Practical Linux tracing with bpftrace

Outline

- What is Linux tracing and bpftrace ?
- Why do we need it ?
- When should we use it ?
- How to use it ?

What is Linux tracing ?

Trace as an English verb

to discover the causes or origins of something by examining the way in which it has developed

(https://dictionary.cambridge.org/dictionary/english/trace)

Trace in Linux context

To find a root cause of software problem in Linux by examining:

- Runtime statistics / events
- Code execution path
- Source code

Tracing tools:

- Provide info about software events and execution path
- Syscall tracing: strace
- Kernel function tracing: ftrace
- Multi-purposes tracing: bcc-tools, bpftrace

Why bpftrace ?

A tool to dig deeper

- Which code is being executed
- How is it being executed ? (Frequency, stack traces, latency)
- What is the status of that code ? (Retval, Params, Local vars)
- When does it start and end ? (Latency, security audit)

As a performance analysis tool

- Easy to use one-liners
- Programmable (awk like syntax)
- Low overhead
- Powerful: function-level metrics, in-kernel data processing

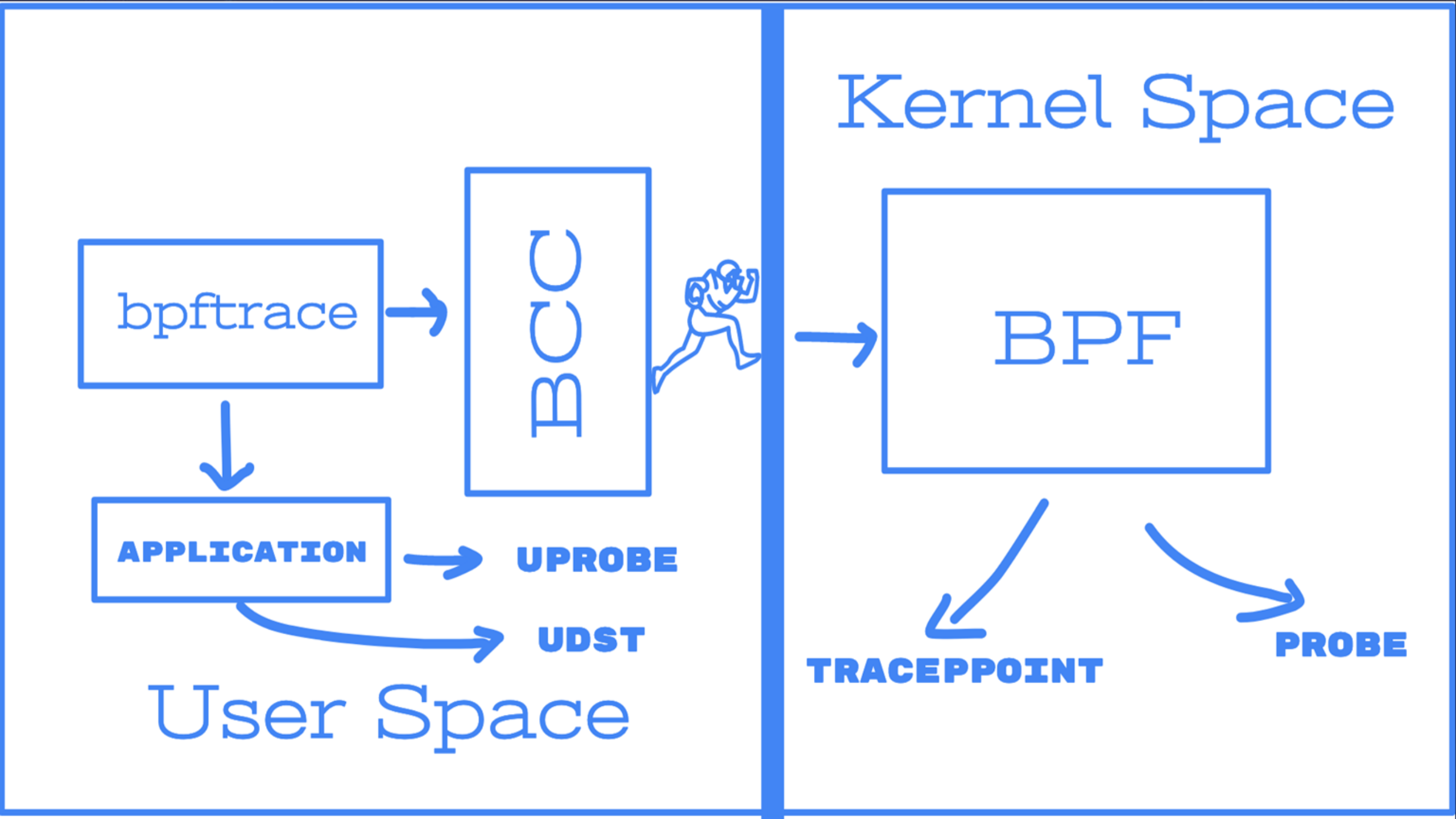
Other lesser-known application

- Continuous monitoring (Netflix Vector - PCP , Cloudflare ebpf-exporter)
- Source code analysis
- Container tracing with kubectl-trace

What is bpftrace ?

What is bpftrace ?

- High-level tracing language
- Track and process Linux tracing events using BCC and eBPF
- eBPF: enhanced Berkeley Packet Filter (in-kernel virtual machine)
- BCC: BPF compiler collections


Kernel Space
bpftrace
BCC
BPF
APPLICATION
UPROBE
UDST
TRACEPPOINT
PROBE
User Space

Practical example

Opensnoop: the beauty of event tracing

- Purpose: tracing opening files
- Useful case: catching short-lived IO
- Practical examples:
  - nginx writes proxy temp data to cache dir
  - mysql writes exceeded buffers queries to tmpdir
  - tracking progress while cloning big git repo

What actually is opensnoop ?

- bpftrace program
- Tracking enter / exit event of open() and openat() syscalls
- No (huge number of) objects tracking required
- Parsing argument to get filename, return code
- Provide more context info by built-in feature / values: key-value storage, tid, pid, comm

On-the-fly TCP backlog tracing

- Purpose: show current backlog value of a specific socket
- Why do we need it ?
  - verify concurrency saturation issue (while there's still plenty of free resource !)
  - reveal random latency spike issue

Queue exceeding triggers TCP retransmit with exponential backoff

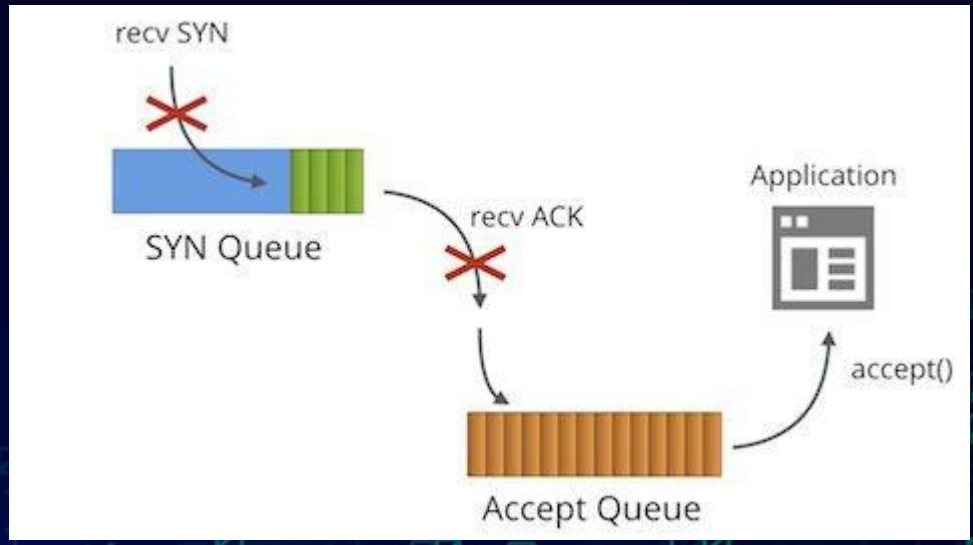
Which socket's queue is currently full ?

How many times it was full ?

What was the queue length that time ?

The tale of two queues and slow application

(https://blog.cloudflare.com/syn-packet-handling-in-the-wild/)



max_backlog bpftrace program

- A tiny tweak from tcpsynbl.bt
- Tracing tcp_v4_syn_recv_sock kernel function
- Parse first argument (struct sock *) for socket info
- Compare current backlog vs max backlog
- Report if exceeds

On-the-fly TCP backlog tracing (cont)

Practical example:

- Redis low somaxconn inside container
- Busy mongodb with low listenBacklog
- Simple health check failure during highload

When do you need bpftrace ?

- When you know what to trace, ok do it anyway

- Otherwise, only try bpftrace after using:
  - Monitoring dashboards
  - Traditional tshoot toolkit: *top, *stat
  - Application / kernel logs

How can we use bpftrace ?

Requirements to meet

- Linux kernel version >= 4.9 (for full feature)
- Glibc >= 2.23
- Root access
- Source code access

Basic concepts to understand

- Tracing / profiling
- Symbols / Debug symbols
- Stack trace
- Tracing sources
- Very basic C: function, params, struct dereference, type cast

Tracing sources

- Dynamic tracing: trace mark are added / removed on demand (uprobe / uretprobe / kprobe / kretprobe)
- Static tracing: hard-coded tracepoints, very stable to use (tracepoint / UDST)
- Profiles: perf_events, taking snapshots of the whole system on fixed frequency

Advance concepts to understand

Linux kernel abstractions:

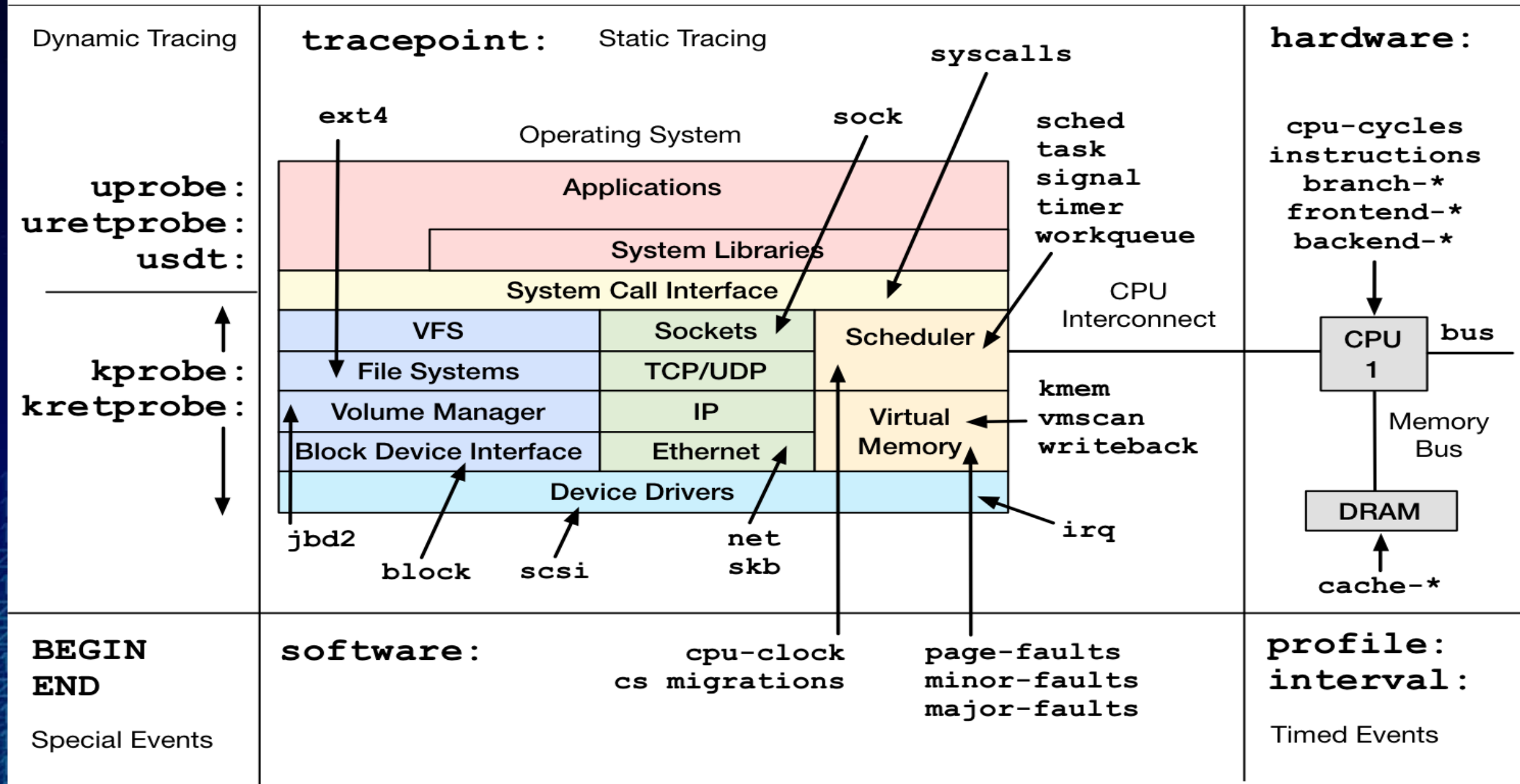
- Process , thread and CPU scheduling
- Virtual memory management
- Filesystem
- TCP implementations

Program execution path

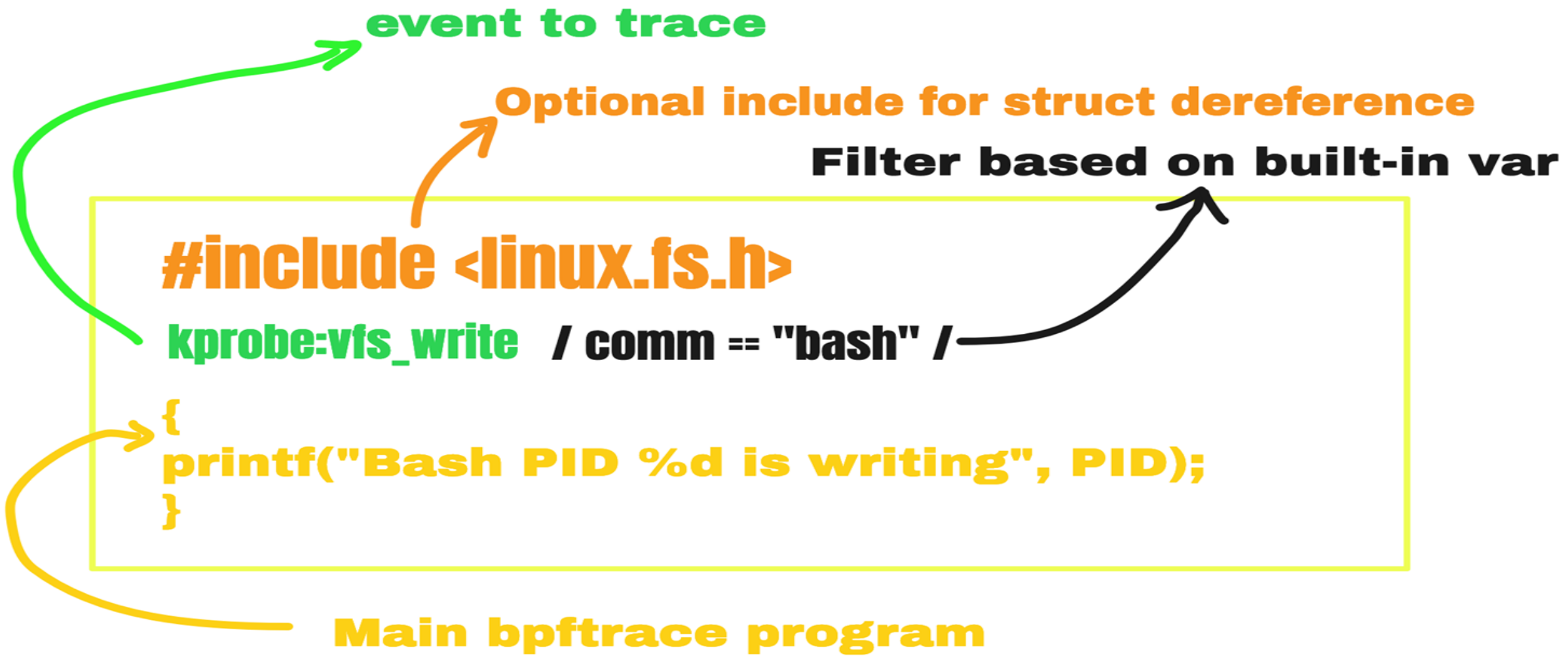
- Standard libc bootstrap
- Application code
- Library code
- Syscall
- Kernel function

Put them all together

bpftrace Probe Types



Bpftrace language


event to trace
Optional include for struct dereference
Filter based on built-in var
#include <linux.fs.h>
kprobe:vfs_write / comm == "bash" /
{
printf("Bash PID %d is writing", PID);
}
Main bpftrace program

bpftrace built-in objects for getting started

- Built-in vars for filtering: pid, tid, comm, nsecs, argX, sargX
- Basic declaration: $ for normal var, @ for map (k/v) object
- Functions: ustack, kstack, printf
- Map functions: count, hist, avg, max, min…
- Type cast: int(), str(), (struct custom_type *) value
- See bpftrace reference guide for much more info

Write your first bpftrace program

- function to work on: vfs_write
- filter based on comm built-in var
- Track input string
- Store more identifiers: pid, uid, comm, nsecs
- Dummy remote-keylogger

References

1. Bpftrace reference guide
2. BPF performance tools book
3. Syn packet handling in the wild
4. CocCoc Engineering Blog
5. Brendan Gregg’s blog
6. Julia Evans’ post about Linux tracing systems
7. How TCP backlog works
8. IO visor project
9. awesome-ebpf

THANK YOU !

cốc cốc 2020 SYSCON