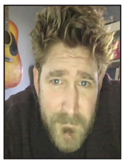
iVoyeur
eBPF Tools

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

I spent my high-school years in a tightly entangled group of four friends. We were basically inseparable, formed a horrible rock band, and I think did a lot of typical '90s Los Angeles kid things like throwing powdered doughnuts into oncoming traffic and making a nuisance of ourselves at 7-Eleven and Guitar Center. We smashed against the breakwater of graduation and went different places, but of the four of us, I was the only one who didn't go off to college to study music theory. Opting instead to eject into the Marine Corps, which is a longer story, and irrelevant to the current metaphor.

Anyway, we kept in touch, and in their letters all three of my friends described the process of learning music theory in a very similar way. As a neophyte musician, you typically have some aptitude with one or two instruments, but very little knowledge about how music itself works. Evidently in the first year of music theory, you are presented with myriad complicated rules. From what I understand, in fact, you do little else the first year but learn the rules and some important exceptions to the rules.

Then bit by bit, as the years progress, the rules are stripped away, until you reach some sort of musical enlightenment, where there are no rules and you work in a kind of effortless innovatory fugue where everything you create just clicks.

I vaguely remember feeling this way about computer science. Having written my first Perl script, flush with optimism and newfound aptitude. "So this is what it feels like to have mastered computering at last," I thought to myself, setting aside my Camel book to cross my arms in a self-satisfied way, and cursing whatever company I was working for at the time with whatever abomination I'd just created.

Many—er, well, several years later, I feel strongly that computer science is something like the exact opposite of how my friends described music theory in those hastily scribbled letters all that time ago. The rules do not so much disappear but rather change and reassemble anew every so often, and instead of effortless enlightenment, I find myself splitting my days between confounded frustration and shocked dismay, each of those punctuated by short bouts of relief and semi-comprehension. In our world I sometimes feel like it's a miracle anything works at all, and the more I learn, the less I seem to know.

In my last article I introduced eBPF, the extended Berkeley Packet Filter, along with a shell tool called `biolatency`, which uses eBPF-based kernel probes to instrument the block I/O (or bio) layer of the kernel and return per-device latency data in the form of a histogram. There is a deeply refreshing crispness about delving into the solar system of eBPF, a brisk undercurrent that pulls one down through abstraction layers and toward the metal. There are over 150 tools in the BCC (https://github.com/iovisor/bcc) tools suite, and you can use them all without knowing how they work, of course. I think you'll find, however, that your effectiveness with BCC tools like `biolatency` scales linearly with your knowledge of kernel internals, and the slightest exploration into their inner workings leads one directly into the kernel source.

Let's begin this second article on eBPF, therefore, with a short discussion of the Linux Kernel's "bio" layer [1]. This is the kernel software layer loosely defined as the contents of the `block` subdirectory of the Linux kernel source. The code here resides between file systems like ext3 and device drivers, which do the work of interacting directly with storage hardware.

At this layer, we are below abstraction notions like files and directories. Disks are represented by a small `struct` inside the kernel called `struct_gendisk` [2], for "generic disk," and reads and writes no longer exist as separate entities. Instead, all types of block I/O operations are wrapped inside a generic request wrapper called `struct_bio` [2], the `struct` for which the "bio" layer is named.

Without delving any further into the bio layer, we can already see how ideally situated the bio layer is for trace-style instrumentation. Above us, in the file systems, we would need to probe every kind of disk operation: a different probe for reads, writes, opens, etc. Below bio we will find vendor-specific code and a mountain of historical, related exceptions and complications. But right here inside bio, we have a single, well-defined data-structure that represents every type of disk I/O possible operation. No writes, no reads, just requests, and one probe can summarize them all.

We can also assume that tracing these requests will give us read access to the `struct_bio` data structure, because we'll need it to see what kind of request we're dealing with (e.g., read/write), what block device each request is destined for, and so on.

We now have the necessary information to take our first cursory glance inside `biolatency.py` [3] to intuit what's going on. The first 53 lines are pretty typical preamble for a Python script: documentation, imports, and argument parsing. The arguments are interesting, but we'll set them aside for now to take a look at the large string that begins on line 55:

```
# define BPF program
bpf_text = """
#include <uapi/linux/ptrace.h>
#include <linux/blkdev.h>
typedef struct disk_key {
```

From our last article you'll remember that eBPF is a virtual machine that resides in the kernel. This string (named `bpf_text`) is the payload intended for that in-kernel VM; it takes up about a quarter of the overall code in the Python script and is written in C. It is a program, embedded within our program, that will be compiled to bytecode and loaded into the kernel's eBPF VM. If you look closely, you'll notice that this C code won't compile as is, because of expressions like this one on line 70:

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

These are string-replacement match targets. These will be replaced in this string with valid code, depending on options passed in by the user. These substitutions begin on line 103 and all take the same general form:

```
if args.milliseconds:
    bpf_text = bpf_text.replace('FACTOR', 'delta /= 1000000;')
    label = "msecs"
else:
    bpf_text = bpf_text.replace('FACTOR', 'delta /= 1000;')
    label = "usecs"
```

All of these substring substitutions follow the same basic pattern: if option X was set by the user, then replace MACRO in the payload program with value Y; otherwise, replace MACRO with value Z. In the example above, we're choosing between micro and milliseconds in the payload string. We're also setting a "label" variable to give hints for properly printing the output later on. This process of rewriting sections of the payload string goes on for most of the options the user passes in. The exception is `-Q`, which selects whether we will include the time an I/O request spends queued in the kernel as part of the latency calculation.

This switch affects our choice of which particular kernel functions we ultimately choose to trace. If we don't care about queue-time, we will want to measure latency starting from the moment the I/O request is *issued*. However if `-Q` is set, we will also want to include the time each request spent waiting on the kernel. We can see how this is implemented starting on line 134:

```
b = BPF(text=bpf_text)
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")
```

First, we instantiate a new BPF Python object, passing in our newly rewritten payload in the process. What happens next depends on the `-Q` option. If we care about the latency induced by in-kernel queue time, then we'll insert our kernel probe at the `blk_account_io_start()` kernel function, which is called when an I/O request is first queued in the kernel. However, if we want to measure "pure" block I/O latency—that is, the amount of time a given generic I/O request took to return—we'll instrument `blk_mq_start_request()` and possibly `blk_start_request()` if the latter function exists in the current kernel. No matter what paths we choose, we'll close each trace at `blk_account_io_done()`.

At this point, our payload is inserted into the running kernel, and we are collecting data. Now we are confronted with some bitwise arithmetic beginning with a collection of constants on line 147 and continuing with some bitmask construction, and constants definition on line 157:

```
REQ_OP_BITS = 8
REQ_OP_MASK = ((1 << REQ_OP_BITS) - 1)
REQ_SYNC = 1 << (REQ_OP_BITS + 3)
REQ_META = 1 << (REQ_OP_BITS + 4)
REQ_PRIO = 1 << (REQ_OP_BITS + 5)
```

This is necessary to understand the data we're collecting. The bit-specifics correspond to the `bi_opf` [4] attribute (bio operational flags) inside `struct_bio`, the central block I/O request struct I mentioned above in the bio layer. The attribute is an unsigned `int` that's used to track metadata about a given block I/O request. You can see the constant defs for this bitmask a few lines down [5] from the `struct_bio` definition in the kernel source. In short, these flags tell us whether a given request was a read, write, cache-flush, etc. and provide some additional metadata about the operation, whether it was priority, backgrounded, read-ahead, etc.

If you continue down to line 171 in biolatency, you'll see that we `AND` the flags value, given to us from the probe, against a bitmask with bit 7 set to determine an integer value that corresponds to the top-level operation type (read, write, flush, discard, etc.). We then proceed to individually check for flags in the bitmask which correspond to subcategories. Prepending these to the top-level operation type:

```
if flags & REQ_SYNC:
    desc = "Sync-" + desc
if flags & REQ_META:
    desc = "Metadata-" + desc
if flags & REQ_FUA:
    desc = "ForcedUnitAccess-" + desc
```

So if the flags mask `AND`'d to a value of 0, which equates to "Read," and then we subsequently discovered that bit 11 was set in the flags mask corresponding to "Sync," we'd wind up filing this bio-request under "Sync-Read." Biolatency can use this data to plot histograms of I/O latency *per category of I/O operation* with the `-F` flag.

The last section in the script deals with printing our output. The script stays in the foreground until it encounters a keyboard interrupt from the user, and then dumps its output depending on how the user specified they wanted to see it in the argument flags. Unfortunately, these all use functions defined deeper inside the BCC Python library code, and scratching at them requires us to understand the eBPF data model, and a little bit more about the line between kernel and userspace, all of which we will get into in our next article.

If you're feeling like you know less than you did when you came in, then you are in a pretty good place. As I said in the intro, studying eBPF internals brings you close to the kernel in short order, which is a refreshing place to be. If you'd like to read a little more about the kernel's bio layer, there is an excellent set of introductory articles at LWN [1], and Brendan Gregg's BCC Python Development Tutorials [6] are another great resource for those wanting to read ahead.

Take it easy.

References

[1] "A block layer introduction part 1: The bio layer": https://lwn.net/Articles/736534/.

[2] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/genhd.h?h=v4.13#n171.

[3] https://github.com/iovisor/bcc/blob/master/tools/biolatency.py.

[4] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/blk_types.h?h=v4.14-rc1#n54.

[5] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/blk_types.h?h=v4.14-rc1#n182.

[6] https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md.