bpftrace Dynamic kernel tracing

START HERE

`bpftrace -l '<pattern>'`
Find probes

`bpftrace -lv '<probe>'`
Show probe arguments

`bpftrace -e '<program>'`
Run a one-liner

`bpftrace <file.bt>`
Run a script

`bpftrace -p <pid> -e '<program>'`
Trace one process

`bpftrace -c '<cmd>' -e '<program>'`
Trace a command

`@[comm] = count()`
Count events per process

`Ctrl+C`
Stop and print maps

Probe kernel or user event → **fires** → **Filter** / predicate / → **passes** → **Action** { ... } block → **updates** → **Map** @name → **on exit** → **Output** printed on Ctrl+C

CONCEPTS

Needs root (or CAP_BPF/ CAP_PERFMON) and a kernel with BPF. Maps you fill with @ are printed automatically when the program ends.

A program is a list of probe / filter/ { action } clauses. Docs for current releases write begin and end in lowercase and use args.field; older scripts use BEGIN/END and args→field.

Probes in a one-liner are cheap to attach, but a hot probe (every syscall, every packet) adds real overhead. Filter by pid or comm first.

RUNNING

`bpftrace -e '<program>'`
Run an inline program

`bpftrace <file.bt>`
Run a script file

`bpftrace -l '<pattern>'`
List probes, wildcards allowed

`bpftrace -lv '<probe>'`
List a probe with its arguments
Also works for types: bpftrace -lv 'struct task_struct'.

`bpftrace -p <pid> -e '<program>'`
Attach to or filter by a PID

RUNNING (cont.)

`bpftrace -c '<cmd>' -e '<program>'`
Run a command and trace it
Exits when the command exits; the child PID is the cpid built-in.

`bpftrace -f json -e '<program>'`
Output as JSON lines

`bpftrace --unsafe -e '<program>'`
Allow unsafe calls such as system()

PROBES

`kprobe:<func>`
Kernel function entry

`kretprobe:<func>`
Kernel function return (retval)

`tracepoint:<cat>:<name>`
Static kernel tracepoint
Arguments are in args; list them with -lv.

`uprobe:<path>:<func>`
User function entry

`begin { ... } / end { ... }`
Run once at start / at exit

`uretprobe:<path>:<func>`
User function return

`usdt:<path>:<prov>:<name>`
User static tracepoint

`profile:hz:<rate>`
Sample on every CPU at a rate

PROBES (cont.)

`interval:s:<n>`
Fire once per interval, on one CPU

`fentry:<func>`
Typed kernel entry, args.<name> (BTF)

`kprobe:tcp_*`
Wildcard in a probe name

BUILT-INS

`pid / tid`
Process / thread ID

`comm`
Process name

`arg0 ... arg5`
Raw function arguments

`retval`
Return value in return probes

`nsecs`
Timestamp in nanoseconds

`args.<field>`
Tracepoint or fentry argument
Older releases write args→field.

`func`
Name of the traced function

`curtask`
Current task_struct pointer

MAPS AND AGGREGATION

`@ = count()`
Count events

MAPS AND AGGREGATION (cont.)

`@[<key>] = count()`
Count per key (comm, pid, probe)

`@[<key>] = sum(<n>)`
Sum a value per key

`@[<key>] = avg(<n>)`
Average per key

`@ = hist(<n>)`
Power-of-2 histogram

`@ = lhist(<n>, <min>, <max>, <step>)`
Linear histogram

`@[<k1>, <k2>] = count()`
Multi-part key

`@start[tid] = nsecs`
Remember a timestamp per thread
Pair with a return probe to compute latency.

`delete(@<map>, <key>)`
Remove one key

`clear(@<map>)`
Remove all entries

`print(@<map>, <n>)`
Print the top n entries

`len(@<map>)`
Number of entries

`has_key(@<map>, <key>)`
Does the key exist?

MAPS AND AGGREGATION (cont.)

Maps without a clear() or delete() are dumped at exit, including temporaries such as @start. Clear them in an end block.

OUTPUT AND FUNCTIONS

`printf("<fmt>\n", <args>)`
Formatted output

`str(<ptr>)`
Read a C string

`exit()`
Stop the program

`kstack(<n>)`
Kernel stack, top n frames

`ustack(<n>)`
User stack, top n frames

LANGUAGE

`probe /<predicate>/ { ... }`
Run the action only if true
Example: /pid == 1234/ or /comm == "nginx"/.

`$x = <value>;`
Scratch variable, local to the probe

`(struct <name> *)<ptr>`
Cast to a kernel struct

`for ($kv : @<map>) { ... }`
Loop over map entries

SCRIPTS AND CONFIG

`#!/usr/bin/env bpftrace`
Shebang for a runnable .bt file

`BPFTRACE_MAX_STRLEN=<n>`
Longer strings (env var)

Fields of kernel structs need BTF (/sys/kernel/btf/vmlinux). Without it, include the headers yourself.

RECIPES

Syscalls by process

```
t:raw_syscalls:sys_enter {
  @[comm] = count();
}
# Ctrl+C prints the counts
```

Files opened, live

```
t:syscalls:sys_enter_openat {
  printf("%s %s\n",
    comm, str(args.filename));
}
```

Read latency histogram (us)

```
kprobe:vfs_read { @start[tid] =
nsecs; }
kretprobe:vfs_read /
has_key(@start, tid)/ {
  @us = hist((nsecs -
@start[tid]) / 1000);
  delete(@start, tid);
}
end { clear(@start); }
```

CPU profile of a process

```
profile:hz:99 /pid == <pid>/ {
  @[ustack(8)] = count();
}
# kernel side: @[kstack(8)]
```