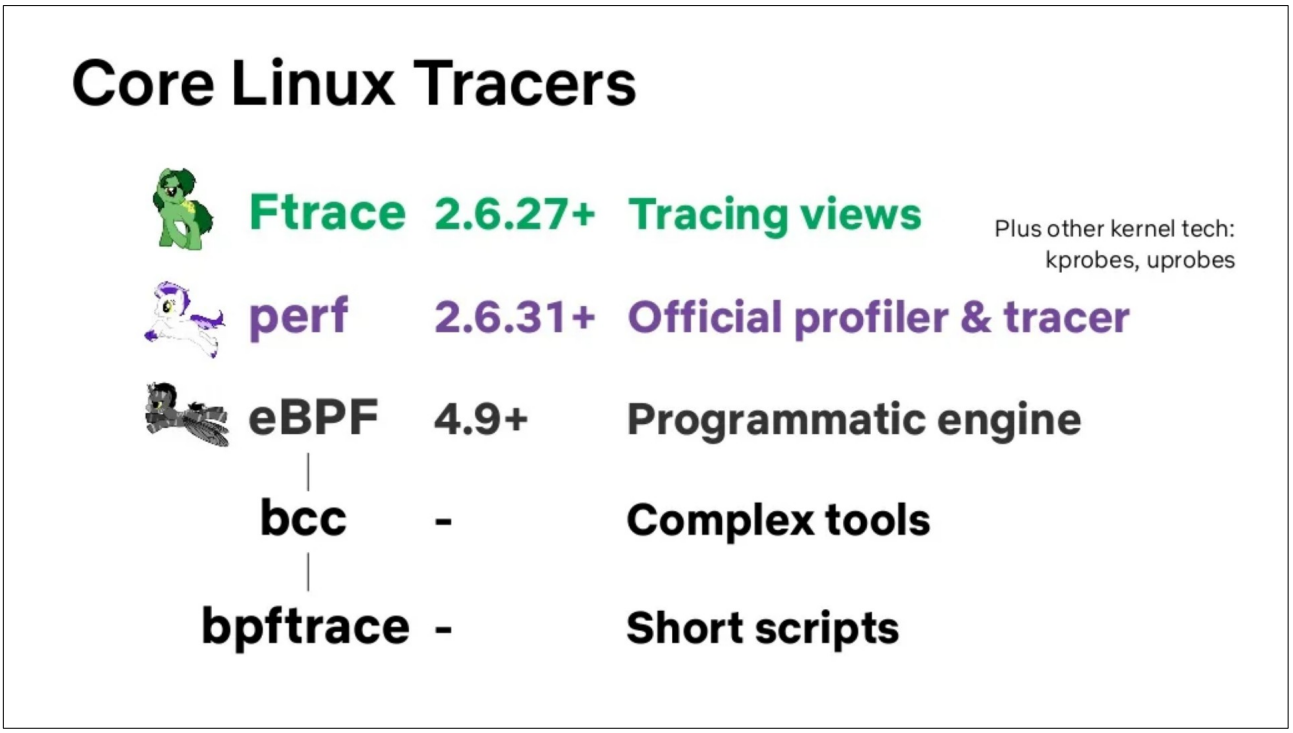
Tracing Tools – Perf, perf-tools, eBPF

Unlike profiling, tracing captures *everything. (As Brendan Gregg puts it:* tracers are like X-Ray machines …).

On Linux, there are many tracing technologies!



*Base technologies are **Ftrace, perf and eBPF**; all others are add-ons.*

From my Linux Kernel Debugging book, Packt, Aug 2022

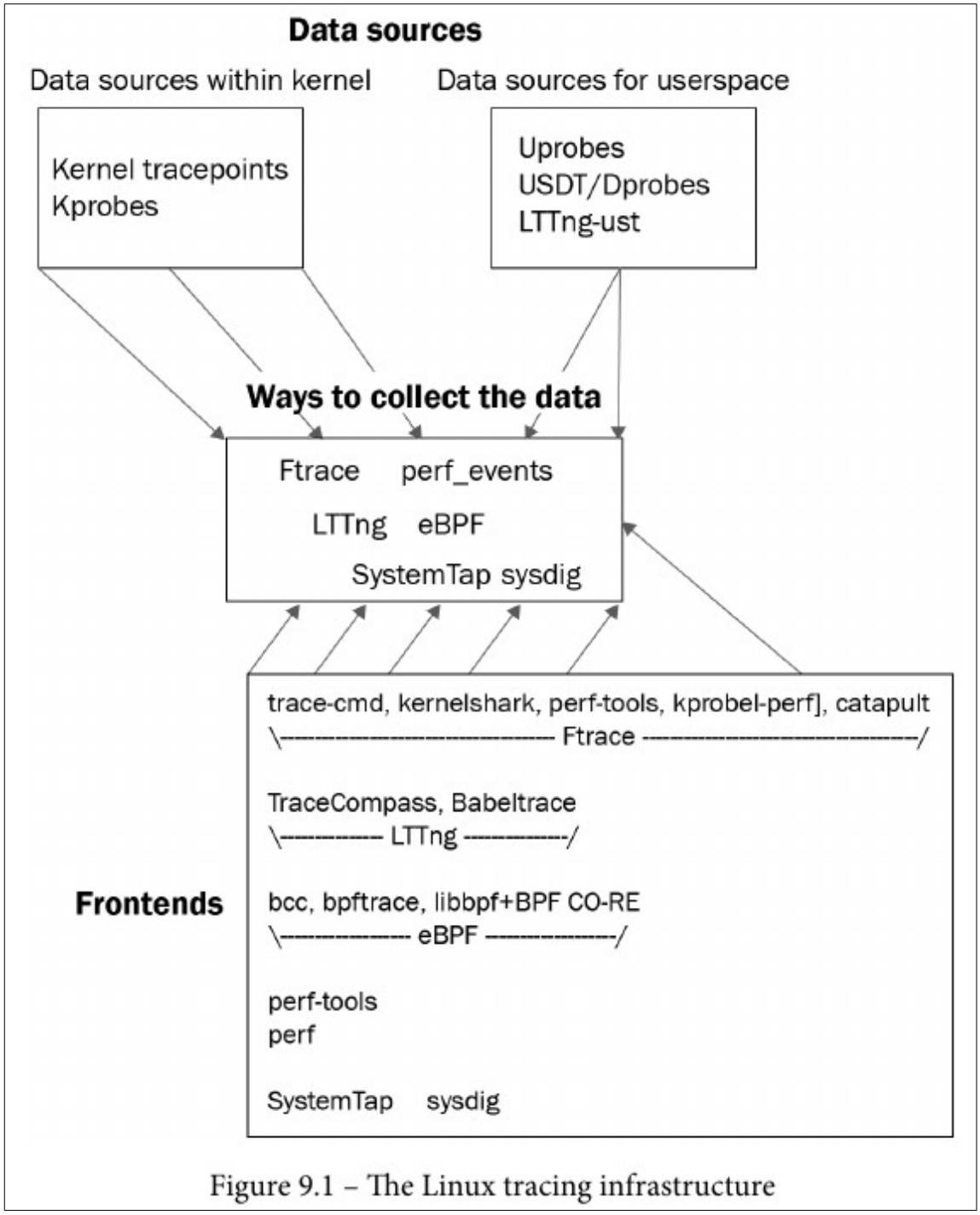


Figure 9.1 – The Linux tracing infrastructure

Ref: *Linux tracing systems & how they fit together*, *Julia Evans*

Data Sources

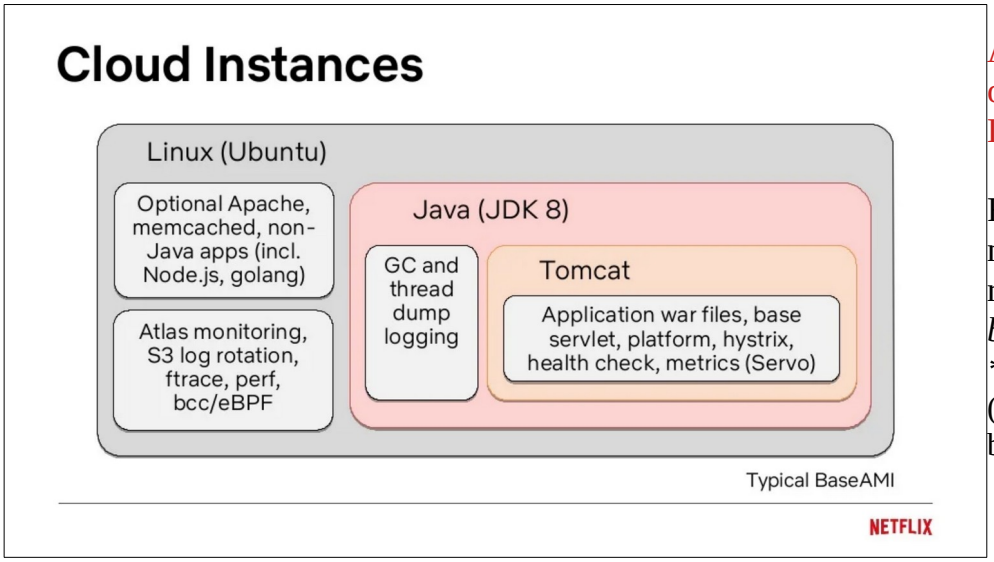
- Kernel
  - kprobes
    - *Note! Jprobes have been removed in 4.15 !*
  - tracepoints
  - dtrace probes (USTD)
- Userspace
  - uprobes
  - lttng-ust

Collecting Data from the kernel

- **Ftrace**
- perf_events: via the *perf_event_open(2)* system call
- eBPF
- sysdig
- LTTng
- SystemTap

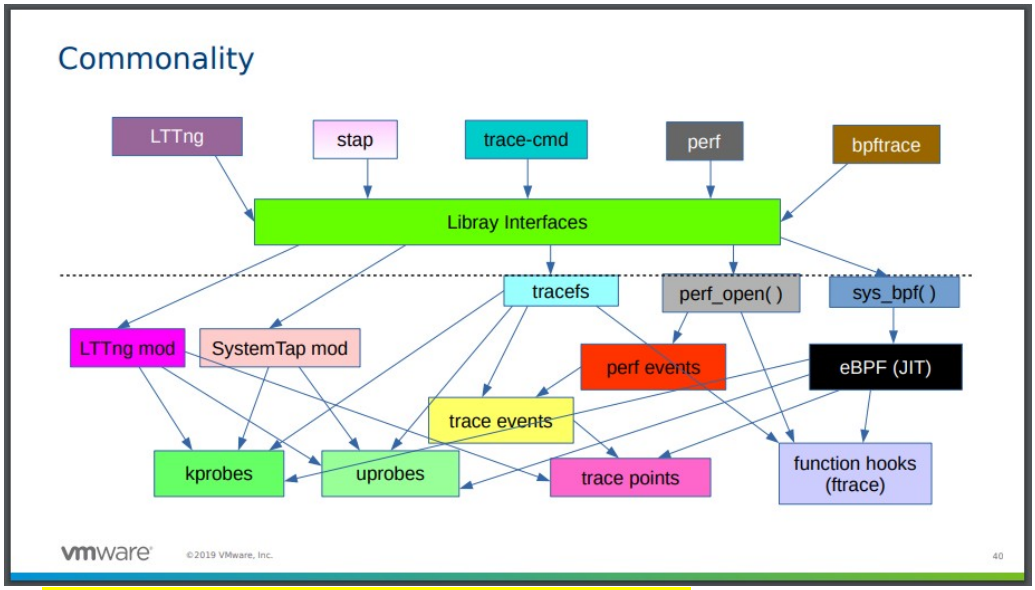
Tracing Infrastructure / Tool	*Front-end*
Ftrace	**trace-cmd**, catapult, kernelshark (gui), perf [trace], *perf-tools* using Ftrace by Brendan Gregg
Perf	perf trace
eBPF	bcc
LTTng	Trace Compass (gui)

<<
Q> Okay, great, but can we use these in production?
A> Yes! Several are lightweight enough to do so!
A good example is from Netflix's AWS cloud (this slide's dt ~ 2018):



As of 2018, Netflix has over ~ 150k Ubuntu Linux instances!

For observability, for metrics, *each* of them runs *Ftrace*, *perf*, *bpfcc/bcc*, *ftrace*, *sar*, **stat*, etc ...
(Look at the lower-left box in the diagram.)



Credit: *Unified Tracing Platform, Bringing tracing together*, Steven Rostedt, VMware 2019:
https://static.sched.com/hosted_files/osseu19/5f/unified-tracing-platform-oss-eu-2019.pdf

Eg. perf-trace: view *all* system calls as they are issued with low overhead:

```
$ sudo perf trace
         ? (     ?   ): dhclient/858  ... [continued]: select()) = 0 Timeout
     0.103 ( 0.016 ms): dhclient/858 getpid() = 858
     0.140 ( 0.069 ms): dhclient/858 sendto(fd: 3<socket:[17092]>, buff: 0x5651ecef59b0,
len: 113, flags: NOSIGNAL) = 113
     0.226 ( 0.415 ms): dhclient/858 sendto(fd: 6<socket:[17117]>, buff: 0x5651eceeef40,
len: 300, addr: 0x7fff20f67590, addr_len: 16) = 300
     0.696 ( 0.017 ms): dhclient/858 select(n: 7, inp: 0x7f414a8e9110, outp: 0x7f414a8e9210,
tvp: 0x7fff20f676c0) = 0 Timeout

--snip--

     1.059 ( 0.010 ms): systemd-journa/1915 fstat(fd: 26, statbuf: 0x7ffc365a6480
) = 0
     1.084 ( 0.038 ms): systemd-journa/1915 read(fd: 26, buf: 0x563d0b5c6540, count: 1024
) = 299
     1.144 ( 0.009 ms): systemd-journa/1915 close(fd: 26
) = 0

--snip--

   832.015 ( 0.231 ms): sshd/2992  ... [continued]: select()) = 1
   832.254 ( 0.004 ms): sshd/2992 rt_sigprocmask(how: BLOCK, nset: 0x7ffcbe7ae510, oset:
0x7ffcbe7ae490, sigsetsize: 8) = 0
LOST 23 events!
   832.254 ( 0.284 ms): sshd/2992  ... [continued]: select()) = 1
   832.547 ( 0.004 ms): sshd/2992 rt_sigprocmask(how: BLOCK, nset: 0x7ffcbe7ae510, oset:
0x7ffcbe7ae490, sigsetsize: 8) = 0
   832.555 ( 0.004 ms): sshd/2992 rt_sigprocmask(how: SETMASK, nset: 0x7ffcbe7ae490,
sigsetsize: 8      ) = 0
   832.564 ( 0.004 ms): sshd/2992 clock_gettime(which_clock: BOOTTIME, tp: 0x7ffcbe7ae530
) = 0
LOST 18 events!
```

```
   832.573 ( 0.305 ms): sshd/2992 read(fd: 11</dev/ptmx>, buf: 0x7ffcbe7aa470, count: 16384
) = 1
   832.885 ( 0.004 ms): sshd/2992 rt_sigprocmask(how: BLOCK, nset: 0x7ffcbe7ae510, oset:
0x7ffcbe7ae490, sigsetsize: 8) = 0
   832.894 ^C( 0.004 ms): sshd/2992 rt_sigprocmask(how: SETMASK, nset: 0x7ffcbe7ae490,
sigsetsize: 8      ) = 0
$
```

USEFUL!

Ftrace Favorites Cheat Sheet - Fun Commands to Try with Ftrace
Kernel Tracing Cheat Sheet
http://www.brendangregg.com/linuxperf.html
https://github.com/goldshtn/linux-tracing-workshop [tutorial: try various tracing tools]

Using Brendan Gregg's superb perf-tools collection

https://github.com/brendangregg/perf-tools

Bash script wrappers over perf and ftrace !

Q. How do I install the package?
A. Try this (on a Debian-based Linux; below o/p from a Raspberry Pi 4 running Raspberry Pi OS):

```
$ sudo apt update
$ grep "^Package:" /var/lib/apt/lists/* | cut -d":" -f3 | grep " *perf"
...
 netperfmeter
 node-performance-now
 perf-tools-unstable
 perforate
 performous
...
$
```

Install it:

```
$ sudo apt install perf-tools-unstable
```

Check what got installed:

```
$ dpkg -L perf-tools-unstable
/.
/usr
/usr/sbin
/usr/sbin/bitesize-perf
/usr/sbin/cachestat-perf
/usr/sbin/execsnoop-perf
/usr/sbin/funccount-perf
/usr/sbin/funcgraph-perf
/usr/sbin/funcslower-perf
/usr/sbin/functrace-perf
/usr/sbin/iolatency-perf
/usr/sbin/iosnoop-perf
/usr/sbin/killsnoop-perf
/usr/sbin/kprobe-perf
/usr/sbin/opensnoop-perf
/usr/sbin/perf-stat-hist-perf
/usr/sbin/reset-ftrace-perf
```

```
/usr/sbin/syscount-perf
/usr/sbin/tcpretrans-perf
/usr/sbin/tpoint-perf
/usr/sbin/uprobe-perf
/usr/share
...
$
```

Tips

- run on a Linux system with a ‘distro’ kernel as far as is possible; avoids issues due to kernel headers or the underlying linux-perf (‘perf’ utility) being unavailable
- some of the <foo>-perf utils require *perf* to be installed

<<
Workaround on my R Pi
Created a soft link in /usr/bin/ to the *perf* util – called perf_4.9; named the soft link perf:

```
rpi4 # ls -lhF perf*
lrwxrwxrwx 1 root root    8 Feb 16 13:48 perf -> perf_4.9
-rwxr-xr-x 1 root root 1.7M Nov  9  2018 perf_4.9
```
>>

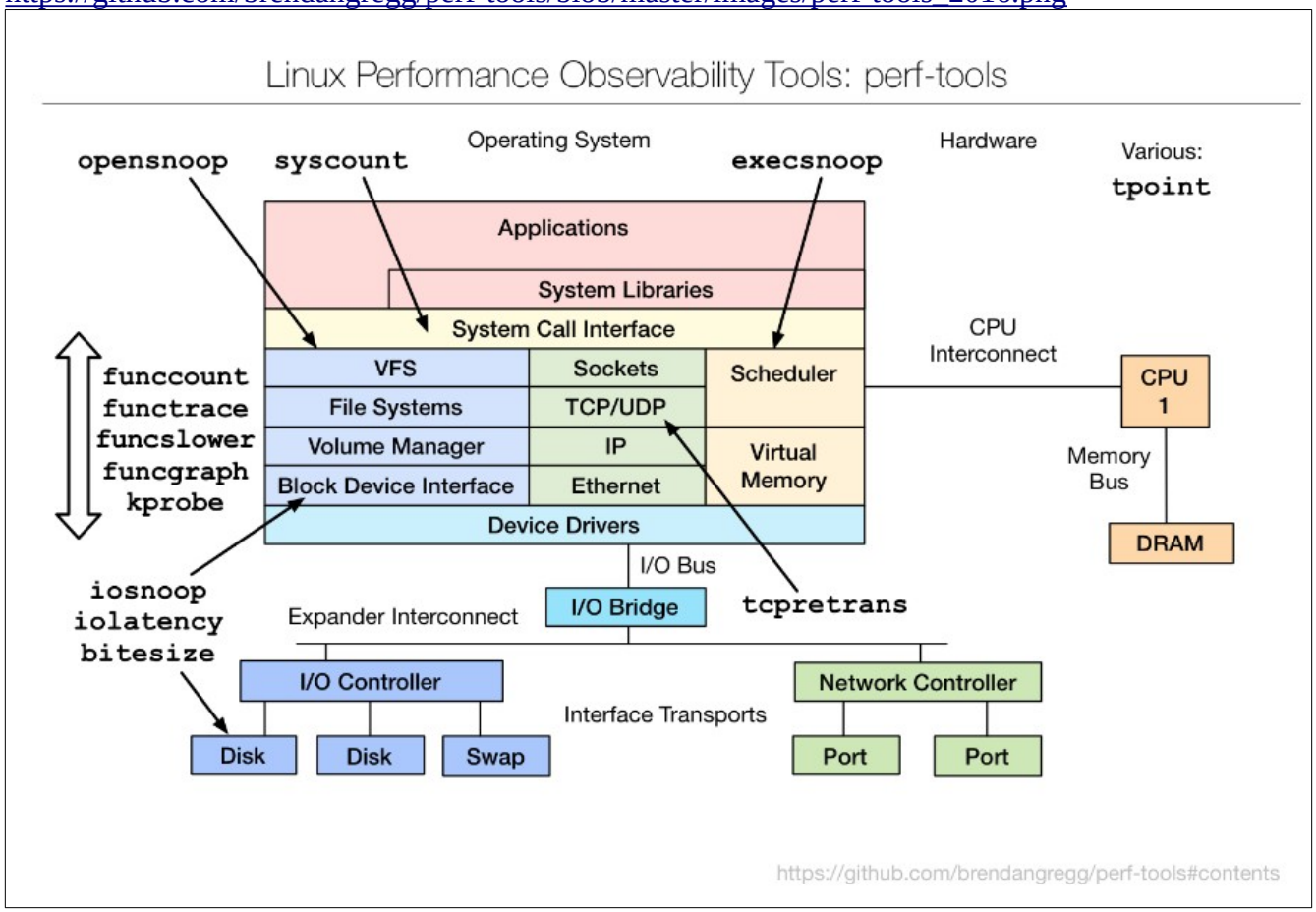
<<
The *kprobe-perf* utility from *perf-tools* uses the Ftrace kernel subsystem to kprobe any (allowed) functions. Allowed functions are visible in */sys/kernel/debug/tracing/available_filter_functions* .

The perf-tools utils:

```
$ ls /usr/sbin/*-perf |sed 's/\/usr.sbin.//g'
bitesize-perf
cachestat-perf
execsnoop-perf
funccount-perf
funcgraph-perf
funcslower-perf
functrace-perf
iolatency-perf
iosnoop-perf
killsnoop-perf
kprobe-perf
opensnoop-perf
perf-stat-hist-perf
reset-ftrace-perf
syscount-perf
tcpretrans-perf
tpoint-perf
uprobe-perf
$
```
>>

Each of them has a man page too! Do refer it, good examples are given there.

https://github.com/brendangregg/perf-tools/blob/master/images/perf-tools_2016.png



Online: better, more detailed with some explanation as well!
https://github.com/brendangregg/perf-tools/tree/master/examples

Below, I use my '*veth' (virtual ethernet)* network driver as an example:

```
$ ./test_veth.sh 1
Loading up the veth NIC driver & bringing up the interface now ...
make -C /lib/modules/4.13.0-43-generic/build
SUBDIRS=/home/seawolf/kaiwanTECH/harman_L2_L5/src/veth_stuff/veth_drv  clean
make[1]: Entering directory '/usr/src/linux-headers-4.13.0-43-generic'
Building with ARCH=x86, KERNELRELEASE=4.13.0-43-generic, CROSS_COMPILE=
[...]

sudo insmod veth.ko
veth                   20480  0
dmesg ========================= [start]
[ 3434.914332] vnet_init:621 : veth: Initializing network driver...
[ 3434.934054] vnet_probe:527 :
[ 3434.938342] setup_debugfs_entries:26 :
```

```
[ 3434.943931] veth ver 0.2 loaded.
[ 3434.981933] vnet_open:474 :
dmesg ========== [end]
veth: flags=4227<UP,BROADCAST,NOARP,MULTICAST>  mtu 1500
        inet 10.10.1.5  netmask 255.0.0.0  broadcast 10.255.255.255
        inet6 fe80::4a0f:eff:fe0d:a02  prefixlen 64  scopeid 0x20<link>
        ether 48:0f:0e:0d:0a:02  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0


1 packet captured
1 packet received by filter
0 packets dropped by kernel
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
./talker_dgram: Sending message over datagram socket to 10.10.1.16:54295 now...
    1  2018-06-02 13:00:00.557217 Out 48:0f:0e:0d:0a:02 ethertype IPv4 (0x0800), length 66: (tos
0x0, ttl 64, id 41673, offset 0, flags [DF], proto UDP (17), length 50)
    10.10.1.5.35070 > 10.10.1.16.54295: UDP, length 22
       0x0000:  0004 0001 0006 480f 0e0d 0a02 0000 0800  ......H.........
       0x0010:  4500 0032 a2c9 4000 4011 81c9 0a0a 0105  E..2..@.@.......
       0x0020:  0a0a 0110 88fe d417 001e 44b2 4865 7920  ..........D.Hey.
       0x0030:  6275 6464 792c 2066 726f 6d20 7461 6c6b  buddy,.from.talk
       0x0040:  6572                                     er
    2  2018-06-02 13:00:00.853625  In 00:00:a0:7b:c5:9c ethertype IPv4 (0x0800), length 80:
truncated-ip - 28 bytes missing! (tos 0x0, ttl 64, id 64, offset 0, flags [none], proto UDP (17),
length 92)
    10.10.1.53.54295 > 10.10.1.5.54295: UDP, length 22
       0x0000:  0000 0001 0006 0000 a07b c59c 0000 0800  .........{......
       0x0010:  4500 005c 0040 0000 4011 6404 0a0a 0135  E..\.@..@.d....5
       0x0020:  0a0a 0105 d417 d417 001e 2421 4865 7920  ..........$!Hey.
       0x0030:  6275 6464 792c 2066 726f 6d20 7461 6c6b  buddy,.from.talk
       0x0040:  6572 0000 0000 0000 0000 0000 0000 0000  er..............
2 packets captured
2 packets received by filter
0 packets dropped by kernel
./talker_dgram: sent 22 bytes.
$


# kprobe-perf -s 'p:myprobe vnet_start_xmit'  << setup a kprobe on vnet_start_xmit
                                                 and show the stack >>
Tracing kprobe myprobe. Ctrl-C to end.
  << Once the transmit routine gets invoked via kernel protocol stack… >>
          <idle>-0     [000] d.s.  3446.384019: myprobe: (vnet_start_xmit+0x0/0x2e0 [veth])
          <idle>-0     [000] d.s.  3446.384039: <stack trace>
 => vnet_start_xmit
 => sch_direct_xmit
 => __dev_queue_xmit
 => dev_queue_xmit
 => ip6_finish_output2
 => ip6_finish_output
 => ip6_output
 => NF_HOOK.constprop.22
 => ndisc_send_skb
 => ndisc_send_rs
```

```
 => addrconf_rs_timer
 => call_timer_fn
 => run_timer_softirq
 => __do_softirq
 => irq_exit
 => smp_apic_timer_interrupt
 => apic_timer_interrupt
 => mwait_idle
 => arch_cpu_idle
 => default_idle_call
 => do_idle
 => cpu_startup_entry
 => rest_init
 => start_kernel
 => x86_64_start_reservations
 => x86_64_start_kernel
 => secondary_startup_64
    talker_dgram-9518  [000] dN..  3448.687974: myprobe: (vnet_start_xmit+0x0/0x2e0 [veth])
    talker_dgram-9518  [000] dN..  3448.687993: <stack trace>
 => vnet_start_xmit
 => sch_direct_xmit
 => __dev_queue_xmit
 => dev_queue_xmit
 => ip_finish_output2
 => ip_finish_output
 => ip_output
 => ip_local_out
 => ip_send_skb
 => udp_send_skb
 => udp_sendmsg
 => inet_sendmsg
 => sock_sendmsg
 => SYSC_sendto
 => SyS_sendto
 => entry_SYSCALL_64_fastpath
          <idle>-0     [000] d.s.  3460.717515: myprobe: (vnet_start_xmit+0x0/0x2e0 [veth])
          <idle>-0     [000] d.s.  3460.717535: <stack trace>
 => vnet_start_xmit
 => sch_direct_xmit
 => __dev_queue_xmit
 => dev_queue_xmit
 => ip6_finish_output2
 => ip6_finish_output
 => ip6_output
 => NF_HOOK.constprop.22
 => ndisc_send_skb
 => ndisc_send_rs
 => addrconf_rs_timer
 => call_timer_fn
 => run_timer_softirq
 => __do_softirq
 => irq_exit
 => smp_apic_timer_interrupt
 => apic_timer_interrupt
 => mwait_idle
[...]
```

eBPF

An extract from my book: Linux Kernel Programming, 2nd Ed, Packt, Feb 2024
From the Online Chapter 1
<<

Online Ch 1
Modern tracing and performance analysis with [e]BPF

An extension of the well-known Berkeley Packet Filter, or BPF, is eBPF, the extended BPF. (FYI, at times it's referred to simply as BPF, dropping the "e" prefix; here we'll explicitly use the term eBPF.) Very briefly, BPF used to provide the supporting infrastructure within the kernel to effectively trace network packets. eBPF is a relatively recent kernel innovation – available only from the Linux 4.0 kernel onward. It extends the BPF notion, allowing you to trace much more than just the network stack. *eBPF is essentially virtual machine technology, allowing one to write (small) programs and run them in a safe, isolated environment within the kernel.* In effect, eBPF and its frontends are a really modern and powerful approach to observability, tracing, performance analysis, and more on Linux systems, even in production.

To use eBPF, you will need a system with the following:

- Linux kernel 4.0 or later
- Kernel support for BPF (https://github.com/iovisor/bcc/blob/master/INSTALL.md#kernel-configuration)
- The BCC or bpftrace frontends installed (link to install them on popular Linux distributions: https://github.com/iovisor/bcc/blob/master/INSTALL.md#installing-bcc)
- Root access on the target system

Using the eBPF kernel feature directly is very hard, so there are several easier frontends to use. Among them, BCC and bpftrace are regarded as very useful. Check out the following link to a picture that opens your eyes to just how many powerful BCC tools are available to help trace different Linux subsystems and hardware:
https://github.com/iovisor/bcc/blob/master/images/bcc_tracing_tools_2019.png.

You can install the BCC tools for your regular host or native Linux distro by reading the installation instructions here: https://github.com/iovisor/bcc/blob/master/INSTALL.md.

Info-box:
Why not on our guest Linux VM? You can, when running a distro kernel (such as an Ubuntu- or Fedora-supplied kernel). The reason: the installation of the BCC toolset includes (and depends upon) the installation of the *linux-headers-$(uname -r)* package; this linux-headers package exists only for distro kernels (and not for our custom 6.1 kernel that we shall often be running on the guest). There is a workaround: building the kernel with the kernel config CONFIG_IKHEADERS=m (you'll learn how to do this in *Chapter 2, Building the 6.x Linux Kernel from Source – Part 1*). In fact, part of the error message from the eBPF tooling spells this out: "*Unable to find kernel headers. Try rebuilding kernel with CONFIG_IKHEADERS=m (module) or installing the kernel development package for your running kernel version.*"

The main site for BCC can be found at https://github.com/iovisor/bcc. We shall make use of a bit of the eBPF tooling in some later chapters.

FYI, my book *Linux Kernel Debugging*, covers using LTTng, Trace Compass, KernelShark, ftrace, trace-cmd, procmap, and many more debugging tools and techniques in depth.
>>

Once installed, man pages are present (with examples!).

Similar to how we did with perf (again, on my x86_64 Ubuntu 20.04 system):

```
$ dpkg -l|grep bpf
ii  bpfcc-tools                                  0.12.0-2
all          tools for BPF Compiler Collection (BCC)
ii  libbpfcc                                    0.12.0-2
amd64        shared library for BPF Compiler Collection (BCC)
...
```

Many ‘bpfcc’ utils get installed (under /usr/sbin):

```
$ ls /usr/sbin/*-bpfcc |sed 's/\/usr.sbin.//g'
argdist-bpfcc*
bashreadline-bpfcc*
biolatency-bpfcc*
biosnoop-bpfcc*
biotop-bpfcc*
bitesize-bpfcc*
...
...
xfsslower-bpfcc*
zfsdist-bpfcc*
zfsslower-bpfcc*
$
```

Examples can be found here!

Measuring scheduler latency via modern BPF tools

Without going into too many details, we'd be amiss to leave out the recent and powerful [e]BPF Linux kernel feature and it's associated frontends; there are a few to specifically measure scheduler and runqueue-related system latencies. (We covered the installation of the [e]BPF tools back in `Chapter 1`, *Kernel Workspace Setup* under the *Modern tracing and performance analysis with [e]BPF* section).

The following table summarizes some of these tools (BPF frontends); all these tools need to be run as root (as with any BPF tool); they show their output as a histogram (with the time in microseconds by default):

BPF tool	What it measures
`runqlat-bpfcc`	Time a task spends waiting on a runqueue for it's turn to run on the processor
`runqslower-bpfcc`	(read as runqueue slower); time a task spends waiting on a runqueue for it's turn to run on the processor, showing only those threads that exceed a given threshold, which is 10 ms by default (can be tuned by passing the time threshold as a parameter, in microseconds); in effect, you can see which tasks face (relatively) long scheduling delays
`runqlen-bpfcc`	Shows scheduler runqueue length + occupancy (number of threads currently enqueued, waiting to run)

The tools can also provide these metrics on a per-task basis, for every process on the system or even by PID namespace (for container analysis; of course, these options depend on the tool in question). Do look up more details (and even example usage!) from the man pages (section 8) on these tools.

Above: an extract from my book: Linux Kernel Programming, Packt, Mar 2021 [link]

Eg.

```
$ runqslower-bpfcc -h
usage: runqslower-bpfcc [-h] [-p PID] [min_us]

Trace high run queue latency

positional arguments:
  min_us             minimum run queue latecy to trace, in ms (default 10000)

optional arguments:
  -h, --help         show this help message and exit
  -p PID, --pid PID  trace this PID only

examples:
    ./runqslower         # trace run queue latency higher than 10000 us (default)
    ./runqslower 1000    # trace run queue latency higher than 1000 us
```

```
    ./runqslower -p 123  # trace pid 123 only
$

$ sudo runqslower-bpfcc 1000
[ ... ]
Tracing run queue latency higher than 1000 us
TIME     COMM             PID           LAT(us)
13:24:09 b'snapd'         1337             1256
13:24:11 b'chrome'        4136657           1053
13:24:11 b'chrome'        3869             1107
13:24:11 b'chrome'        4109284           1330
...
```
