUs generate an interrupt
- NMI interrupts CPU and a stack trace is taken
- NMI has highest priority and is not affected by masking
- In Linux, perf_events framework is used