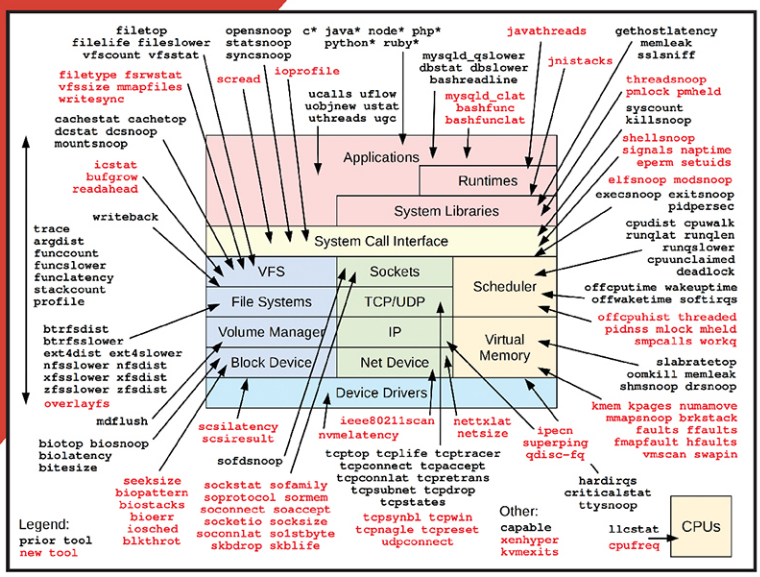
BPF Performance Tools

Linux System and Application Observability

Brendan Gregg



Foreword by Alexei Starovoitov,
creator of the new BPF

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

BPF Performance Tools

```
         2 -> 3          : 243      |************************               |
         4 -> 7          : 1        |                                       |
         8 -> 15         : 0        |                                       |
        16 -> 31         : 384      |****************************************|
        32 -> 63         : 0        |                                       |
        64 -> 127        : 0        |                                       |
       128 -> 255        : 0        |                                       |
       256 -> 511        : 0        |                                       |
       512 -> 1023       : 0        |                                       |
      1024 -> 2047       : 267      |***************************            |
      2048 -> 4095       : 2        |                                       |
      4096 -> 8191       : 23       |**                                     |

[...]
```

This output shows that there were many reads in the 16- to 31-byte range, as well as the 1024- to 2047-byte range. The `-C` option to argdist(8) can be used instead of `-H` to summarize as a frequency count of sizes rather than a histogram.

This is showing the read requested size since the entry to the syscall was instrumented. Compare it with the return value from the syscall exit, which is the number of bytes actually read:

```
# argdist -H 't:syscalls:sys_exit_read():int:args->ret'
[09:12:58]
     args->ret          : count     distribution
         0 -> 1         : 481      |****************************************|
         2 -> 3         : 116      |*********                               |
         4 -> 7         : 1        |                                        |
         8 -> 15        : 29       |**                                      |
        16 -> 31        : 6        |                                        |
        32 -> 63        : 31       |**                                      |
        64 -> 127       : 8        |                                        |
       128 -> 255       : 2        |                                        |
       256 -> 511       : 1        |                                        |
       512 -> 1023      : 2        |                                        |
      1024 -> 2047      : 13       |*                                       |
      2048 -> 4095      : 2        |                                        |

[...]
```

These are mostly zero- or one-byte reads.

Thanks to its in-kernel summary, argdist(8) is useful for examining syscalls that were called frequently. trace(8) prints per-event output and is suited for examining less-frequent syscalls, showing per-event timestamps and other details.

bpftrace

This level of syscall analysis is possible using bpftrace one-liners. For example, examining the requested read size as a histogram:

```
# bpftrace -e 't:syscalls:sys_enter_read { @ = hist(args->count); }'
Attaching 1 probe...
^C

@:
[1]                 1102 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[2, 4)               902 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@          |
[4, 8)                20 |                                                    |
[8, 16)               17 |                                                    |
[16, 32)             538 |@@@@@@@@@@@@@@@@@@@@@@@@@                           |
[32, 64)              56 |@@                                                  |
[64, 128)              0 |                                                    |
[128, 256)             0 |                                                    |
[256, 512)             0 |                                                    |
[512, 1K)              0 |                                                    |
[1K, 2K)             119 |@@@@@                                               |
[2K, 4K)              26 |@                                                   |
[4K, 8K)             334 |@@@@@@@@@@@@@@@                                     |
```

And the return value:

```
# bpftrace -e 't:syscalls:sys_exit_read { @ = hist(args->ret); }'
Attaching 1 probe...
^C

@:
(..., 0)             105 |@@@@                                                |
[0]                   18 |                                                    |
[1]                 1161 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[2, 4)               196 |@@@@@@@@                                            |
[4, 8)                 8 |                                                    |
[8, 16)              384 |@@@@@@@@@@@@@@@@@                                   |
[16, 32)              87 |@@@                                                 |
[32, 64)             118 |@@@@@                                               |
[64, 128)             37 |@                                                   |
[128, 256)             6 |                                                    |
[256, 512)            13 |                                                    |
[512, 1K)              3 |                                                    |
[1K, 2K)               3 |                                                    |
[2K, 4K)              15 |                                                    |
```

bpftrace has a separate bucket for negative values ("(..., 0)"), which are error codes returned by read(2) to indicate an error. You can craft a bpftrace one-liner to print these as a frequency count (as shown in Chapter 5) or a linear histogram so that the individual numbers can be seen:

```
# bpftrace -e 't:syscalls:sys_exit_read /args->ret < 0/ {
    @ = lhist(- args->ret, 0, 100, 1); }'
Attaching 1 probe...
^C

@:
[11, 12)             123 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
```

This output shows that error code 11 was always returned. Checking the Linux headers (asm-generic/errno-base.h):

```
#define EAGAIN          11      /* Try again */
```

Error code 11 is for "try again," an error state that can occur in normal operation.

6.3.12 funccount

funccount(8), introduced in Chapter 4, is a BCC tool that can frequency-count functions and other events. It can be used to provide more context for software CPU usage, showing which functions are called and how frequently. profile(8) may be able to show that a function is hot on CPU, but it can't explain why[20]: whether the function is slow, or whether it was simply called millions of times per second.

As an example, this frequency-counts kernel TCP functions on a busy production instance by matching those that begin with "tcp_":

```
# funccount 'tcp_*'
Tracing 316 functions for "tcp_*"... Hit Ctrl-C to end.
^C
FUNC                                    COUNT
[...]
tcp_stream_memory_free                 368048
tcp_established_options                381234
tcp_v4_md5_lookup                      402945
tcp_gro_receive                        484571
tcp_md5_do_lookup                      510322
Detaching...
```

20 profile(8) can't explain this easily. Profilers including profile(8) sample the CPU instruction pointer, and so a comparison with the function's disassembly may show whether it was stuck in a loop or called many times. In practice, it can be harder than it sounds: see Section 2.12.2 in Chapter 2.

This output shows that tcp_md5_do_lookup() was most frequent, with 510,000 calls while tracing.

Per-interval output can be generated using `-i`. For example, the earlier profile(8) output shows that the function get_page_from_freelist() was hot on CPU. Was that because it was called often or because it was slow? Measuring its per-second rate:

```
# funccount -i 1 get_page_from_freelist
Tracing 1 functions for "get_page_from_freelist"... Hit Ctrl-C to end.

FUNC                                    COUNT
get_page_from_freelist                 586452

FUNC                                    COUNT
get_page_from_freelist                 586241
[...]
```

The function was called over half a million times per second.

This works by using dynamic tracing of the function: It uses kprobes for kernel functions and uprobes for user-level functions (kprobes and uprobes are explained in Chapter 2). The overhead of this tool is relative to the rate of the functions. Some functions, such as malloc() and get_page_from_freelist(), tend to occur frequently, so tracing them can slow down the target application significantly, in excess of 10 percent—use caution. See Section 18.1 in Chapter 18 for more about understanding overhead.

Command line usage:

```
funccount [options] [-i interval] [-d duration] pattern
```

Options include:

- **`-r`:** Use regular expressions for the pattern match
- **`-p PID`:** Measures this process only

Patterns:

- ***`name`*** or ***`p:name`*:** Instrument the kernel function called *name*()
- ***`lib:name`*:** Instrument the user-level function called *name*() in library *lib*
- ***`path:name`*:** Instrument the user-level function called *name*() in the file at *path*
- **`t:`*`system`*`:`*`name`*:** Instruments the tracepoint called *system*:*name*
- **`*`:** A wildcard to match any string (globbing)

See Section 4.5 in Chapter 4 for more examples.

bpftrace

The core functionality of funccount(8) can be implemented as a bpftrace one-liner:

```
# bpftrace -e 'k:tcp_* { @[probe] = count(); }'
Attaching 320 probes...
[...]
@[kprobe:tcp_release_cb]: 153001
@[kprobe:tcp_v4_md5_lookup]: 154896
@[kprobe:tcp_gro_receive]: 177187
```

This can be adjusted to do per-interval output, for example, with this addition:

```
interval:s:1 { print(@); clear(@); }
```

As with BCC, use caution when tracing frequent functions, as they may incur significant overhead.

6.3.13 softirqs

softirqs(8) is a BCC tool that shows the time spent servicing soft IRQs (soft interrupts). The system-wide time in soft interrupts is readily available from different tools. For example, mpstat(1) shows it as %soft. There is also /proc/softirqs to show counts of soft IRQ events. The BCC softirqs(8) tool differs in that it can show time per soft IRQ rather than event count.

For example, from a 48-CPU production instance and a 10-second trace:

```
# softirqs 10 1
Tracing soft irq event time... Hit Ctrl-C to end.

SOFTIRQ          TOTAL_usecs
net_tx                   633
tasklet                30939
rcu                   143859
sched                 185873
timer                 389144
net_rx               1358268
```

This output shows that the most time was spent servicing net_rx, totaling 1358 milliseconds. This is significant, as it works out to be 3 percent of the CPU time on this 48-CPU system.

softirqs(8) works by using the irq:softirq_enter and irq:softirq_exit tracepoints. The overhead of this tool is relative to the event rate, which could be high for busy production systems and high network packet rates. Use caution and check overhead.

Command line usage:

```
softirqs [options] [interval [count]]
```

Options include:

- **-d:** Shows IRQ time as histograms
- **-T:** Includes timestamps on output

The -d option can be used to explore the distribution and identify whether there are latency outliers while servicing these interrupts.

bpftrace

A bpftrace version of softirqs(8) does not exist, but could be created. The following one-liner is a starting point, counting IRQs by vector ID:

```
# bpftrace -e 'tracepoint:irq:softirq_entry { @[args->vec] = count(); }'
Attaching 1 probe...
^C

@[3]: 11
@[6]: 45
@[0]: 395
@[9]: 405
@[1]: 524
@[7]: 561
```

These vector IDs can be translated to the softirq names in the same way the BCC tool does this: by using a lookup table. Determining the time spent in soft IRQs involves tracing the irq:softirq_exit tracepoint as well.

6.3.14 hardirqs

hardirqs(8)[21] is a BCC tool that shows time spent servicing hard IRQs (hard interrupts). The system-wide time in hard interrupts is readily available from different tools. For example, mpstat(1) shows it as %irq. There is also /proc/interrupts to show counts of hard IRQ events. The BCC hardirqs(8) tool differs in that it can show time per hard IRQ rather than event count.

21 Origin: I first created this as inttimes.d on 28-Jun-2005, for printing time sums and intoncpu.d for printing histograms on 9-May-2005, which was based on intr.d from the "Dynamic Tracing Guide," Jan 2005 [Sun 05]. I also developed a DTrace tool to show interrupts by CPU but have not ported it to BPF since Linux has /proc/interrupts for that task. I developed this BCC version that does both sums and histograms on 20-Oct-2015.