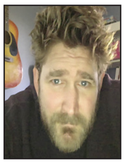
iVoyeur
BPF—Part Three

DAVE JOSEPHSEN


Dave Josephsen is a book author, code developer, and monitoring expert who works for Fastly. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

It is a little known fact [1] that as pre-teens Romeo and Juliet, both by nature predisposed to notions of impossible love and emo anti-parental overreaction, independently happened upon and fell in love with Ovid's *Metamorphoses*, wherein is related the tale of the OG suicidal power couple, Pyramus and Thisbe.

Neighbors, whose dwellings were built upon a common center wall in the lovely city of Babylon, Pyramus and Thisbe were cruelly forced apart by their respective families, who shared not only a foundation wall, but also a bitter long-running feud. So close, and yet so far; the phrase itself might have been invented to literally describe their specific predicament, for although their love burned so bright the gods took notice, they might as well have been separated by an ocean.

Until one day, a crack formed low in the basement wall that separated their dwellings. Each noticing separately, and then by degrees stealing into the basement in the night to hear each other whisper their love through the crack in the wall, and to sometimes pass messages and precious tokens of love as opportunity allowed. Eventually they both stabbed themselves. A lion was somehow involved—the precise details escape me, but in probably humanity's earliest example of negative media influence on youth [2], Romeo and Juliet followed in kind some 1500 years later.

Anyway, I know exactly what you're thinking. The basement wall is a textbook perfect metaphor for the memory-enforced separation of kernel space and userspace in monolithic kernel architecture! I know, right? Each side yearning for and depending upon the other?! Each sharing a common heartbeat but never an embrace! Doomed forever to content themselves with whispered secrets and messages passed through cracks in the wall forever holding them apart. Sigh.

Passing Messages

In my last article [3], we took a first look at the `biolatency.py` source code and dove into the kernel source to get a basic understanding of the block I/O layer and what requests at that layer of Linux look like. In this article, as promised, we're going to talk about message passing and the three mechanisms BPF gives us to whisper precious data through the wall between kernel space and our userspace Python runtime. I'll briefly cover all three, though the third and final method is the one we really care about, as it's the one used by biolatency itself.

The first method we have to send ourselves a message-in-a-bottle from kernel space is the `bpf_trace_printk()` function. For an example of its use, consider the BCC tools' one-liner "Hello, World!" program [4]:

```
from bcc import BPF
BPF(text='int kprobe__sys_clone(void *ctx) { bpf_trace_printk("Hello, World!\\n");
return 0; }').trace_print()
```

The C portion of this program attaches to the `sys_clone()` system call and uses `bpf_trace_printk()` to print the string "Hello, World!" to the system "common trace pipe" (`/sys/kernel/debug/tracing/trace_pipe`) whenever a new process is created. On the Python side, we slurp it from the pipe with the `trace_print()` method, which opens the file and prints whatever it finds within [5].

This approach is straightforward and makes for easy one-liner style tool development, but it has a few problems that make it unusable for anything but light testing and one-off tools. Primarily, it's called the "common" trace pipe because it's shared by every BPF filter that uses `bpf_trace_printk()`.

Ignoring the other technical limitations for a moment, this mechanism doesn't even work well with my diligently constructed *metaphor*—more akin to shouting our messages out the window than surreptitiously passing notes in the classroom; making `bpf_trace_printk()` not just a technical but, more importantly, a *literary* non-starter. I think you'll agree, if Romeo and Juliet had to depend on world-readable sockets for message passing, their love would never have survived long enough to result in tragic mutual suicide.

Obviously, to write tasteful trace programs, we'll need a way to get data from our probe without the pollution of a system-common datapath.

Let's therefore abandon `printk` and move on to the second means of data-transfer from a kernel-side probe: `BPF_PERF_OUTPUT()`. This is a ring-buffer of shared memory that contains a pointer to some data that you want to pass from kernel space into your Python program. A proper piece of shared memory, safe from prying eyes. Let's take a look at how the C-side (kernel-side) code uses `BPF_PERF_OUTPUT()`; this snippet is from the `hello_perf_output.py` example in the BCC tools repo [6]:

```
// define output data structure in C
struct data_t {
    u32 pid;
    u64 ts;
    char comm[TASK_COMM_LEN];
};
BPF_PERF_OUTPUT(events);

int hello(struct pt_regs *ctx) {
    struct data_t data = {};

    data.pid = bpf_get_current_pid_tgid();
    data.ts = bpf_ktime_get_ns();
    bpf_get_current_comm(&data.comm, sizeof(data.comm));

    events.perf_submit(ctx, &data, sizeof(data));

    return 0;
}
```

Now this is more like it. At the top of this probe, we define `data_t`, an arbitrary data structure whose contents are controlled by us. This is the envelope we will press through the crack in the wall between kernel and userspace. Its secret contents, completely our discretion. In this example, we have three bits of info: the PID of the process that triggered the probe (`pid`), the current system time in nanoseconds (`ts`), and the name of the current process (`comm`).

Each of these three tantalizing intimacies is retrieved by a `bpf_get` function and packed into an instance of `data_t` called, unimaginatively, `data`. There is a small number of these helper-functions [7] available in BPF to retrieve various pieces of context from the kernel at the time the probe was fired. `bpf_ktime_get_ns()` is an extremely common bit of passed data, given that we are almost always timing system calls, or system-call frequency, with BPF. Once packed into our data envelope, we deliver our message with a method call on the `BPF_PERF_OUTPUT` ring buffer, which we've named *events*:

```
events.perf_submit(ctx, &data, sizeof(data));
```

I need to call a quick time out here, before we head back to the Python side, to more closely examine the call to `BPF_PERF_OUTPUT(events);` and talk about variable scope in your C-side probe code. `BPF_PERF_OUTPUT(events);` is the call that creates the ring-buffer we need to pass our `data` struct into userspace (and gives it the name `events`), and I want to explicitly point out *where* in the code it's being called, namely, above our `hello()` function, making it a *globally scoped* variable within the context of our probe. That is, `events` persists between invocations of our `hello()` function, so every time the kernel calls `sys_clone()` and wakes up our probe, the new invocation of `hello()` will reuse the same `BPF_PERF_OUTPUT` instance.

Stated more explicitly, our `hello()` function will only be in scope for a single triggering of a `sys_clone()` system call. It fires and exits with each new process created by the kernel, and then it returns, its context sacrificed to the reallocation gods. This is fine if we just want to blurt a "hello" into the world per invocation, but what if we want to do something more stateful? Like, for example, counting the total number of `sys_clone()` calls throughout the lifetime of our probe's invocation?

The globally scoped `events` ring buffer implies the answer. Because it's scoped outside our `hello()` function, it remains in memory as long as our Python script is running. Hence, if BPF provided something more like a map than a ring buffer (spoiler alert; it does), we could use that map to store data between probe invocations and slurp it up on a timer, or when we catch a keyboard-interrupt on the Python side.

Speaking of the Python side, let's return there now, where we use a blocking call to `perf_buffer_poll()` inside an unbounded loop to check for new data from our `events` ring buffer, like Pyramus constantly slipping downstairs to check for a message from his cherished neighbor. This polling method is called on the top-level BPF object, once we've explicitly opened the ring buffer with `open_perf_buffer()`, the first line of the blurb below:

```
b["events"].open_perf_buffer(print_event)
while 1:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        exit()
```

There are two important things to note about this open_perf_buffer() call. The first is its argument, in this case print_event; this is a function pointer or "callback." It tells perf_buffer_poll() where to send the love letters gleaned from the far side of the wall. The second and more important is how we're dereferencing the events ring buffer itself, as a dictionary entry from the top-level BPF object b["events"].

This brings us to the third means we have of smuggling sweet nothings through the wall between our kernel space probe and our userspace Python script: *Maps*. As I implied above, BPF provides myriad Map-like data-structures [8] that you can use to capture stateful information like invocation counts and timings between the system calls captured by your probe. These data structures can all be accessed on the Python side as dictionary values attached to the top-level BPF object, in the same way we're accessing the events ring buffer in the code blurb above: b["events"].

Let's take a moment to think about biolatency.py's requirements. From my last article, you'll remember that we're inserting not just one but *two* block I/O layer probes. The first (depending on whether we care about queue-time or not) fires on the blk_start_request() system call and invokes our probe's trace_req_start() function. The second fires on the kernel's blk_account_io_done() and invokes our trace_req_done() probe function. In other words, one probe fires when the block I/O event starts, and the other fires when it ends.

Here's the code [9]:

```
if args.queued:
    b.attach_kprobe(event="blk_account_io_start", \
       fn_name="trace_req_start")
else:
    if BPF.get_kprobe_functions(b'blk_start_request'):
        b.attach_kprobe(event="blk_start_request", \
           fn_name="trace_req_start")
    b.attach_kprobe(event="blk_mq_start_request", \
           fn_name="trace_req_start")
b.attach_kprobe(event="blk_account_io_done",
    fn_name="trace_req_done")
```

If you've inferred, without needing to look at the C-side trace-req functions, that we're going to be using bpf_ktime_get_ns() to capture the "start" system time, and again to capture the "end" system time, and then subtract them to derive an elapsed time from trace_req_start to trace_req_done, you are absolutely correct. We'll use a globally scoped BPF_HASH data structure to store the start times until they can be matched up to their respective "done" events. The invocation to create the hash in the biolatency code looks like this:

```
BPF_HASH(start, struct request *);
```

The map structures provided by BPF are sort of reminiscent of Java generics in that you specify their type as arguments. The first argument in the call above is its name: start. The second argument specifies the type of the key value in the hash. Here, we're specifying that the hash will be keyed by a struct pointer, literally a number that represents the memory address where a block I/O request struct is stored. This is a pretty clever value for a hash key because it's terse and will always uniquely identify a given I/O request between the start syscall and done syscall. The third argument, which would define the value-type of the hash, is omitted here, so it defaults to a u64, which happens to be exactly the return type of bpf_ktime_get_ns().

This BPF_HASH structure is only used to hold the timestamps of each start probe firing. It doesn't communicate anything to userspace since its values are set by the start probe and dereferenced by the done probe to compute an elapsed time for the I/O request. This means we need another structure to store the elapsed times and communicate these through the wall to the Python side.

You might remember from my first article on eBPF tools [10] that biolatency.py presents these values in the form of a histogram, keyed in various ways based on user-provided options (overall summary, per-disk, per I/O-type (read/write etc.)). The use of a histogram here makes a lot of sense because, as you can probably imagine, a busy box may produce a high cardinality of I/O request syscalls. If we tried to shove a note through the wall for every I/O request as we did in the previous examples, we might undermine the wall and send the house collapsing down on top of our heads.

Instead, biolatency keeps the data kernel-side, using globally scoped HISTOGRAM data-structure to collect the timings computed by our probe's done function, as a series of counters within a distributed series of "buckets" representing the range of their values. This is easy on kernel memory (since we're merely storing 64 counters) as well as on the userspace boundary (since we only need to transfer these values once, when we tear down the probe).

Unfortunately, things get a little muddled here since biolatency.py needs to use a few different storage back ends and techniques depending on end-user options. Rather than glossing over the interesting details in the space I have left, I will see you in the next issue, where we will take a brief tour of histogram theory, base-two logarithms and the "powers of two rule," and decode biolatency.py's series of substitution choices for the different kinds of block I/O histograms it can depict.

Take it easy.

References

[1] Not a fact. Completely made up.

[2] Romeo and Juliet were imaginary characters who never read Ovid, and that's not how media influence works.

[3] D. Josephsen, "iVoyeur: eBPF Tools," *;login:*, vol. 45, no. 2 (Summer 2020): https://www.usenix.org/publications/login/summer2020/josephsen.

[4] https://github.com/iovisor/bcc/blob/master/examples/hello_world.py.

[5] https://github.com/iovisor/bcc/blob/10603c7123c4b2157190151b63ea846c04c76037/src/python/bcc/__init__.py#L1214.

[6] https://github.com/iovisor/bcc/blob/master/examples/tracing/hello_perf_output.py.

[7] https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#data.

[8] https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#maps.

[9] https://github.com/iovisor/bcc/blob/master/tools/biolatency.py#L135-L142.

[10] D. Josephsen, "iVoyeur—eBPF Tools: What's in a Name?" *;login:*, vol. 45, no. 1 (Spring 2020): https://www.usenix.org/publications/login/mar20/josephsen.